2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Integrating Parallel SystemC Simulation into Simics® Virtual Platform

Daniel Mendoza and Ajit Dingankar, Intel Corporation, Folsom, USA

Zhongqi Cheng and Rainer Doemer, University of California, Irvine, USA

***Abstract*—The SystemC library is widely used to model and simulate system-level designs at an early stage for functional and performance analysis. Simics® is a tool for development and simulation of virtual platforms and is used to enable software development to be done earlier in the product development process. With the introduction of the Simics SystemC Library in Simics 5, it supports IP block, device and subsystem models developed in SystemC. In this paper, we demonstrate the integration of a novel engine for parallel SystemC simulation, called RISC (Recoding Infrastructure for SystemC), into Simics. We show promising experimental results using RISC, with speedups up to 212x.**

Keywords*— *SystemC; parallel simulation; Simics; multi-threading; conflict analysis; virtual platform

I. INTRODUCTION

The Wind River Simics® virtual platform framework [1] is used to develop full system simulators for the early development and test of software as a primary use-case. It is applied to a variety of virtual platform tasks such as pre-silicon software development, hardware validation and BIOS regression testing. Simics supports system models written in C, C++, Python, DML, and SystemC, and allows users to instantiate multiple sub-system models written in different languages into a single simulation. In this paper we focus on SystemC [2] models in Simics and introduce a new way to enable parallel execution of SystemC threads in Simics simulations.

Simics features the Simics SystemC Library [3], which enables users to co-simulate SystemC models inside Simics. The Simics SystemC Library not only handles the scheduling and synchronization of SystemC models with other Simics devices in a Simics co-simulation, but also provides users the software facilities to enable communication between SystemC models and Simics devices.

A. SystemC Multithreading

SystemC threads in Simics are managed by a standard SystemC kernel. One of the constraints of standard SystemC thread scheduling is that there can only be one thread active at a time [13], and thus potential parallelism within the model cannot be exploited during simulation. For speedup, standard SystemC offers temporal decoupling [4], which allows the developer to define a time interval in which synchronization points are postponed. The intent of temporal decoupling is to decrease the amount of context switching between threads for increased speedup. However, the gained speedup comes at the cost of potential simulation inaccuracy.

B. Automatic Out-of-Order Parallel SystemC Multithreading

With the proliferation of multicore hosts, enabling parallel multithreading is an attractive way to reduce simulation time. We introduce the Recoding Infrastructure for SystemC (RISC) [5] as an advanced technology to automatically exploit parallelism at the SystemC thread level. RISC has been developed as a research project at the University of California, Irvine, sponsored by Intel [6]. We describe its integration into Simics simulations without any manual manipulation of the source code or loss of simulation accuracy. RISC features both a dedicated SystemC compiler that uses static analysis to identify potential data conflicts between SystemC threads and an out-of-order parallel simulator that uses the data conflict analysis to make quick scheduling decisions for parallel multithreading. In many cases RISC has been successful and has achieved up to 212x speedup [7].

Figure 1 shows the difference in simulation speeds between standard SystemC and RISC on a "best case" for purposes of illustration. Threads *th1, th2*, *th3*, and *th4* are independent of one another, and thus can safely run simultaneously. However, a SystemC simulation managed by standard SystemC kernel only has one thread active at a time, and thus those threads are not run in parallel. On the contrary RISC's conflict analysis can determine that threads *th1*, *th2*, *th3*, and *th4* are independent, and runs those threads in parallel. As a result, RISC simulation is much faster than the standard SystemC approach. Furthermore, RISC preserves the standard SystemC semantics to the maximum extent and runs simulations without loss of accuracy [8].

The non-determinism inherent in the standard SystemC simulator is also present in RISC, but it is important to note that the RISC scheduler does not introduce any additional non-determinism due to multi-threading. Hence the impact on Simics simulation (which is highly deterministic) is the same as with the integration of any regular SystemC models.

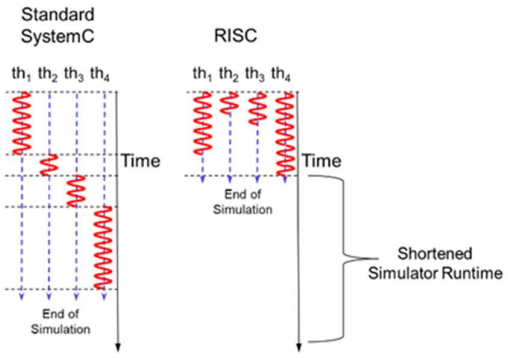

Figure 1: Juxtaposition between standard SystemC and RISC thread activity over time.

This paper discusses the integration of RISC into Simics and shows two successful example Simics simulations using RISC for scheduling SystemC threads.

C. *Related Work*

Other attempts to parallelize SystemC simulations have been conducted. [9] proposes a method for manual translation of the sequential model into a parallel model. However, this technique requires the user to understand and avoid conflicting variable accesses between threads. [10] and [11] offer tool flows for parallel simulation of only RTL SystemC models. In contrast, the RISC solution is not restricted to RTL models.

II. RECODING INFRASTRUCTURE FOR SYSTEMC

RISC automatically parallelizes a SystemC model via static conflict analysis at compile time. In this section, we first introduce the approach of how RISC enables multi-core exploitation and then discuss RISC integration into Simics.

A. *RISC technique for parallel SystemC simulation*

The RISC flow for compilation of a SystemC model is shown in Figure 2. Once the source code for an input model has been identified, RISC's first step in compilation of the model is to use the ROSE compiler [12] to build an Abstract Syntax Tree (AST). By leveraging the ROSE compiler, RISC has the support to generate the AST, analyze the AST, and transform the input SystemC model into an out-of-order parallel SystemC model. The AST is analyzed to generate an internal representation of the input model. The internal representation recognizes SystemC primitives such as port connectivity and module hierarchy. RISC then analyzes the internal representation of the model to build a segment graph data structure [13]. The segment graph data structure is used to determine the potential conflicts between the simulation threads and then the source code is transformed into a safe out-of-order parallel SystemC model. The instrumented code is finally compiled and linked to the RISC simulation library for out-of-order parallel simulation.

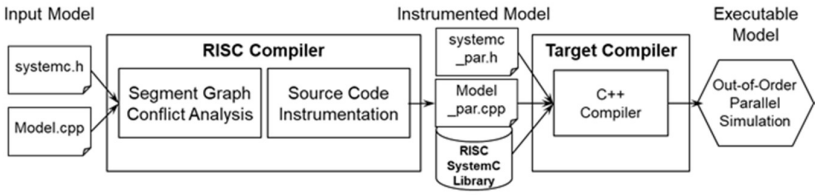

Figure 2: RISC Compiler and Simulator Flow [14]

Segment Graphs (SG) are directed graphs that represent the possible flow of execution of SystemC threads. A node in a segment graph represents a segment of execution in a SystemC thread. Each `wait` statement in the SystemC code indicates the beginning of a segment in the segment graph. For instance, see Figure 3, which exemplifies a piece of SystemC code and its corresponding segment graph. Conditional statements in the code indicate branches and every possible sequence of `wait` statements in a thread is included in the segment graph. This is exemplified in Figure 3 as after the first `wait` statement, the graph has two pathways with one pathway having an extra `wait` statement.

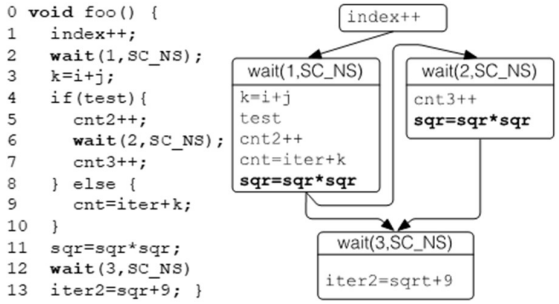

Figure 3: Example SystemC Code and corresponding Segment Graph [13]

Each of the segment's memory accesses are analyzed to determine if segments are potentially in conflict with one another. RISC assigns a segment ID for each segment and then determines possible conflicts. The conflict information is stored as a table that is passed to the simulator and the simulator then uses the table to determine if code segments are in conflict with one another so that it can make fast scheduling decisions during simulation. For instance consider the segment graph and its corresponding conflict table drawn in Figure 4. Segment 2 contains two reads to variables *a* and *b*, one write to variable *x*, and one read/write to variable *z*. Segment 3 has two reads to variables *a* and *b* and two writes to variable *x* and *y*. Since Segment 2 and Segment 3 write to the same variable *x*, RISC determines that these two segments are in conflict with one another. The table in Figure 4 indicates the conflicts between segments by the red color. The green entries in the table indicate that those segments are never in conflict with one another and thus can be executed simultaneously.

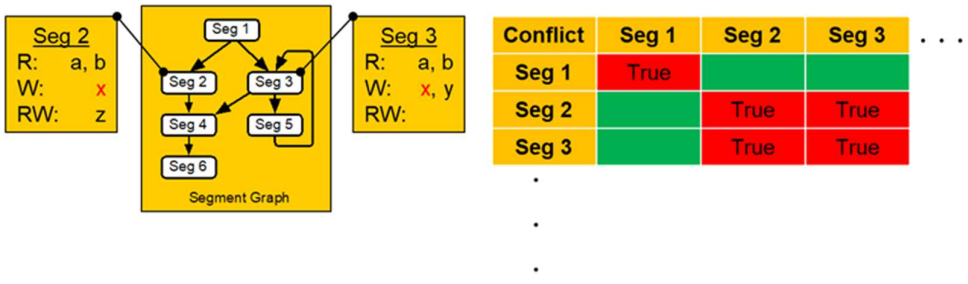

Figure 4: Segment Graph Based Conflict Analysis [8]

RISC automatically parallelizes the execution of SystemC threads for simulation speedup. However, some SystemC models contain more exploitable parallelism than others. For instance, SystemC models that contain threads that frequently wake up and go to sleep may be parallelizable, but the time reduction in the simulation from

parallelism may be limited by large context switching overhead. Models that benefit greatly from thread-level parallelism contain many threads with significant computational load and minimal dependency on one another. Currently, the RISC approach is best fit for high abstraction level modeling such as transaction-level modeling (TLM). Modeling at lower levels of abstraction such as register-transfer level (RTL) modeling remain to be investigated. A list of SystemC constructs that are currently supported by RISC can be found in [14].

RISC enables users to reduce simulation time through the use of parallel multithreaded algorithms. A user who has deep understanding of the functional algorithms used in a model may redesign the model to leverage parallel multithreading. Furthermore, more fine grained parallelism may be exploited through utilizing data level parallelism constructs. Specifically, vectorization involves multiple hardware units executing the same operation simultaneously on different data. Users may insert `#pragma simd` into sections of the source code that can safely utilize the vectorization optimization. Through leveraging both thread and data level parallelism RISC has been able to achieve 212x speedup [7].

B. RISC Integration into Simics

The Simics SystemC Library handles all SystemC aspects in a Simics simulation. When a Simics user desires to integrate a plain SystemC model into a Simics simulation, the user must first create an `Adapter` class that is used as a wrapper for the SystemC model. The adapter class simply instantiates the SystemC model into the Simics simulation and sets up the facilities for communication between Simics devices and the SystemC model.

Each SystemC device in a Simics simulation is linked to its own SystemC kernel. In a Simics simulation with an instantiated SystemC device, Simics will periodically interface with the Simics SystemC Library to synchronize the SystemC model's simulation time with the global simulation time maintained by Simics. The Simics SystemC Library makes calls to the SystemC kernel to run the local SystemC simulations of each SystemC device.

Typically, a standard sequential SystemC kernel is linked to a SystemC device. However, we aim to link the RISC kernel in order to enable out-of-order parallel SystemC thread scheduling. Simics provides compilation scripts for SystemC devices that contain configurable flags so that a model developer can simply set a compilation flag in order to link the RISC kernel instead of a standard SystemC kernel. Figure 5 exemplifies the switch between the standard SystemC kernel (blue) and the RISC kernel (red). The Simics SystemC Library calls `sc_start`, which then schedules threads in the SystemC kernel. In order to enable out-of-order parallel SystemC thread scheduling, the user simply links the RISC kernel with the devices such that the Simics SystemC Library will interface with the RISC kernel instead of the standard SystemC kernel.

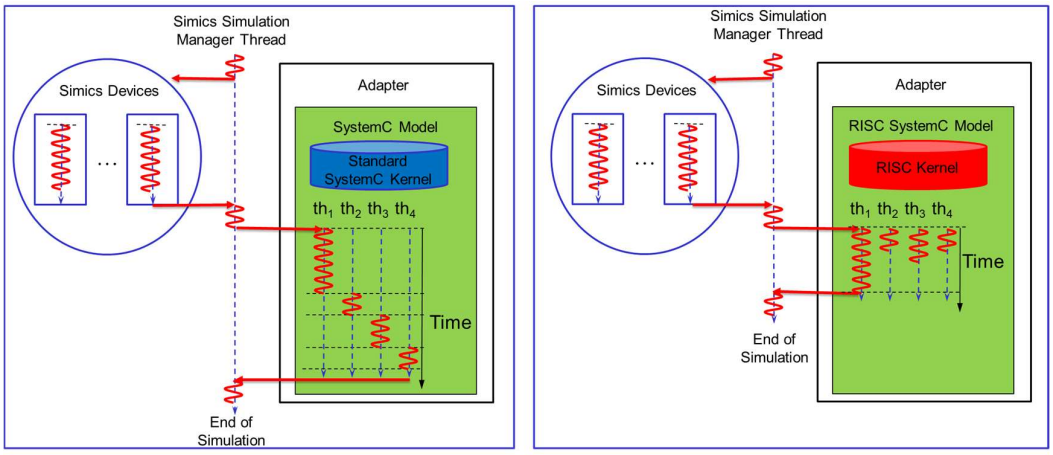

Figure 5: Two different Simics simulations of the same model with the left-side using a standard SystemC kernel and the right-side featuring RISC kernel for out-of-order parallel multithreading of SystemC threads

Gaskets are media that use SystemC TLM-2.0 for communication between the SystemC model and other Simics devices. The RISC research team is currently developing the facilities to support TLM-2.0 constructs in SystemC

models. The team has had success in this endeavor and official support for TLM-2.0 in RISC is planned to be established by the end of 2019.

Nevertheless, RISC integration into Simics is possible because of extensible and flexible nature of gaskets. Users of Simics can instantiate gaskets using SystemC TLM-2.0 constructs defined in the Simics SystemC Library. However, in the interest of avoiding TLM-2.0, we've implemented a few custom gaskets that do not contain TLM-2.0 constructs. To implement these special non-TLM-2.0 gaskets, we took the implementation of a gasket in the Simics SystemC Library as a starting point and replaced the TLM-2.0 sockets with traditional ports and channels. Because we have not modified the actual functionality of the gasket, the simulation of a model that uses the special implementation of the gaskets will exhibit the same behavior as a simulation of the model that uses the TLM-2.0 gaskets provided in the Simics SystemC Library.

The approach for a Simics device to send a TLM payload to a SystemC device, and vice versa, is slightly different. Figure 6 shows how a Simics device sends a TLM payload to a SystemC device. As illustrated, the Simics device requests to send a payload to a certain address where a SystemC device is located in the Simics device mapping. The Simics Device interfaces with Simics which then redirects the access to the gasket connected to the SystemC model that is located at the specified address. Figure 7 exemplifies how a SystemC device sends a TLM payload to a Simics Device. The SystemC model simply interfaces with the gasket through a port or `b_transport` call, and the rest of the operation is handled by Simics.

Current limitations of RISC include support only of TLM-1.0 while TLM-2.0 and RTL modeling are in development. Furthermore, Simics gaskets and compilation flow must be customized to allow the RISC SystemC kernel to manage SystemC threads in a Simics simulation.

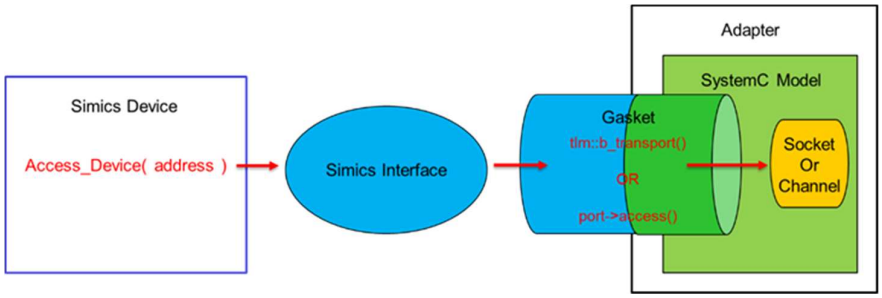

Figure 6: Simics-to-SystemC communication

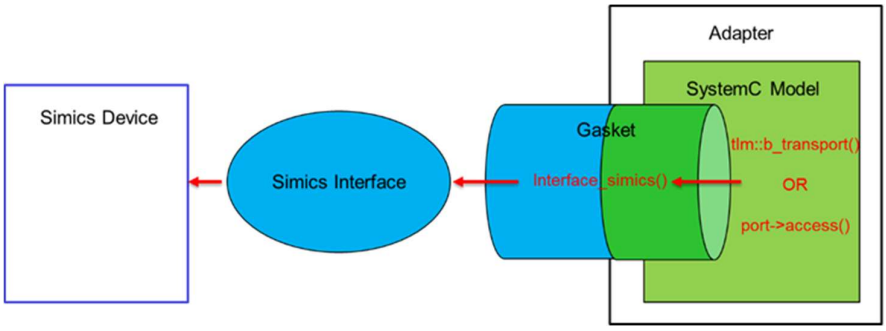

Figure 7: SystemC-to-Simics communication

III. EXPERIMENTS AND RESULTS

In this section we show two applications that exemplify RISC integration into Simics. Both example simulations feature a SystemC device communicating with a Simics device.

A. Mandelbrot Renderer

The Mandelbrot Renderer is an example application that resembles the computations performed in a graphics rendering pipeline. The goal of the application is to generate 20 Mandelbrot frames with varying zoom factor.

Figure 8 shows the runtime statistics of the RISC Mandelbrot Renderer model. Through the use of thread and data level parallelism, the peak speedup with RISC is 212x [7].

# slices	seq.simd (M)	par (N)	par.simd (NxM)
1	6.9153	1.001	6.94462
2	6.9176	1.682	11.7748
4	6.9183	3.042	21.1943
8	6.9176	5.845	40.0967
16	6.9167	11.37	72.5175
32	6.9124	21.32	137.213
64	6.896	41.07	208.413
128	6.8948	46.29	212.957
256	6.8736	49.9	194.187

Figure 8: RISC Simulation Performance of Mandelbrot Renderer on a 60 core Intel® Xeon Phi™ host [7].

We have instantiated the Mandelbrot Renderer model into a Simics simulation. Figure 9 illustrates the Simics device mapping in the Mandelbrot Renderer Simulation. In this simulation, a Simics Vacuum virtual platform communicates data back and forth from the SystemC device through Simics RAM. The Vacuum writes coordinates to the RAM device and the SystemC model reads the coordinates via Simics-to-SystemC communication and then generates a Mandelbrot frame corresponding to the input coordinates. Once the SystemC device is finished rendering the Mandelbrot frame, the device will use SystemC-to-Simics communication to indicate to the Vacuum that it is ready to process another coordinate pair.

The significance of this Simics simulation is that it features both SystemC-to-Simics and Simics-to-SystemC communication via gaskets. Since these gaskets are the only media in Simics for communication between SystemC devices and Simics devices, this example shows that all Simics-to-SystemC and SystemC-to-Simics communication is feasible with RISC.

Figure 10 exemplifies the simulation statistics of the Simics simulation running on an 8 core Intel® Xeon® Processor E5-2670 (2.60GHz) in comparison to the peak speedup of the simulation executing on a 60 core Intel® Xeon Phi™ from parallel multithreading. Due to the limitations of the host running the Simics simulation, data level parallelism could not be exploited. We can observe that both the simulation in Simics and without Simics exhibit similar efficiency of approximately 80%. Thus the speedup of 6.40x in the Simics simulation is consistent with the statistics of the simulation without Simics since the Simics simulation was executed on an 8 core host.

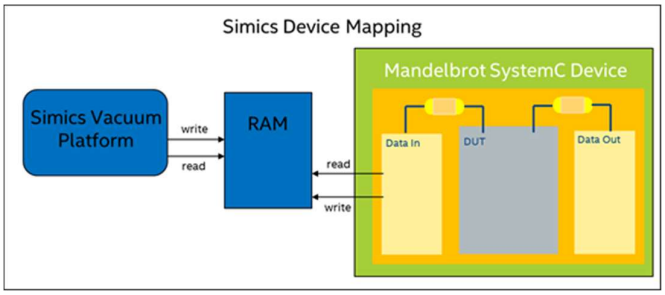

Figure 9: Simics Device set up for Mandelbrot Renderer example

	Standard SystemC Runtime	Standard SystemC CPU Utilization	RISC Runtime	RISC CPU Utilization	RISC Speedup	# of cores	Efficiency
Simics	59.93s	99%	9.37s	641%	6.40x	8	80.0%
Without Simics	394.2s	99%	7.9s	4902%	49.90x	60	83.2%

Figure 10: Simulation statistics of Mandelbrot Renderer simulation in Simics and plain SystemC

B. Panorama Graphics Filter

The Panorama Graphics Filter simulation features a SystemC model that performs an image transformation on input photos. Specifically, the SystemC model inputs images and attempts to identify the moving objects in the input stream and then generates output images without the moving objects [15].

Figure 11 illustrates the device mapping for the simulation of the Panorama Graphics Filter. In this example, a Linux-based virtual platform called QSP communicates with the SystemC device via a PCI bus. This example demonstrates RISC's compatibility with a practical and realistic Simics simulation.

The speedup is not as drastic as the previous example due to dependency of the SystemC model. However, the simulation in Simics and outside Simics exhibit similar efficiency which indicates that Simics integration with RISC is not the bottleneck for speedup.

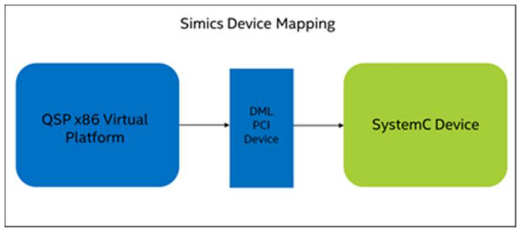

Figure 11: Simics Device set up for Panorama Graphics Filter example

	Standard SystemC Runtime	Standard SystemC CPU Utilization	RISC Runtime	RISC CPU Utilization	RISC Speedup	# of cores	Efficiency
Simics	50.11s	77%	27.20s	141%	1.84x	8	23.0%
Without Simics	73.42s	91%	49.47s	137%	1.48x	8	18.4%

Figure 12: Simics simulation statistics of Panorama SystemC Model with Linux-based VP on an 8 core Intel® Xeon® CPU E5-2670 (2.60GHz)

IV. CONCLUSION

Parallel multithreading is a readily available avenue for optimization today due to the prevalence of multicore hosts. In this paper, we introduced RISC, a dedicated SystemC compiler and simulator that automatically and safely parallelizes SystemC threads. We have presented RISC integration into a Simics virtual platform and showed two successful cases of a Simics simulation leveraging RISC for its SystemC thread management. Each of the case studies exhibits significant speedup. The two example Simics simulations using RISC demonstrate that the combination of RISC and Simics is feasible and valuable. Additionally, the examples show that RISC can be used in a practical and realistic Simics simulation environment.

A. Future Work

The RISC development team is currently working on RISC support for SystemC TLM-2.0 constructs and is expected to officially support SystemC-TLM2.0 by the end of 2019. We're planning to further facilitate RISC integration into Simics through the development of a Simics SystemC simulation containing SystemC TLM-2.0 constructs that is managed by the RISC kernel. Furthermore, RISC supports debugging facilities such as tracing. We plan to investigate how to support the Simics debugging features with RISC integration.

ACKNOWLEDGMENTS

This work has been supported in part by substantial funding from Intel Corporation for the project titled " Scaling the Recoding Infrastructure for Parallel SystemC Simulation". The authors thank Intel Corporation for the valuable support.

REFERENCES

[1] "Wind River Simics," Wind River Systems, 2019. [Online]. Available: https://www.windriver.com/products/simics/.

[2] IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2011, 2011.

[3] J. Engblom, A. Hedstroem, X. Wang and H. Zeffer, "Integrating Different Types of Models into a Complete Virtual System - The Simics SystemC* Library," in Design and Verification Conference (DVCon) Europe 2016, Munich, Germany, 2016.

[4] J. Engblom, "Temporal Decoupling - Are 'Fast' and 'Correct' Mutually Exclusive," in Design and Verification Conference Europe (DVCon Europe) 2018, Munich, Germany, 2018.

[5] "Recoding Infrastructure for SystemC (RISC)," Center for Embedded & Cyber-Physical Systems, 2019. [Online]. Available: http://www.cecs.uci.edu/~doemer/risc.html

[6] "Scaling the Recoding Infrastructure for Parallel SystemC Simulation," Intel Software and Solutions Group Research Program, 2018.

[7] G. Liu, T. Schmidt and R. Doemer, "Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation," in Design Automation Conference (DAC) 2017, Austin, TX, 2017.

[8] R. Doemer, "On the Limits of Standard-compliant Parallel Simulation of the IEEE SystemC Language," in Forum on specification & Design Languages (FDL) 2018, Munich, Germany, 2018.

[9] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras and A. Hoffman, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in Design, Automation and Test in Europe (DATE), 2016.

[10] N. Bombieri, F. Fummi and S. Vinco, "On the Automatic Generation of GPU-oriented Software Applications from RTL IPs," in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013.

[11] C. Roth, S. Reder, H. Bucher, O. Sander and J. Becker, "Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures," in Digital System Design (DSD), 17th Euromicro Conference, 2014.

[12] D. Quinlan, "ROSE: Compiler Support for Object-Oriented Frameworks," Parallel Processing Letters, vol. 10, pp. 215-226, 2000.

[13] R. Doemer, G. Liu and T. Schmidt, "Parallel Simulation," in Handbook of Hardware/Software Codesign by S. Ha and J. Teich, Dordrecht, Netherlands, Springer, 2017, p. Chapter 17.

[14] G. Liu, T. Schmidt, Z. Cheng, D. Mendoza and R. Doemer, "RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset," Center for Embedded and Cyber-Physical Systems, Irvine, CA, 2018.

[15] T. Azumi, Y. Samei, Y. Hara-Azumi, H. Oyama and R. Doemer, "TECSCE: HW/SW Codesign Framework for Data Parallelism Based on Software Component," in International Embedded Systems Symposium "Embedded Systems: Design, Analysis and Verification" (ed. G. Schirner, M. Goetz, A. Rettberg, M. Zanella, F. Rammig), Springer, Paderborn, Germany, 2013.