SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments

Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Dietmar Petras* and Andreas Hoffmann*
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University, Aachen, Germany
*Synopsys GmbH, Aachen, Germany
Email: jan.weinstock@ice.rwth-aachen.de

***Abstract*—Virtual platforms have become essential tools in the design process of modern embedded systems. Their accessibility and early availability make them ideal tools for design space exploration and debugging of target specific software. However, due to increasing platform complexity and the need to simulate more and more processors simultaneously, performance of virtual platforms degrades rapidly.**

This work presents SystemC-Link, a segment based parallel simulation framework for SystemC simulators. It achieves high simulation performance by using a parallel and time-decoupled simulation approach. Furthermore, it offers a virtual sequential environment for each simulation segment. This enables use of legacy models by allowing operation on global state without risking race conditions during parallel simulation.

The approach is evaluated in a variety of scenarios, including a contemporary multi-core platform based on the OpenRISC architecture running Linux. For this benchmark, a 3.2× higher simulation performance was achieved with SystemC-Link compared to standard SystemC on a regular workstation PC.

I. Introduction

In an era with increasingly complex embedded systems, developers rely on virtual platforms for software debugging and for keeping time to market short. Yet, their viability as a productivity tool is threatened due to reduced simulation performance, which is a result of the increasing number of processors in modern designs. To boost simulation speed again, various approaches are being explored. Lowering the simulation detail yields faster simulation speeds by sacrificing accuracy. Transaction Level Modeling (TLM) [1] is a widely used example of this. Communication between simulation components is abstracted using Interface Method Calls (IMC) instead of modeling the individual signals. Further acceleration can be achieved by using fast instruction set simulators (ISS) [2] or by improving the performance of the underlying simulation engine. Due to the high availability of multi-core PCs today, the adoption of parallel simulation technologies appears an attractive goal for modern Electronic System Level (ESL) designs.

Problems hindering adoption are in part the requirement to use simulation models (e.g., processors, buses and peripherals) that are coded in a thread-safe way. State accessed from different simulation processes must be protected from concurrent access which would otherwise lead to race conditions [10]. Since the de-facto industry standard simulation environment, SystemC [1], operates strictly sequentially, most models do not account for this.

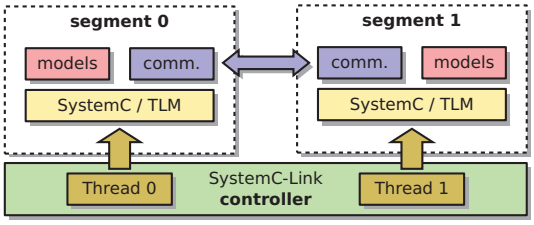

Fig. 1. SystemC-Link overview

Therefore, a viable parallel simulation technology must not only provide a high speed up, but also emulate a sequential environment that protects models from race conditions. This work presents *SystemC-Link*, a simulation environment that executes SystemC based simulation *segments* in parallel as illustrated by Fig. 1. High simulation speed is achieved by deploying time-decoupling between the segments. This relaxes the synchronization requirement between segments and allows the simulation controller to let the simulation run in parallel for a certain time before synchronizing. In summary, the contributions of this work are:

- A parallel *simulation controller* for SystemC based simulation segments
- Inter-segment *communication infrastructure* based on regular TLM and supporting time-decoupling
- A *virtual sequential environment* to protect global state of each simulation segment from races

The rest of this work is structured as follows. An overview of related work regarding parallel SystemC simulation is given in Sec. II. Subsequently, Sec. III and Sec. IV describe the simulation controller and the communication infrastructure used for inter-segment communication. To elaborate on the efficacy of this approach, Sec. V presents experimental results, including a contemporary multi-processor system based on the *OpenRISC* architecture, capable of running a modern Linux kernel. Finally, conclusions are given in Sec. VI.

978-3-9815370-6-2/DATE16/

II. Related Work

Previous work in the domain of parallel SystemC simulation focusses on conservative, synchronous simulation, e.g., *parSC* [12] and *RAVES* [14]. Synchronous simulators distribute the execution of simulation processes within the same delta cycle among multiple threads. Further approaches exist that can additionally offload processes to GPUs, e.g., *SAGA* [15] and *SCGPSim* [9]. However, performance of synchronous approaches in general is limited by the amount of activity within a single delta cycle. This is further supported by the results of SystemC on the Intel SCC [11] and a light-weight C++ simulation kernel [6].

This issue has been identified as a general problem for parallel ESL simulators [5], motivating techniques that parallelize beyond the delta cycle boundary. E.g., a *duration* primitive is proposed in [8] which allows manual specification of processes that may run in parallel over multiple cycles. *SimParallel* [4] allows parallel simulation at a global clock cycle level, possibly spanning multiple delta cycles. Time-decoupling is used in the parallel SystemC kernel *SCope* [16]. Each thread executes its own simulation loops and advances time at its own pace. This time-decoupling is limited to a maximum time difference between the fastest and slowest thread to allow temporal correct cross thread communication.

Most parallel SystemC approaches rely on the simulation models to not use shared state and communicate with each other only using standard signals or sockets in order to avoid race conditions. Individual models that share data must be grouped on the same thread [12], [16], thereby effectively disabling parallel simulation of those models. For the domain of SpecC, static source code analysis can be applied to identify simulation processes which may safely run in parallel [3]. Unfortunately, SystemC models are often only provided in a precompiled form without source code. Moreover, SystemC makes heavy use of C++ inheritance and IMCs, further complicating static analysis.

In contrast to other approaches, *SystemC-Link* is able to handle parallel simulation of models that access global state in an unsynchronized fashion (for example via C/C++ pointers) by placing them in a *virtual sequential environment*. *Time-decoupling* is deployed to further increase simulation performance in typical ESL scenarios.

III. Simulation Architecture

In *SystemC-Link*, a simulation consists of *segments* interconnected with *channels*. It is executed within one host process using a single address space. Each segment contains its own SystemC kernel and models. Channels are used for one-way communication between two segments. A simulation *controller* is used to advance simulation by *stepping* each segment. For example, a segment could consist of an ISS embedded into a SystemC environment. It could be connected to a memory component in another segment using a channel like shown in Fig. 2.

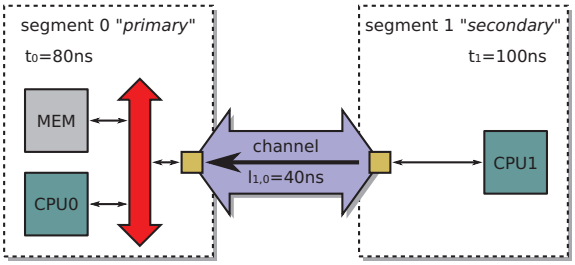

Fig. 2. Time-decoupled segments via latency channels

Manual partitioning allows the programmer to explicitly state which segments are allowed to share data using channels. Moreover, by representing the simulation as a list of segments, parallel simulation becomes trivial, since it is not necessary to alter the internal discrete event simulation facilities of the SystemC kernel. Instead, each segment may be stepped independently and in parallel.

A. Time-Decoupled Co-Simulation

As already stated in Sec. II, synchronous simulators do not offer enough parallelization potential to significantly accelerate ESL simulators. *SystemC-Link* therefore makes use of time-decoupling [16] to allow segments to run for a certain duration before synchronizing. However, this complicates inter-segment communication, since each segment now operates in its own time zone with a local time t_i. In order to bridge time zones between interconnected segments, channels are assigned a latency, denoted as $t_{l,ij}$ for a channel between segments i and j. Channel latency describes the amount of time allowed to pass after putting a token into a channel before it must be fetched by the receiver. Segments and latency channels form a *channel latency network*. It is used by the simulation controller to determine the amount of time a segment may simulate ahead of time before risking to miss a channel token from its peers. It is the task of the simulation controller to make sure that a segment has never advanced too far ahead in time. To that extent, the controller computes a limit time $t_{\mathrm{lim},i}$ for each segment i:

$$t_{\mathrm{lim},i} = \min_{j \in peers(i)} \left(t_j + t_{l,ji} \right) \tag{1}$$

Segments that have not reached their limit time are considered *ready to simulate*, while others are considered *waiting* for their peers to catch up.

Fig. 2 presents a *SystemC-Link* simulation consisting of two segments: a primary segment containing processor, bus and memory components and a secondary segment only holding a processor. The connection between the secondary processor and memory on the primary segment is made using a channel with a latency $t_{l,10} = 40\,\mathrm{ns}$. Assuming that the local time stamp of segment 1, t_1, is currently 100 ns, segment 0 does not need to check for new channel tokens earlier than 140 ns ($t_1 + t_{l,10}$), allowing it to simulate uninterruptedly for a duration of 60 ns.

B. Scheduling

SystemC-Link uses a cooperative scheduling approach for simulating segments using coroutines similar to those used by SystemC for thread processes. Once started, the scheduler picks the first segment from the *ready to simulate* queue and switches to its execution context. If multiple threads are used for simulation, each thread picks a segment until the queue is empty. Within its execution context, a segment is first initialized before simulation is started, using interface functions *init* and *step*. A default implementation for *step* is provided in *SystemC-Link*, *init* must be added by the user to construct the module hierarchy. Once the simulation finishes (either by reaching the end time stamp or by calling *stop* in any segment), the function *exit* will be invoked to allow for clean-up.

The scheduler can be configured with different operation modes. These modes control the conditions under which the coroutine that executes the simulation loop of a segment yields control back to the scheduler. Two different modes are available called as-soon-as-possible (ASAP) and as-late-as-possible (ALAP):

- *ASAP scheduling*: a segment yields control back to the scheduler once it wants to advance its time to a different time stamp. This results in shorter channel communication delays since segment time stamps will be closer together, but will increase the number of context switches, reducing simulation performance.
- *ALAP scheduling*: a segment yields control back to the scheduler once it reaches its limit time $t_{\mathrm{lim},i}$. This reduces the amount of context switches throughout the simulation, but local time stamps of segments will differ by a larger amount.

The choice of scheduling mode presents a trade-off between simulation performance and timing accuracy. While currently, scheduling modes can only be specified globally, future work might enable per-segment configuration to support fine-tuning.

C. Virtual Sequential Environment

Segments form virtual sequential environments, meaning that all simulation processes and all incoming IMCs do not execute concurrently within one segment. This frees the programmer from protecting shared state using locks and enables the use of non-thread-safe, off-the-shelf models in parallel simulation. Furthermore, singleton models that rely on global state (e.g., an ISS using global variables to store register values), may be used multiple times in different segments without costly redesign.

Segments are implemented as shared object files. All software that uses global variables which should be replicated for each segment must be statically linked. For example, the *OSCI* SystemC kernel uses the global variable *simcontext* to store status information, such as local time stamps and discrete event lists. Since this data needs to be replicated for each segment, a copy of the simulation library must be statically linked to the segment.

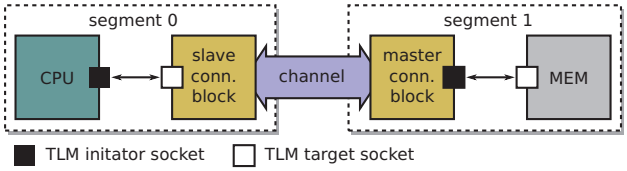

Fig. 3. Cross segment communication via connector blocks

In some situations it might be efficient to load a segment multiple times. Considering the example in Fig. 2, the system can easily be extended to a quad-core system by instantiating the secondary segment multiple times and creating new channel connections. The Linux dynamic loader, however, will not load the same shared object twice and will instead make a reference to the first one. Since this violates the global state replication guarantee for segments, a workaround is deployed: segments are assigned a unique file name before loading to deter the loader from detecting that they reference the same shared object.

IV. Communication Infrastructure

Inter-segment communication is done via channels. To access these channels, special *connector blocks* must be placed into the simulation and connected using regular signals or TLM sockets as illustrated by Fig. 3. Slave connector blocks receive transactions and send them over the channel. Master connector blocks receive those transactions and forward them to their destination using regular TLM communication interfaces. During simulation elaboration, a channel entry point must always be linked to one slave connector block and an exit point to one master block. In order to identify connector blocks simulation wide, their hierarchical SystemC name is preceded with the unique ID of the segment they were instantiated in.

Channels support two methods for transmitting messages: queue-based and IMC-based communication. Both methods are described in the following.

A. Queue-based Communication

The queue-based communication flow is used for transactions transmitted using the blocking transport interface. It respects timing semantics by annotating the time spent to forward the transaction through the channel to the receiver using the local time offset parameter t_d of the TLM blocking transport interface.

The queue-based transmission procedure is outlined in Fig. 4. Internally, the channel uses two queues for forward and return directions. Once a transaction is received by the target socket of the connector block (1), it is put into the forward queue (2). The calling SystemC thread process is then suspended, waiting for a wakeup event (3). On the receiver side, a collect process continuously polls the forward queue for new messages (4). Polls must be carefully timed in order to not waste execution time but also not miss messages. Since the scheduler defines a limit

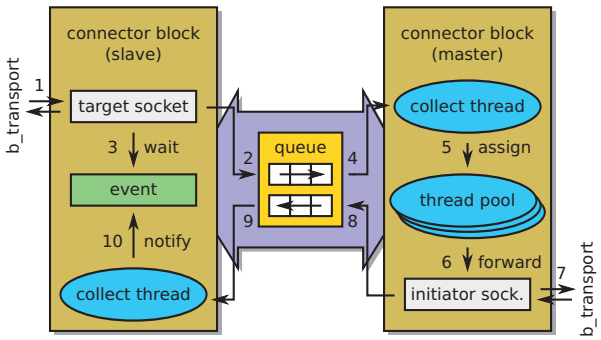

Fig. 4. Queue-based communication flow

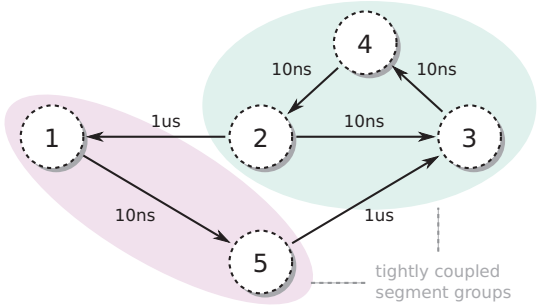

Fig. 5. Channel latency network experiment setup

time ($t_{\mathrm{lim},i}$) to which it is safe to run up to without missing messages, polls must be scheduled to run no later than $t_{\mathrm{lim},i}$. Once a message is taken from the forward queue, it is assigned to a forward process from a thread pool (5). Since the receiver might call wait during forwarding, the collect thread cannot be used for forwarding, as it could block and potentially miss messages. Using the TLM initiator socket of the master connector, the transaction is forwarded to the receiver (6). On the return path (7), the response is put into the backward queue (8). The slave block polls this queue continuously (9), similarly to the collect thread on the forward path. Once the response is taken out of the queue, the corresponding wakeup event is notified (10) to allow the calling process to resume.

Timing annotation is performed when the transaction is removed from the queue (4). Eq. 2 shows how the local time offset t_d is adjusted by the time difference between the sending segment at the point where it put the transaction into the queue t_i, and the receiving segment when it took the transaction out of the queue t_j. The same procedure is repeated on the return path (9).

$$t_d = \begin{cases} t_d + t_i - t_j & \text{if } t_d > t_j - t_i \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Note that time-decoupling guarantees that the time difference between two segments never exceeds the channel latency t_l (see Sec. III-A). Therefore, transactions must be stated sufficiently ahead of time, so that $t_d > t_l$. Otherwise a timing error $t_{\mathrm{err}} = t_j - t_i - t_d$ is incurred.

B. IMC-based Communication

The IMC-based communication flow is used for TLM communication interfaces that cannot normally be interrupted, such as the debug and direct memory interfaces. The queue-based flow cannot be used, since waiting on a wakeup event is forbidden (Fig. 4, step 3). Instead, master connector blocks propagate an IMC interface to the slave block at the beginning of the simulation. Using this interface, slave blocks can forward IMCs directly into the connected segment. Since caller and callee of the IMC reside in different segments, race conditions may appear.

To avoid those, channels also support *exclusive* IMCs. A channel marked for exclusive IMC propagation will ensure that all calls into peer segments are guaranteed to be race free. A segment may only execute its exclusive IMC, while all other segments are paused. To achieve this, once an exclusive IMC is issued over the channel, the calling segment yields execution (see Sec. III-B). It is then put into an exclusive waiting queue by the scheduler. All other segments currently running are allowed to finish their current iteration but are not resumed afterwards. While all segments are waiting, the exclusive IMC is executed by taking the corresponding segment from the exclusive waiting queue and resuming its execution. Once the exclusive IMC returns, all other segments resume execution.

V. Experimentation

To demonstrate the efficacy of the proposed approach, it is evaluated in various experiments. First, section Sec. V-A presents an analysis of the impact of the scheduling mode on performance and timing accuracy. The potential benefits of the channel latency network compared to a static global lookahead approach [16] are investigated in Sec. V-B. Lastly, Sec. V-C shows that the presented approach is able to improve simulation performance of a contemporary ESL simulator. It is based on an *OpenRISC* [13] multi-processor system that is detailed enough to boot a present-day Linux kernel.

All experiments were performed on an Intel i7-5960X workstation PC with 16 GB RAM. Dynamic temperature-based overclocking (Intel TurboBoost) was disabled to ensure consistent benchmarking results for all tests.

A. Scheduling Mode Analysis

The first experiment analyzes the impact of the choice of scheduling mode when sending cross segment transactions insufficiently ahead of time. First, a simulation is constructed consisting of five segments and interconnected with latency channels as shown in Fig. 5, with corresponding channel latencies annotated next to the channels. For each outgoing channel, a segment has a *sender* module, which continuously sends TLM transactions at a frequency of f_{send} and with $t_d = 0$. Segments also contain *receiver*

TABLE I
EXPERIMENT SEGMENT CONFIGURATION

segment	load / NOP loop	f_{work}	f_{send}
1	$10k$ host cycles	100MHz	1MHz
2	$10k$ host cycles	100MHz	10MHz
3	$10k$ host cycles	100MHz	10MHz
4	$10k$ host cycles	100MHz	10MHz
5	$1M$ host cycles	1MHz	1MHz

modules for each incoming channel that acknowledge reception of a transaction and return immediately. Finally, a segment contains a *work* module, mimicking the behavior of an typical ISS or co-processor unit. It executes *NOP loops* at a frequency of f_{work}, with each full NOP loop consuming *load* cycles on the host PC. This load is varied between $10k$ and $1M$ cycles. The exact values used for the experiment can be found in Tab. I.

The experiment consists of running this simulation setup for a total duration of 1 ms using two threads. Afterwards, the average time t_c each transaction spent in the channel is extracted. Fig. 6 shows t_c for each channel as a multiple of its latency t_l for both ASAP and ALAP scheduling modes.

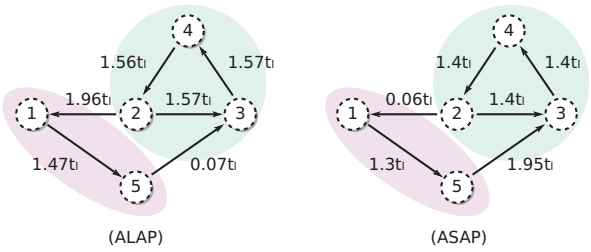

Fig. 6. Average transaction delay t_c

In general, it can be observed that $0 \leq t_c \leq 2t_l$. This is expected since transactions need to pass through the channel twice: once on the forward path and once on the return path. Secondly, ASAP scheduling yields on average 8.4% shorter t_c compared to ALAP mode, resulting in smaller simulation time alterations. Since segments yield execution each time before advancing to the next time stamp, simulation times are kept closer together. Due to the reduced amount of context switches, simulation in ALAP mode ran 18% faster than in ASAP mode.

B. Channel Latency Network Analysis

This experiment analyzes the benefits of a channel latency network compared to a global lookahead [16]. A global lookahead is emulated using the same latency for all channels. To ensure comparable timing behavior, this global latency must be set to the minimum over all channel latencies, i.e., $t_l = 10$ ns. Simulation runtime for a duration of 1 ms is measured while increasing the number of threads used from one to five. The results are presented in Fig. 7.

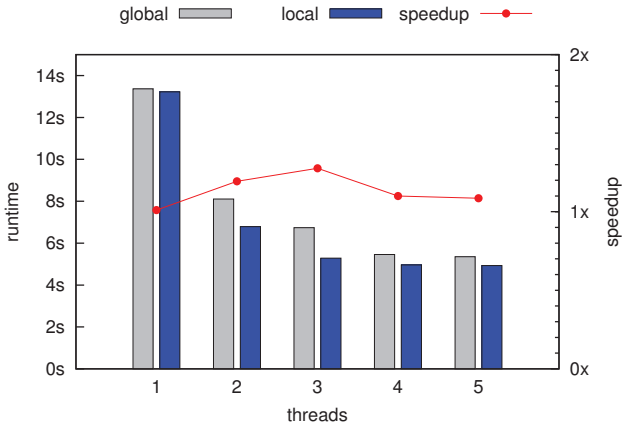

Fig. 7. Performance of global versus local channel latency models

Overall, performance improves when using a channel latency network versus a global lookahead for the presented scenario. The improvement varies between 1% in the single thread case and 28% when using three threads. Using a global lookahead, the simulation waits for the worker of segment 5 every microsecond to finish execution of its relative long NOP loop (see Tab. I). Since all other segments may only simulate ahead by 10 ns as dictated by the channel latency towards their peers, not enough parallel work is available to utilize more than one thread.

Permitting different latencies on each channel causes forming of tightly coupled segment groups, as illustrated by Fig. 5: the first group is formed by segments 1 and 5, the remaining segments 2, 3 and 4 form a second group. Those groups may simulate ahead of each other by 1 µs according to the latencies of channels $2 \rightarrow 1$ and $5 \rightarrow 3$. While one thread is busy simulating the worker of segment 5, other threads may advance simulation of segments from the second group independently, yielding better utilization of the available simulation threads.

Such situations are often encountered in ESL scenarios. Load spikes such as produced by segment 5 can be caused by peripheral components, for example a virtual hard disk that stores its content infrequently to the file system of the host. Another example is a virtual network adapter causing load spikes when sending or receiving Internet packets via the host network.

C. OpenRISC Multicore Platform

The *OpenRISC* multicore platform is constructed from a primary *SystemC-Link* segment, containing one *OpenRISC* ISS, main memory, storage components and virtual I/O. It also holds a configurable amount of master connector blocks to allow connecting further processors in other segments. Secondary segments only hold one *OpenRISC* ISS and a slave connector block each. The platform can be configured to operate in uniprocessor mode (UP) or dual or quad SMP mode (SMP2 and SMP4, respectively). Fig. 8 shows the system in a SMP4 configuration.

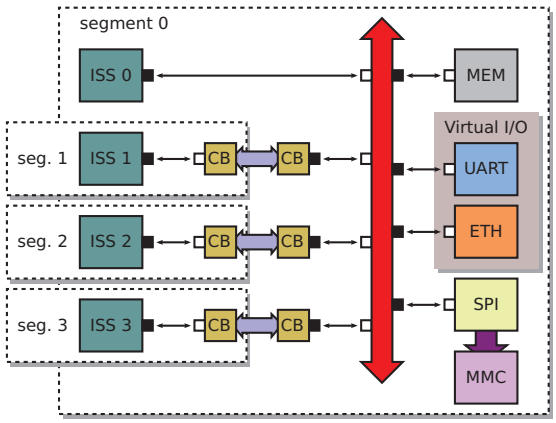

Fig. 8. OpenRISC multi-processor system

For comparison, an *OSCI* SystemC based simulator was also constructed. Since the *OpenRISC* ISS makes heavy use of global data structures, it is only possible to instantiate it once per simulator. Thanks to the virtual sequential environment and global state replication of segments, it is possible to instantiate one ISS per segment, allowing construction of a multiprocessor system in the first place.

The experiment consists of booting a current Linux kernel [7] on different system configurations (UP, SMP2 and SMP4) and comparing simulation performance when using one, two and four threads. Simulation duration is set to 2 s. During that time, all cores are kept active by Linux and produce simulation load. Performance is measured in million instructions per seconds (MIPS) accumulated over all processors. The results are presented in Fig. 9.

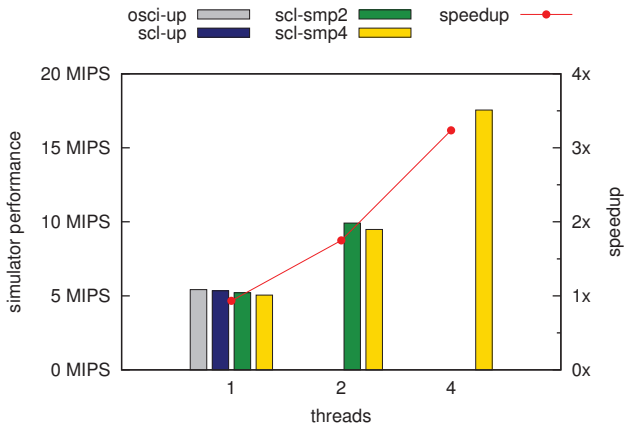

Fig. 9. Simulation speed using OSCI and *SystemC-Link* (SCL)

The uniprocessor *OSCI* version requires 37 s to boot Linux, resulting in a simulation performance of 5.42MIPS. The uniprocessor version constructed of a single segment boots in 37.5 s achieving 5.34 MIPS - a slowdown of 1.5%. The quad core version operates at a speed of 9.5 MIPS when using two threads, and 17.5 MIPS when using four. Comparing with the *OSCI* base line version, this corresponds to a 1.75× and a 3.23× speedup, respectively.

VI. Conclusion

This work has presented *SystemC-Link*, a simulation framework that enables time-decoupled parallel simulation for SystemC based simulators. While its main focus lies on boosting simulation performance, it also supports integration of legacy components that were not developed for parallel simulators. A speedup of 3.2× is achieved for a benchmark of a contemporary ESL virtual platform, running real-world software such as the Linux kernel.

Future work will be directed towards improvement of the inter-segment communication facilities. Special focus will be put towards achieving deterministic behavior during transmission of TLM transactions. Further performance improvements are also possible by extending the segment scheduler with host cache awareness.

References

[1] IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*.

[2] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneder, P. Sasidharan, and S. Singh. The next generation of virtual prototyping: Ultra-fast yet accurate simulation of HW/SW systems. In *Design, Automation and Test in Europe Conference (DATE)*, 2015.

[3] W. Chen, X. Han, and R. Dömer. May-happen-in-parallel analysis based on segment graphs for safe ESL models. In *Design, Automation and Test in Europe Conf. (DATE)*, 2014.

[4] M.-K. Chung, J.-K. Kim, and S. Ryu. SimParallel: A high performance parallel SystemC simulator using hierarchical multithreading. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2014.

[5] R. Dömer, W. Chen, and X. Han. Parallel discrete event simulation of transaction level models. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012.

[6] R. S. Khaligh and M. Radetzki. Modeling constructs and kernel for parallel simulation of accuracy adaptive TLMs. In *Design, Automation and Test in Europe Conference (DATE)*, 2010.

[7] S. Kristiansson. OpenRISC Linux [online; accessed 5/2015]. https://github.com/openrisc/linux.

[8] M. Moy. Parallel programming with SystemC for loosely timed models: a non-intrusive approach. In *Design, Automation and Test in Europe Conference (DATE)*, 2013.

[9] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2010.

[10] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1992.

[11] C. Roth, S. Reder, O. Sander, M. Hübner, and J. Becker. A framework for exploration of parallel SystemC simulation on the single-chip cloud computer. In *Conference on Simulation Tools and Techniques (SIMUTOOLS)*, 2012.

[12] 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 *Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2013.

[13] J. Tandon. The OpenRISC processor: Open hardware and linux. *Linux*, (212), 2011.

[14] N. Ventroux, J. Peeters, T. Sassolas, and J. C. Hoe. Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014.

[15] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi. SAGA: SystemC acceleration on GPU architectures. In *Proceedings of the 49th Design Automation Conference (DAC)*, 2012.

[16] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto. Time-decoupled parallel SystemC simulation. In *Design, Automation and Test in Europe Conf. (DATE)*, 2014.