Chapter 8
A Framework for Distributed, Loosely-Synchronized Simulation of Complex SystemC/TLM Models

Christian Sauer, Hans-Martin Bluethgen, and Hans-Peter Loeb

Abstract Today's virtual prototypes model complex many-core platforms. In application domains such as network processing, they may comprise hundreds of processors, which makes simulation speed the key issue due to the single-threaded execution semantics of SystemC. We propose CoMix, the Concurrent Model Interface, for the distributed simulation of large-scale SystemC models. CoMix provides robust communication between simulator peers, enables their loose synchronization, and manages the overall life cycle. It is an overlay technology neither requiring modified simulators nor depending on a hosts' communication infrastructure. The CoMix framework is small (2k Lines of C++ Code) and easily deployable. We quantify its overhead on synthetic benchmarks and observe reasonable speedups for synthetic benchmarks as well as a large real-world example, e.g., 3.3X and 4X for a 4-peer simulation.

8.1 Introduction

Enabled by maturing standards, the availability of platform libraries, and wider tool support, SystemC (SC)-based simulation models are increasingly deployed early in the design cycle of System-on-Chip (SoC) platforms. Such models facilitate the development of embedded software for full, highly complex systems, as they abstract irrelevant details for faster simulation while providing sufficient insights into the interplay between software and hardware. This way, high-quality software can be developed sooner and more concurrently to a platform's hardware. In addition, these models may serve as entry points into exploration, design, and verification flows, because they capture the system intent in a functionally correct way [2].

C. Sauer (✉) • H.-M. Bluethgen • H.-P. Loeb
Cadence Design Systems, Munich, Germany
e-mail: sauerc@cadence.com

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_8

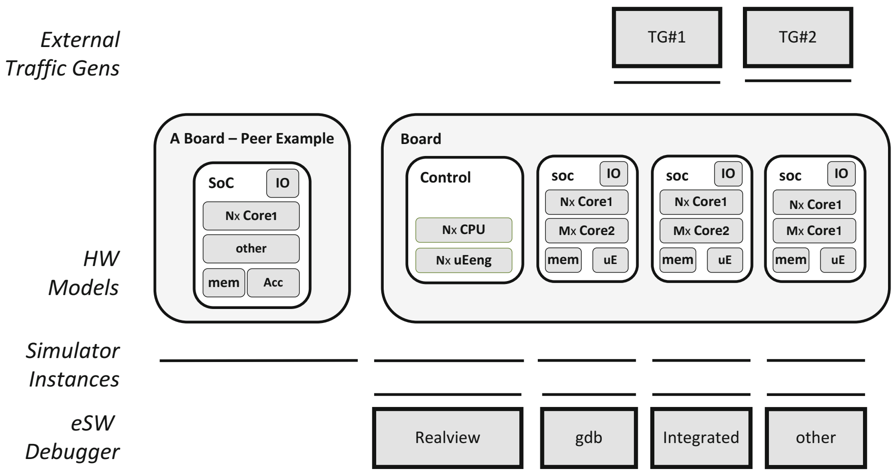


Fig. 8.1 Generalized SW development use case for a distributed simulation with heterogeneous debug and simulation tools

Contemporary SoCs are complex many-core platforms [3]. Especially network infrastructure, such as radio base stations or routers, may easily comprise multiple SoCs each with 10s–100s of processor cores along with memories, interconnect hierarchies, and various accelerator and IO modules. Models in this domain can instantiate 10s of thousands of SC objects. Their joint simulation with instruction-precise processor models makes the speed of the simulation a key issue. With fixed requirements on abstraction level (e.g., programmer's view) and modeling techniques (TLM—transaction level modeling), other ways are needed to improve simulation speed and to tackle the complexity of the models. Distributing the SystemC/TLM simulation into multiple parts that run in parallel, potentially on different simulation hosts, is a promising approach.

Yet, for the model to be widely usable, a suitable solution should support the generalized use case as in Fig. 8.1. A hierarchical simulation model is set up to run a multi-SoC simulation in a distributed fashion. Its parts run on different SC simulators and comprise heterogeneous cores which are to be debugged simultaneously with different tools, be it a core's native tool chain, standalone 3rd-party debuggers, or integrated multi-core debuggers. None of these tools nor the used communication infrastructure must monopolize the execution and block other tools and simulators from functioning. These requirements exclude prior solutions relying on IO virtualization of dedicated SoC interfaces [1] or on changes to the SC simulation engine [12, 15, 17].

We propose *CoMix*, the Concurrent Model Interface, as orchestrating infrastructure for the distributed simulation of large-scale SystemC models. CoMix provides robust, asynchronous communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. It is a modular, vendor-independent overlay technology supporting the full range of SC and TLM communication primitives.

Before going into the key principles of our solution in Sect. 8.3, we overview SystemC and TLM briefly in Sect. 8.2. Section 8.4 details the implementation of the CoMix framework and discusses its interaction with the SystemC/TLM simulation libraries. Section 8.5 evaluates and characterizes CoMix using a set of synthetic benchmarks as well as a real-world virtual prototype. We compare our approach to related work (Sect. 8.6) and conclude on its features in Sect. 8.7.

8.2 SystemC and TLM

SystemC [9] is a system modeling language and C++ class library which adds distinct notions of hierarchy, concurrency, and simulation time to C++. A system is composed hierarchically from modules that communicate explicitly via ports/exports and channels. Its actual function is described within the modules as a collection of concurrent SC processes which explicitly synchronize on events (notify/wait). These processes are scheduled cooperatively. Execution is thread-safe as the SC scheduler runs them sequentially within a single OS thread.

TLM [9] is a modeling library on top of SC, which provides abstractions for the communication protocol and interfaces between SC modules. Modules may use sockets (sets of ports and exports) to exchange transactions between initiators and targets. Such exchange may be split into several phases or timing points, increasing the temporal resolution of the transfer. Blocking transfers have two timing points, while non-blocking transfers have at least four timing points. The former are modeled as a single function call that blocks the initiator's execution until the result is available, while the latter enable continued execution, potentially initiating further transactions before the first one completes (via callbacks, sequencing through the phase diagram, cf. Fig. 8.4).

TLM also introduces the concept of temporal decoupling, allowing processes to run ahead of the simulation time up to an upper limit, the quantum. At points of communication, a SC process may choose to first synchronize its local time with the global simulation time, i.e., to yield to other processes, or may continue its execution unsynchronized, maintaining a delta time. Synchronization guarantees correctness of an access, e.g., to a shared state. Unsynchronized continuation just accesses the current state accepting the temporal error associated with accessing that state too early or too late. Temporal decoupling is commonly used in the context of virtual platform simulations where the software stack does not depend on the low-level timing details of the hardware, which means the temporal error does not manifest functionally. Trading off simulation speed and accuracy, the error can be controlled by the value of the quantum, which depends on the application: A too large value may harm the system's function (e.g., trigger a software timeout), while a too small value yields frequently and slows down the simulation.

8.3 CoMix Fundamentals

CoMix is a modular collaboration infrastructure which allows heterogeneous SC simulators to concurrently execute a distributed SC/TLM model.

Partitioned SC/TLM Model As a prerequisite, a simulation model, i.e., a hierarchy of communicating SC modules, must be cut and grouped into parts for the individual simulators, as shown in Fig. 8.2. In the process, all connection cuts are assigned a unique identifier (cut id). Such a partitioning does not necessarily have to be along SC hierarchies [12, 13]. Yet, following natural boundaries of SoCs or IP subsystems will avoid hierarchy inconsistencies and maintain accessibility for tools. This may require a structural transformation of the original model. The result is a collection of SC modules for each part with open ports or sockets representing a cut. In a sequential simulation, these parts can be instantiated, connected, and simulated together, e.g., for verification purposes (Fig. 8.2, top).

In the distributed case, the individual parts are loaded into different simulators (Fig. 8.2, bottom). On each of the simulators a CoMix *peer* module is inferred and all open ports are bound to CoMix *connectors* either directly (TLM sockets) or via a channel (SC ports). Alternatively this may happen explicitly, coded as part of the SC netlist.

Network of Peers Before the distributed simulation starts, those peers that share at least one cut SC/TLM connection establish a direct TCP/IP link for the exchange of messages. Peers without communication requirements are not connected and do not synchronize.

Synchronization CoMix follows a loosely timed synchronization scheme that is similar to the concept of temporal decoupling in TLM. Each simulator advances its local time up to a configurable quantum, called the sync credit. Once the quantum has been reached (i.e., available credit has been consumed), synchronization with connected peers takes place, which means credit is granted to connected peers. The simulation halts if and only if there is insufficient credit available. Such a scheme preserves and exploits TLM's temporal decoupling semantics as each simulator may advance SC time locally at its own speed during sync intervals. As with TLM, the temporal error between peers can be controlled by the sync interval. Credits may be received any time, not just at the end of sync intervals. Thus, depending on the duration of the sync interval and the load distribution between peers, the slowest simulator can be expected to never stop its advancement of SC time, which effectively minimizes the synchronization overhead for the overall execution time.

Communication During the decoupled execution, communication between SC modules on different peers may take place. Such communication carries a time stamp that can be used to synchronize with a target's local simulation time. Recognizing the need for different, application-dependent schemes, CoMix encapsulates the handling of a connection's synchronization requirements within the associated pair of connectors. They may synchronize to the local time, so that a message is

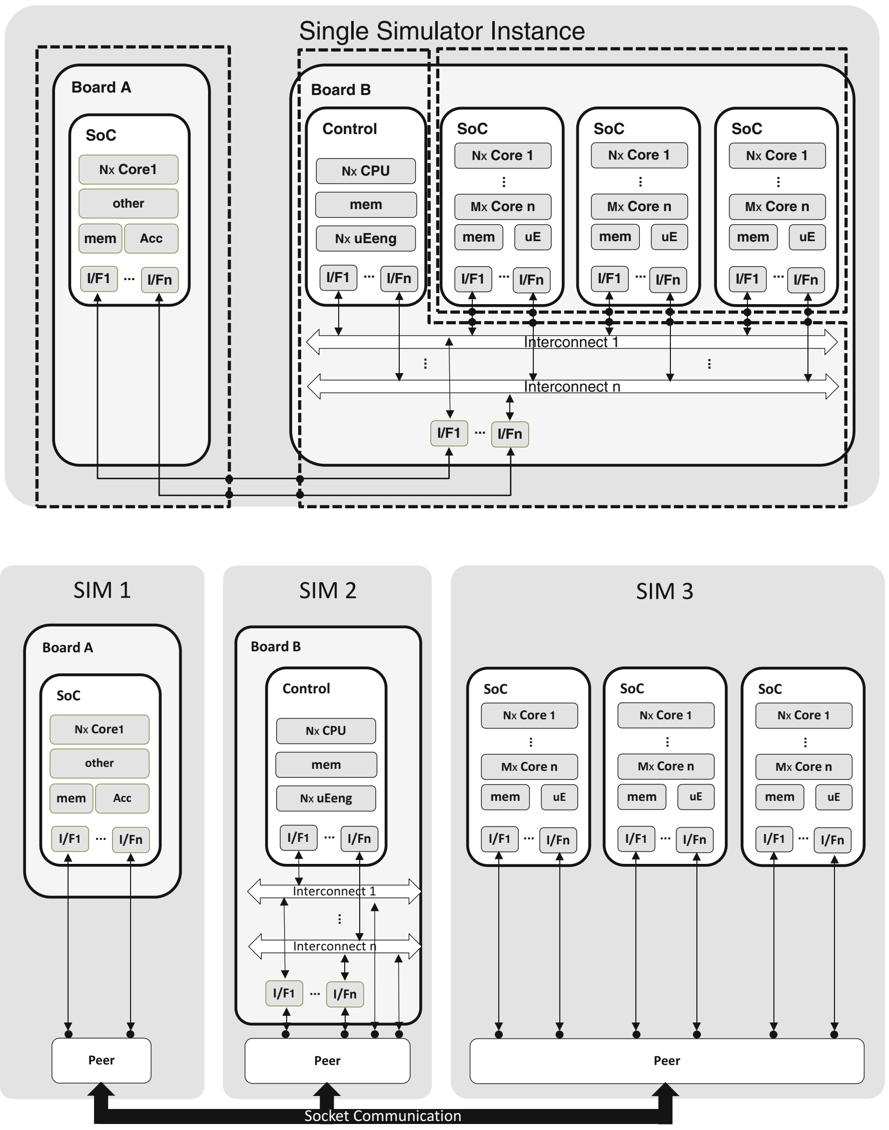


Fig. 8.2 A model (cf. Fig. 8.1) (*top*) is partitioned manually cutting SC connections, and distributed across three peers using CoMix (*bottom*), which handles the communication between cuts

not processed before its creation time, or may handle it immediately at the time of its reception. In both cases, the order of transactions within the same stream is maintained while independent streams may interleave differently towards the same target.

SC Support and Simulator Interaction CoMix supports the full range of SystemC communication primitives, i.e., communication via ports and channels as well as communication via TLM sockets. A current limitation is the lack of DMI support between sockets on different hosts. Life cycle management is controlled by the four simulator callbacks into the CoMix Peer. Interaction towards the simulator is required in only two cases: (1) if sync credit is lacking, the simulator is starved, and (2) in the event of SC communication, which is received asynchronously by CoMix and results in an *async_request_update()* call [9].

8.4 CoMix Framework

CoMix connects the distributed parts of a simulation, provides robust communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. The framework is implemented in C++ and only relies on SC/TLM and Boost's ASIO library.

Figure 8.3 shows the main building blocks of the framework and their interaction. A single instance of the CoMix peer manages its enclosing simulation in the distributed setup. It comprises a multi-socket object, an asynchronous receive queue, and functions for message routing, life cycle management, and the synchronization with other peers. This peer is associated with a collection of CoMix connectors that are bound to the previously open SC ports/sockets of the model. These connectors translate SC communication into message sequences and vice versa. The peer itself

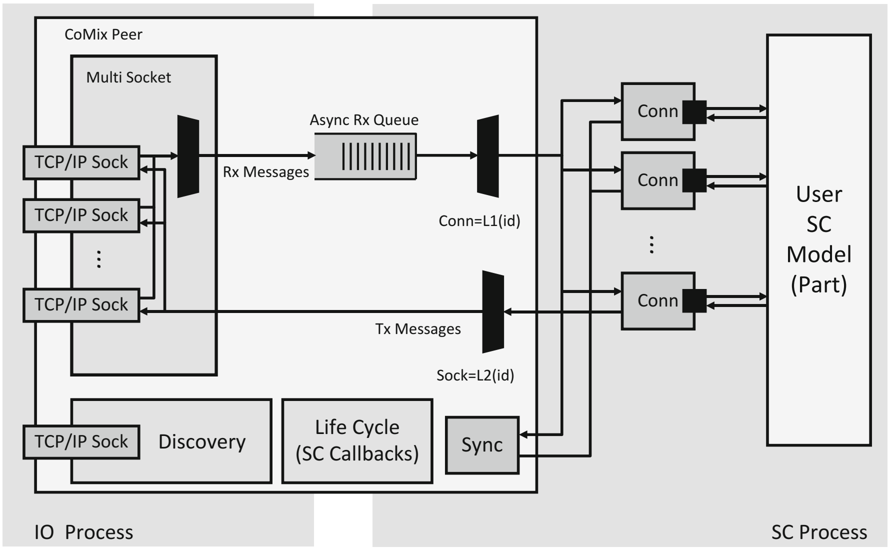


Fig. 8.3 Components of the CoMix framework

handles messages related to synchronization and life cycle management directly. Within the multisocket class an extra, shared OS thread is introduced which handles most IO operations asynchronously to SystemC executed in the main thread. Only outbound messages are sent synchronously.

8.4.1 CoMix Peer

Most of the CoMix function is encapsulated within the CoMix peer. As an SC object this class can be integrated into a model like any other module. All peers within one distributed simulation are identical.

During startup, peers discover each other and form a mesh as required for the connectivity of the simulation model. For this, the peers run a discovery protocol in which the peer started first becomes a super peer running on a specified listening address (IP, port). Others can connect to it, authenticate, and announce their local cut ids together with their own listening address. The super peer broadcasts this info to all its other connections. Upon reception of such a message a peer opens a direct and authenticated connection to the originating address, but only if they share a cut id. Once all local cut ids are associated with their remote counterpart, a peer's setting is considered sane and the connection to the super peer may be closed. Tables with remote and local cut ids are kept for routing messages locally to the socket connection (send) or the local CoMix connector (receive), respectively.

Since the reception of a message is asynchronous in a separate OS thread, it must be explicitly synchronized with the SystemC simulation. This is handled by the receive queue which guards accesses with locks and asynchronously notifies the kernel. In the event of written data a SC process is activated, which performs the lookup and forwards the message to the appropriate connector.

The peer also handles the synchronization of SC time between simulators. Listing 8.1 shows the pseudo code for one of the synchronization modes, the fully starved mode. After using up its own credit (6), a peer sends credit to all of its connected peers (7) and then starves the SC simulation completely within the inner loop (9) until it received sufficient credits from its peers. While SC is blocked, asynchronous reception and processing of messages must continue, hence the asynchronous receive queue is read periodically (12).

Listing 8.1 Fully-starved synchronization scheme
```
 1
 2  void sync_th() {
 3
 4    while ( !canSync( SIM_STOP) ) {
 5
 6      wait( sc_time( credit_ns , SC_NS) );
 7      send_credit(ALL_PEERS, SYNC_CREDIT, credit_ns);
 8
 9      while (!canSync(QUOTA_SYNC)) {
10        if (canSync(SIM_STOP) ) break;
11        usleep(interval_us);
12        nb_recvMessage();
13         }
14
15         clearSync(QUOTA_SYNC);
16     }
17
18     send_credit(ALL_PEERS, STOP_CREDIT);
19     sc_stop();
20 }
```

Another synchronization mode is the delta-only mode, partly shown in Listing 8.2. In this case, the SC simulator is not starved but continues advancing time in delta cycles (12) while waiting for sync credit. This way transactions arriving late may still be processed within the past quota, which can reduce the temporal error at the initiator side. Explicit reads from the asynchronous receive queue are not required.

Listing 8.2 Delta-only synchronization scheme

```
 3
 4         ...
 5
 6         wait( sc_time( credit_ns , SC_NS) );
 7         send_credit(ALL_PEERS, SYNC_CREDIT, credit_ns);
 8
 9         while (!canSync(QUOTA_SYNC)) {
10           if (canSync(SIM_STOP) ) break;
11           usleep(interval_us);
12           wait(SC_ZERO_TIME);
13         }
14
15         ...
```

The outer loop (4) handles the synchronization at the end of the simulation. Before a peer stops (19), it sends out stop credit to its peers (18), enabling them to stop as well. Further life cycle management is achieved by means of SystemC's simulation callbacks, which are forwarded to notify peers.

8.4.2 Connectors

CoMix connectors link open TLM and SC ports of a partitioned design with the CoMix messaging infrastructure. They are bound to their respective SystemC port or TLM socket and associated with the CoMix peer. An extensible set of CoMix connectors exists that can be categorized by:

1. Synchronization of inbound messages. Some connectors contain a payload event queue for inbound messages, which naturally handles their synchronization to the local SC time on a per connection and message type basis. For instance, *scsignal* value updates or *btransport* calls may be synchronized; *transportdbg* calls do not consume time and are not synchronized.
2. Handling of delta time. Outbound TLM connectors may synchronize a delta-time before sending a message or just annotate it. Similarly, inbound transactors may annotate future time as delta time on transactions instead of synchronizing it locally.

3. SC interfaces. Connectors for the different SC port types and TLM socket types are specialized from common bases. In some cases, standard-compliant protocol transformations are required, e.g., for transitioning through approximately timed communication, see Fig. 8.4.
4. Optimized return paths. Connectors are fully SC/TLM protocol compliant, which requires signaling back the result of a (potentially erroneous and delayed)

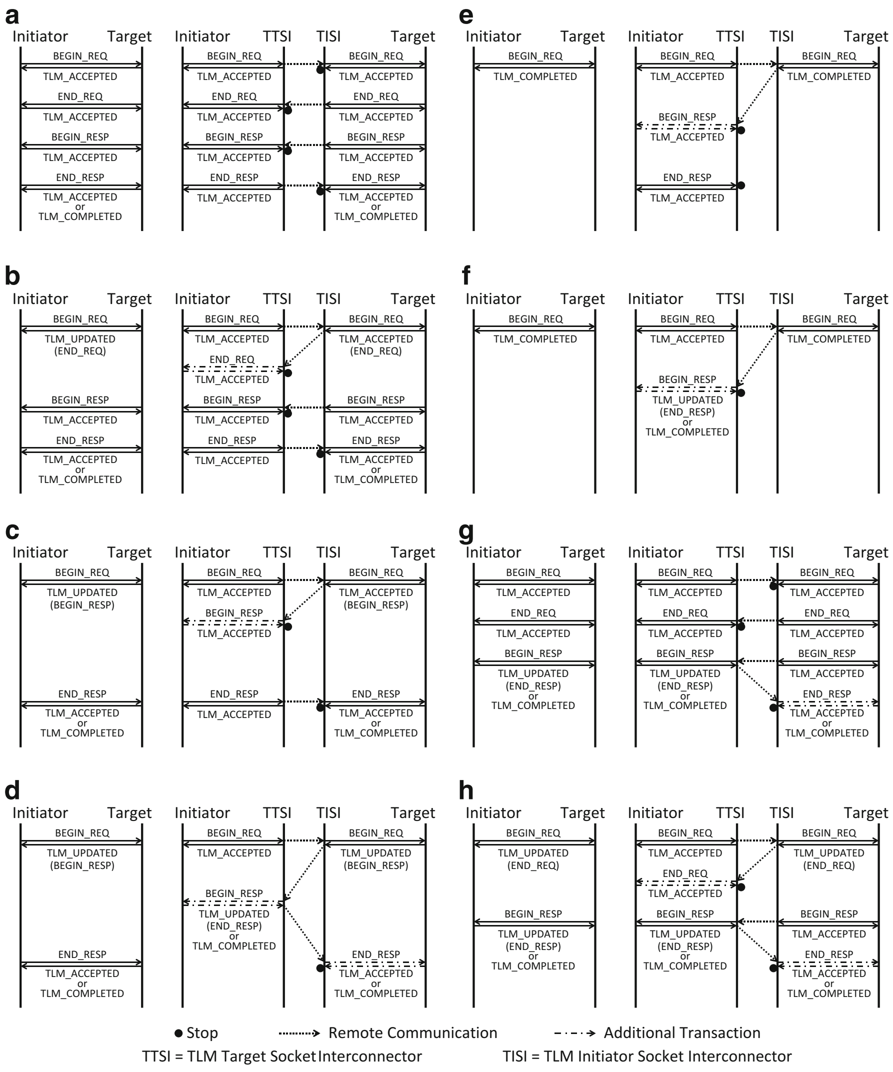


Fig. 8.4 Protocol conversion by nb-transport connectors. Variants a–h show different state transitions between initiator and target for the uncut case (*left*) and for the distributed case with connectors (*right*)

transaction to the initiator. While this cannot be avoided for, e.g., blocking read accesses, it may not be required in all applications. In case of writes, for instance, optimized connectors may skip the status response and instead assert on potential errors. This way, a blocking write call will never block its initiator.

Connectors were designed such that they can be created and configured dynamically by a factory and configuration infrastructure [14]. This enables setting their cut ids through a parameter interface from a suitable design description, which may also comprise the partitioned design.

8.4.3 CoMix Multisocket

The communication between peers is based on TCP/IP sockets which are accessed via the boost asio library. The CoMix multisocket holds a set of connections and a TCP acceptor. It also manages the shared OS thread for the asynchronous IO operations. Messages exchanged over socket connections are translated into boost property trees. Using this standard format ensures that arbitrary message types with widely differing content can be handled robustly in a generic way. At the lowest level of the socket IO library, functions are available for easy serialization of these data structures to character streams and vice versa.

8.4.4 Framework Characteristics

One design goal of our framework was to keep code complexity as low as possible by using standard libraries for both stability and maintainability. Although CoMix provides a powerful feature set, its complexity in terms of lines of code with 2k LoCs remains fairly low, cf. Table 8.1. The SC library, for instance, has 40X as much code. The framework is extensible and even supports, e.g., interfaces to non-SystemC tools, such as (remote) debuggers or traffic generators, by means of specialized peers and connectors.

Table 8.1 Code size of the CoMix framework compared to the SC/TLM library (as reported by cloc)

CoMix	Lines of code
Base	572
Peer	275
Connectors, btransport(), and signal	492
Connectors, nbtransport(), other	648
CoMix total	1.987
SystemC & TLM (2.3.0)	78.359

8.5 Case Study and Results

We apply the CoMix framework to the domain of packet processing and deploy it to a complex real-world many-core platform used for software development. In addition, we report achievable speedups and quantify the temporal error using a set of synthetic benchmarks.

8.5.1 Setup and Measurements

CoMix was tested using an integrated regressable test bench which starts and controls parallel execution of peers in individual shells. Peers run on different processors. Simulations were carried out on virtual machines using 2–4 host CPUs running CentOS as well as on a dedicated Intel-Xeon servers with four cores running RHEL.

We measure the application's *runtime* as the wall clock difference between end- and start-of-simulation callbacks. In distributed settings, we report the overall execution time as runtime of the slowest peer. *Speedups* are calculated dividing non-distributed by distributed runtime.

The *computation-to-runtime* ratio is calculated as the wall clock time a system spends between issuing transactions, divided by the overall runtime. For the synthetic benchmarks, this, e.g., is the time a producer spends in the loop body, outside of the (blocking) send-transaction call. As a more directly measurable variant, we also look at the number of transactions per second (wall clock) as an indication of the communication/computation ratio. The more computation a model performs per transaction, the fewer transactions are handled per second. In settings with constant total numbers of transactions, the throughput is also indicative of the overall runtime.

As an indication for the accumulated temporal error, we measure the overall SC time required for the execution of a particular software task (e.g., communicating a fixed number of tokens). The overall temporal error is calculated as the relative difference in SC time between distributed and non-distributed runs. A distributed simulation may require extra SC simulation time because initiators are waiting (i.e., are blocked) for responses from targets while their SC time advances.

8.5.2 Achievable Speedup

In order to quantify the overhead of our solution, we first look at a synthetic benchmark which combines a producer (P) and consumer (C) in one simulation part. The producer has a token-generating process that can be adjusted in its computational load (i.e., SC and host time consumption) per token, and issues a

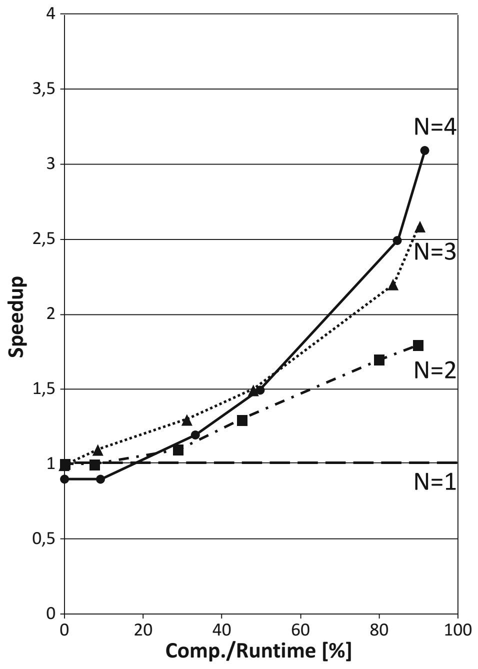


Fig. 8.5 Speedups for the synthetic benchmark distributed to $N = 2..4$ peers compared to the uncut simulation ($N = 1$)

token at the end of each iteration. The consumer receive tokens and verifies their sequence and inter arrival time without consuming SC time, it too can be adjusted in its computational load. Connectors are fully TLM compliant, which requires back signaling (cf. Sect. 8.4.2). Four of these parts are chained in a ring (P1-C2.P2-C3.P3-C4.P4-C1) and distributed onto up to four simulators, resulting in a symmetric load scenario. An additional parameter is the sync interval of the distributed parts which is kept constant for the measurement of the speedup (cf. Fig. 8.5). For the four simulator setting, for instance, a reasonable speedup of up to 3.2 is achieved depending on the computation to communication ratio. As expected, the figure confirms that no or only little speed can be gained for communication dominated settings (0–20 %).

8.5.3 Synchronization Interval

We look at a distributed producer-consumer benchmark (P1-C2) to analyze the sensitivity on the sync interval and report transactions/s in Fig. 8.6. Both parts have identical, generation-rate independent background task loads that are scheduled at 1/10th of the production interval. The throughput is impacted by the sync interval. Fine-grain synchronization limits the throughput, caused by the increased numbers of sync messages (also shown in the diagram) saturating the communication

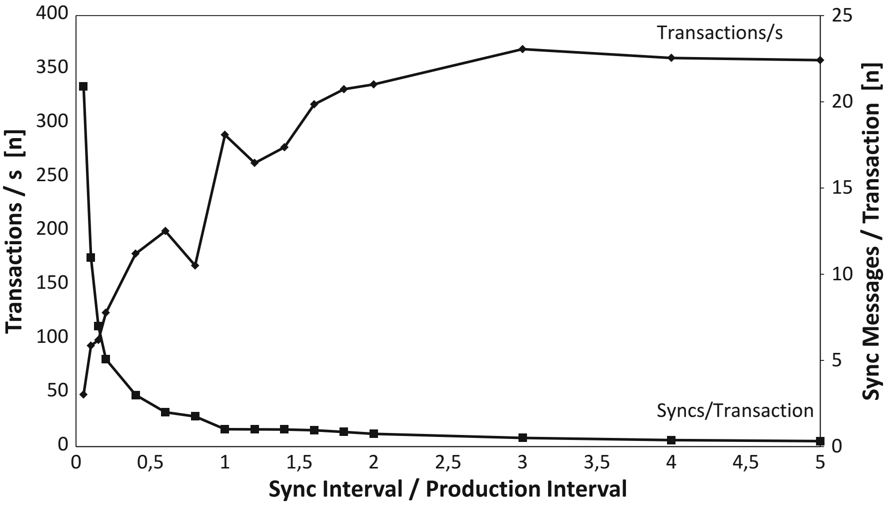


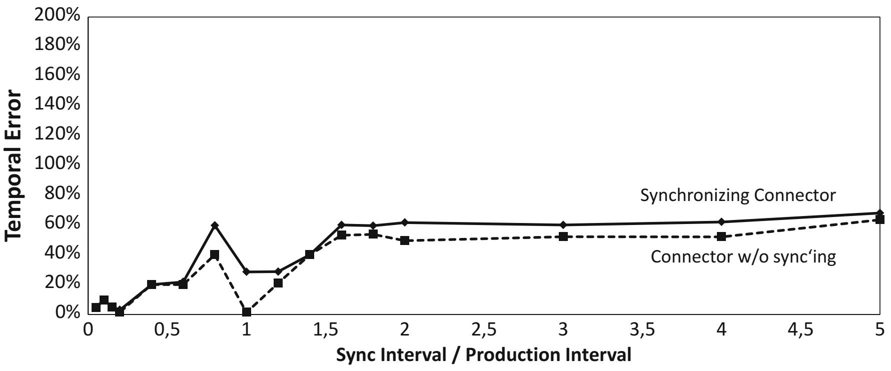
Fig. 8.6 Sensitivity of the simulation speed and throughput (transactions/s), on the sync interval (normalized to the production interval)

channel. For sync intervals set around the generation rate, some instability can be observed. Maximum throughput is reached for sync intervals set about 3X of the production interval. In this case, throughput is limited by the computational load of the background tasks, the latency of the communication channel, and by the temporal error. Larger sync intervals moderately increase temporal error further, leading to a slight degradation (cf. next section). For this measurement, connectors are used which do not synchronize transactions to the local time, as explained next.

8.5.4 *Temporal Error*

In settings with generation-rate independent computational tasks, the temporal error increases the overall runtime as these tasks continue to be executed, e.g., while the initiator is waiting for a response. But the computational background load also slows down the advancement of SC time which effectively lowers the temporal error up to a point where there is none. In contrast, an idle system, i.e., without background load, will always fast forward to the end of a synchronization interval, which means a response is never received before the end of the quota, so that the temporal error solely depends on the sync interval.

These effects can be modulated and (to some extent) compensated for by the schemes used for peer-to-peer and per-transaction synchronization, as Fig. 8.7 shows for the P1-C2 benchmark. With symmetric background task loads for producer and consumer (top), the temporal error has an upper bound that is relatively independent on the per-transaction synchronization scheme of the connector, fine-grain synchronization lowers the temporal error.



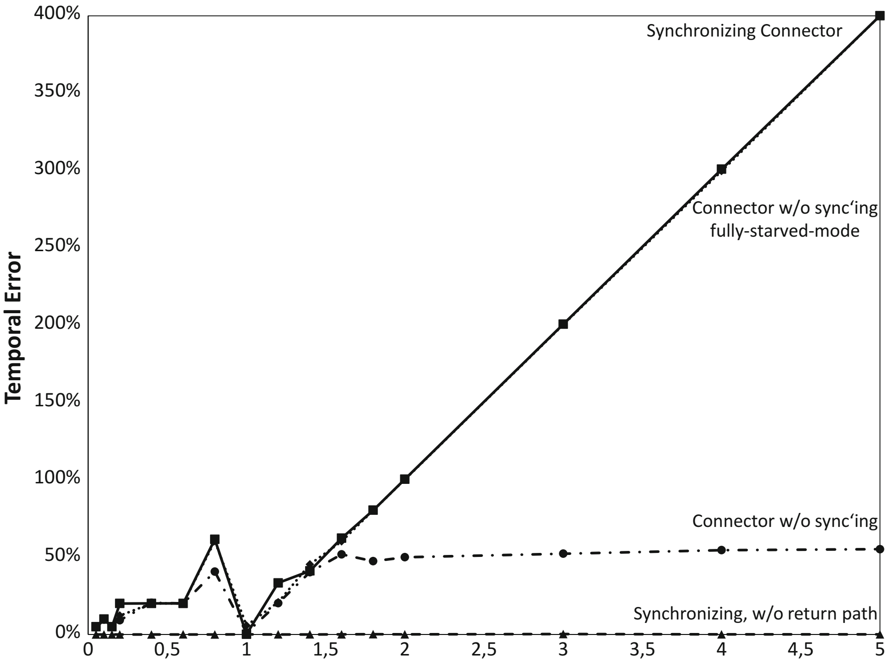


Fig. 8.7 Temporal errors for a symmetric load setting (*top*) and with idle consumer (*bottom*)

In cases of asymmetric loads (bottom, here with idle consumer), the smaller sync intervals cause the same temporal error as before. Above 1.6X, the temporal error increases linearly with the size of the sync interval for the synchronizing connector due to the end-of-quota effect, while the not-synchronizing connector remains at the constant level. However, this only is the case for the delta-only simulator synchronization, which handles late transactions still within the past quota. The fully-starved mode leads to the same error as the syncing connector. For our write-only setting, the temporal error is always negligible if the return path is avoided.

Table 8.2 Elaboration report for a real packet processing platform with four slices and some other functions resulting in about 5k instantiated SC modules

SystemC primitive	Slice	Platform
sc_modules	1037	4745
sc_ports	1459	6680
sc_signals	916	3962
sc_semaphores	8193	32772
sc_methods	436	2094
sc_threads	892	4410
sc_events	3645	17514
tlm2_initiator_sockets	1984	9457
tlm2_target_sockets	2280	10784

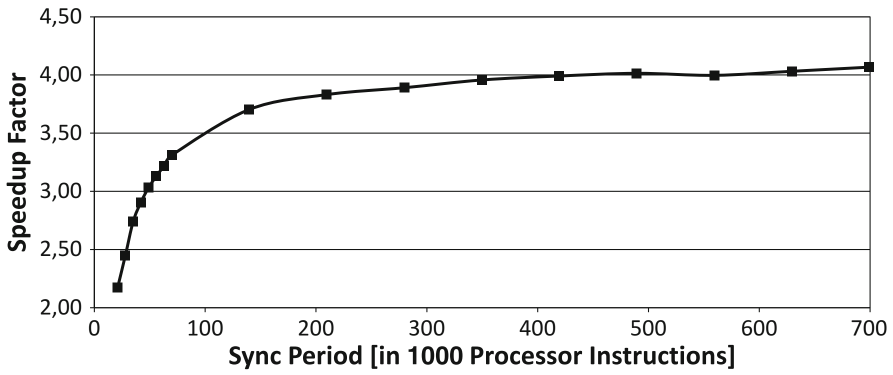


Fig. 8.8 Speedup for the real-world case study over the sync ratio

8.5.5 *Packet Processing Platform*

In a second step we apply CoMix to a real-world packet processing platform. The model has a considerable complexity as shown by its elaboration report in Table 8.2. For the purpose of this book chapter, we distribute a set of four slices into a simulation with four parts. Each of the parts comprises several 10s of binary-translating processor models that are busy running embedded software in temporally decoupled execution. The four parts are connected along write-only IO ports.

We vary the synchronization interval and run 10 simulations per data point to mitigate any load deviations on the simulation hosts. Figure 8.8 shows the speedup over the synchronization interval for the given setting expressed in processor instructions (CPI=1) and normalized to the clock frequency. Starting with about 380k instructions, a speedup of 4X is reached for the given setting, a computation dominated simulation setup with sparse communication and only loose synchronization between the parts.

8.6 Related Work

PDES Synchronization Policy Parallel discrete event simulation (PDES) is researched for several decades. SystemC is a discrete event simulator with unpredictable communication. According to Fujimoto [7] distributed SystemC simulation techniques can be categorized by their synchronization into conservative and optimistic approaches. Conservative schemes [4, 5, 12, 15] require the simulator to be aware of the minimum duration between two communication events in order to ensure temporal correctness while optimistic schemes [10] speculate on their future state. The former schemes impose high communication and synchronization overhead, especially with unpredictable communication (minimum sync period must be assumed), while the latter depend on checkpointing and rollback mechanisms in cases of incorrect speculation.

For unpredictable communication Peeters et al. [13] propose a hybrid synchronization scheme which (1) depends on write-exclusive access to shared memory for functional consistency, (2) avoids expensive frequent synchronization by accepting a temporal error in otherwise asynchronous communications, and (3) synchronizes explicitly at regular system-wide intervals using a blocking double handshake protocol. Similarly, our CoMix uses explicit synchronization intervals, but peers may grant different sync credits to each other which are received asynchronously and non-blocking. Sync messages must not be acknowledged explicitly. Shared memory and write-exclusive access is not required for functional consistency. Communication events from peers are received asynchronously but their processing is scheduled by the SC scheduler maintaining the single-threaded execution semantics of SystemC. CoMix customizable connectors support the full range of SC and TLM interfaces.

The conservative lookahead technique in [17] requires communication to be known ahead of time by at least on synchronization period, i.e., to be predictable. This avoids causality issues due to communication arriving late (as long as the return path is ignored [17]). In such a confined setting, CoMix does behave similarly accurate and without timing error (cf. Fig. 8.7).

SystemC Kernel Modifications Most prior approaches suggest changes to a the simulation kernel for adding communication, synchronization, or parallelization support, e.g., [4, 6, 10, 12, 15–17]. However, this causes a severe maintenance problem for evolving simulator versions and is not feasible in settings with heterogeneous, potentially commercial tools without source code access. Others avoid kernel modifications by providing add-on libraries which interact with the simulator only through the regular SystemC language interface [8, 13]. This interaction depends on the synchronization scheme and might be tight, e.g., per delta cycle as in [8], or rather loose. Our CoMix is such an overlay technology, interacting with the simulation engine only in cases of inbound communication events or explicit synchronization.

Peer-to-peer Protocols and Host Systems Several communication protocols are used for passing messages between peers, including MPI [5, 13], CORBA, and SOAP [11]. We use regular TCP/IP sockets similar to, e.g., [16] to limit the dependency on other libraries and a leaner protocol stack. Most related approaches use SMP machines as simulation hosts, e.g., [8, 10, 12, 15], potentially depending on SMP properties, such as shared memories and caches [13]. Our CoMix is intended for the use in load sharing facilities and geographically distributed settings, similar to [16]. However, CoMix recognizes the potential for optimizations and is factored for the support of other communication protocols.

SC/TLM Primitives and Modeling Styles Especially the kernel modifying approaches may impose special coding styles. Mello et al. [12], for instance, depend on approximately timed modeling semantics in their models. Others require thread safety, at least on distribution boundaries [15]. Both, kernel modifying approaches and overlay solutions often do not support the full spectrum of SystemC and TLM communication primitives [13, 17] or require explicit clocks [11]. Trams et al. [16] is limited to signal communication semantics. In [17], the partly supported TLM communication must not consume SC time. CoMix does not impose modeling restrictions and supports the full range of SC *and* TLM communication primitives by means of connectors for dedicated port/socket types.

8.7 Conclusion

We have presented CoMix, the Concurrent Model Interface, which enables the distributed simulation of large-scale SystemC-based virtual prototypes. CoMix provides robust communication between peers, enables their loose synchronization, and comprehensively manages the overall life cycle. Its modular design supports various synchronization strategies for peers and their communication, which may be chosen depending on a platform's specific requirements. CoMix' asynchronous IO infrastructure integrates into SystemC efficiently and avoid blocking third-party tools, such as embedded software debuggers.

For a set of synthetic, token-passing benchmarks we have shown the benefits of CoMix to be a trade-off between local computation and communication and the synchronization interval. The temporal error caused by the distribution can be lowered if double-synchronized round-trips are avoided by skipping the return path or not synchronizing it. These results were confirmed on a complex real-world platform, where we found speedups of up to 4X in a four part simulation for the given application. To date, CoMix is used in virtual prototypes of many-core network processing systems comprising several hundred instruction-precise processor models.

Acknowledgements In parts, this work has been supported by Lei Lang, Eric Frejd (Ericsson AB, Sweden), and Linmu Cui (Cadence, Germany).

References

1. Bailey, B., Martin, G.: Virtual prototypes and mixed abstraction modeling. In: ESL Models and their Application, pp. 173–224. Springer, Berlin (2010)
2. Bailey, B., McNamara, M., Balarin, F., Stellfox, M., Mosenson, G., Watanabe, Y.: TLM-Driven Design and Verification Methodology. Lulu Enterprises, Raleigh, NC (2010)
3. Benini, L., Flamand, E., Fuin, D., Melpignano, D.: "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," In: Design, Automation & Test in Europe Conference & Exhibition (DATE 2012), pp. 983–987, 12–16 March 2012. doi:10.1109/DATE.2012.6176639. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6176639&isnumber=6176405 (2012)
in Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 983–987, 12–16 (2012) doi: 10.1109/DATE.2012.6176639 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6176639&isnumber=6176405
4. Combes, P., Caron, E., Desprez, F., Chopard, B., Zory, J.: Relaxing Synchronization in a parallel systemC kernel. International Symposium on Parallel and Distributed Processing with Applications (ISPA) (2008)
5. Cox, D.R.: RITSim: distributed systemC simulation. Master's thesis, Rochester Institute of Technology (2005)
6. Ezudheen, P., Chandran, P., Chandra, J., Simon, B., Ravi, D.: Parallelizing systemC kernel for fast hardware simulation on SMP machines. In: 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS) (2009)
7. Fujimoto, R.M.: Parallel and distributed simulation. In: Proceedings of the Winter Simulation Conference (1999)
8. Huang, K., Bacivarov, I., Hugelshofer, F., Thiele, L.: Scalably distributed systemC simulation for embedded applications. In: International Symposium on Industrial Embedded Systems (SIES'08) (2008)
9. IEEE SystemC Language Reference Manual. IEEE Std 1666–2011 pp. 1–638 (2012)
10. Jones, S.: Optimistic parallelisation of systemC. Technical Report, University Joseph Fourier, MoSiG DEMIPS (2011)
11. Meftali, S., Dziri, A., Charest, L., Marquet, P., Dekeyser, J.L.: SOAP based distributed simulation environment for system-on-chip (SoC) design. In: Forum on Specification and Design Languages (FDL) (2005)
12. Mello, A., Maia, I., Greiner, A.; Pecheux, F.: "Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations," In: Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), pp. 606–609, 8–12 March 2010. doi:10.1109/DATE.2010.5457136. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5457136&isnumber=5456897 (2010)
13. Peeters, J., Ventroux, N., Sassolas, T., Lacassagne, L: "A systemc TLM framework for distributed simulation of complex systems with unpredictable communication," In: 2011 Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 1–8, 2–4 November 2011. doi:10.1109/DASIP.2011.6136847. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6136847&isnumber=6136840 (2011)
in Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on, pp. 1–8, 2–4 (2011) doi: 10.1109/DASIP.2011.6136847 http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6136847&isnumber=6136840
14. Sauer, C., Loeb, H.P.: A lightweight infrastructure for the dynamic creation and configuration of virtual platforms. In: 3rd Workshop on Virtual Prototyping of Parallel and Embedded Systems (VIPES) along with Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV) (2015)
15. Schumacher, C., Leupers, R., Petras, D., Hoffmann, A: "parSC: synchronous parallel SystemC simulation on multi-core host architectures," In: 2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 241–246,

24–29 October 2010. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5751508&isnumber=5751486 (2010)

16. Trams, M.: Conservative distributed discrete event simulation with systemC using explicit lookahead. Technical Report, www.digital-force.net (2004)
17. Weinstock, J.H., Schumacher, C., Leupers, R., Ascheid, G., Tosoratto, L: "Time-decoupled parallel SystemC simulation," In: Design, Automation and Test in Europe Conference and Exhibition (DATE 2014), pp. 1–4, 24–28 March 2014. doi:10.7873/DATE.2014.204. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6800405&isnumber=6800201 (2014)