RISC: Recoding Infrastructure for SystemC, Open Source Framework for Parallel Simulation

Rainer Dömer*, Zhongqi Cheng*, Daniel Mendoza* and Ajit Dingankar†
*Center of Embedded and Cyber-physical Systems, University of California, Irvine
†Intel Corporation

Abstract—**The IEEE SystemC language is widely used in industry and academia to model and simulate system-level designs. Despite the availability of multi- and many-core host processors, however, the Accellera reference simulator is still based on sequential discrete event simulation, utilizing only a single core at any time. While many advanced parallel simulation approaches have been proposed, most require modification of the SystemC source code so that the model is free from parallel access conflicts and rely on the designer to manually perform this difficult transformation.**

The Recoding Infrastructure for SystemC (RISC) project addresses the parallel SystemC simulation problem with automatic compiler-based analysis and source code transformation. A dedicated SystemC compiler and corresponding parallel simulator provide safe static analysis and recoding, and thus automatically achieve fast parallel simulation of SystemC models.

We share the RISC framework as open source in order to enable easy evaluation, foster collaboration, and further extend our proof-of-concept implementation.

I. Introduction

The IEEE standard SystemC language [1], is widely used for the specification, modeling, validation and evaluation of Electronic System Level (ESL) models. The Accellera Systems Initiative maintains not only the official SystemC language definition, but also provides an open source proof-of-concept library that can be used to simulate SystemC design models [2]. However, implementing the classic scheme of Discrete Event Simulation (DES), this reference simulator runs sequentially and cannot utilize the parallel computing resources available on multi- and many-core processor hosts. This severely limits the execution speed of SystemC simulation.

In order to provide faster execution, Parallel Discrete Event Simulation (PDES) [3] techniques can be applied. While significant obstacles exist specifically for the SystemC language [4], many parallel simulation approaches have been proposed [5]–[11]. Beyond these synchronous PDES techniques, *out-of-order* PDES [12] is even more aggressive. By localizing the simulation time to individual threads and carefully handling events at different times, the simulator engine can issue threads in parallel and *ahead of time*, following a partial ordering without loss of accuracy. This results in better exploitation of the available parallelism and thus maximum simulation speed.

The *Recoding Infrastructure for SystemC (RISC)* project described in this paper implements out-of-order PDES for the IEEE SystemC language as open source. Specifically, RISC provides a dedicated SystemC compiler and corresponding out-of-order parallel simulator [13]–[15]. Compared to the other approaches, RISC automatically analyzes the SystemC source code, identifies all potential race conditions, and then instruments the model to prevent any conflicts. This transformation does not require any manual recoding or application-specific knowledge.

We share our RISC proof-of-concept implementation with the EDA community as an open source software project in order to facilitate evaluation, promote parallel SystemC simulation, and achieve fruitful collaboration [16], [17].

II. RISC Framework

While the RISC software framework may be used for many other analysis and transformation tasks on SystemC models, parallel simulation is the main purpose. To perform semantics-compliant parallel simulation with out-of-order scheduling, we introduce a dedicated SystemC compiler that works hand in hand with a new simulator. This is in contrast to the traditional SystemC simulation flow where a SystemC-agnostic C++ compiler includes the SystemC headers and links the design model directly against the Accellera reference library.

As shown in Figure 1, the RISC compiler acts as a frontend that processes the input model and generates an intermediate model with special instrumentation for conflict-free parallel execution. The instrumented model is then linked against the extended RISC SystemC library by the target compiler (a regular C++ compiler, such as GNU `gcc` or Intel `icpc`) in order to produce the output executable model. Out-of-order parallel simulation is then performed simply by running the generated executable model.

From the user perspective, we simply replace the regular C++ compiler with the SystemC-aware RISC compiler (which in turn calls the underlying C++ compiler). Otherwise, the overall SystemC validation flow remains the same as the traditional tool flow. Simulation is just faster due to the parallel execution. Note also that this process is fully automated. No user interaction or manual code transformation is necessary.

A. RISC Compiler

In order to produce a safe parallel model, the RISC compiler performs three major tasks, namely Segment Graph construction, conflict analysis, and finally source code instrumentation.

1) Segment Graph Construction: A Segment Graph (SG) [12] is a directed graph that represents the source code segments executed during the simulation between scheduling

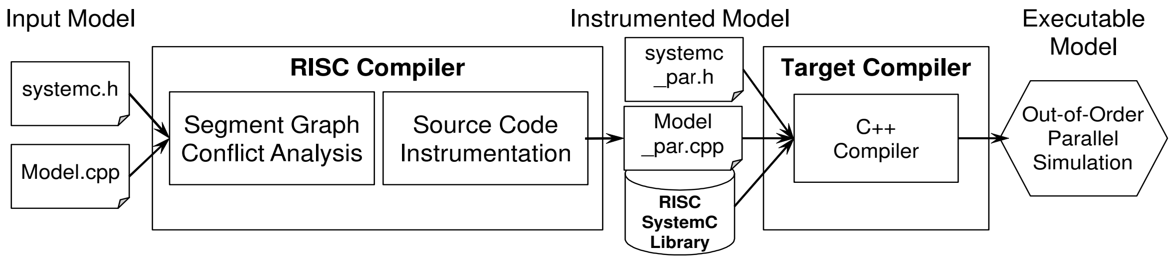

Fig. 1: RISC tool flow for out-of-order parallel simulation of SystemC models [14]

steps. More specifically, every segment is associated with a corresponding scheduler entry point, namely a `wait` statement in SystemC. All other statements in the SystemC source code become part of those segment nodes where they are executed when the `wait` statement resumes its execution.

The segment graph construction is a fully automatic but complex process which we will not describe here (see [12] for detailed coverage). However, the RISC compiler must parse the SystemC input model first into an Abstract Syntax Tree (AST). Since SystemC is syntactically regular C++ code, RISC relies here on the ROSE compiler infrastructure [18]. The ROSE Internal Representation (IR) provides RISC with a powerful C/C++ compiler foundation that supports AST generation, traversal, analysis, and transformation.

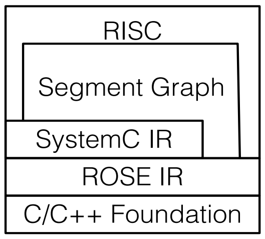

Fig. 2: Software stack of the RISC compiler [13]

As illustrated with the RISC software stack shown in Figure 2, the RISC compiler then builds on top of the ROSE IR a SystemC IR which accurately reflects the SystemC structures, including the module and channel hierarchy, port connectivity, and other SystemC-specific constructs. On top of the SystemC IR, the compiler architecture then builds the Segment Graph generator and data structures, as well as all other RISC analysis and transformation functions.

2) Conflict Analysis: The Segment Graph data structure serves as the foundation for segment conflict analysis. At run time, the scheduler in the simulator must ensure that every parallel thread to be issued has no conflicts with any other threads currently in the $READY$ and RUN queues. For this we use the RISC compiler to detect any possible conflicts between these threads already at compile time.

Potential conflicts in SystemC include data hazards, event hazards, and timing hazards, all of which may exist among the segments executed by the threads considered for parallel execution. Again, we refer to [12] for a detailed discussion of these hazards and their static or dynamic detection in RISC. However, we note that if the hazards would be ignored, this would lead to race conditions at run time and jeopardize the correctness of the SystemC simulation.

3) Source Code Instrumentation: As a result of the conflict analysis, the RISC compiler generates a set of conflict and timing tables that describe all possible hazards between any two threads. Using this conservative conflict information, the simulator can then at run-time quickly determine by a simple table look-up whether or not it is safe to issue a given thread in parallel or ahead of time.

As shown above in Figure 1, the RISC compiler and simulator work closely together. The compiler performs conservative conflict analysis and passes the analysis results to the simulator which then can make safe scheduling decisions quickly.

To pass information from the compiler to the simulator, we use automatic source code instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) code which the simulator can then safely rely on.

At the same time, the RISC compiler also instruments the SystemC `wait` statements with corresponding segment ID and furnishes user-defined channels with automatic protection against race conditions among communicating threads.

B. RISC Simulator

The RISC simulator supports out-of-order discrete event simulation (OoO PDES) [12] for fast SystemC simulation. In OoO PDES, we break the strict order of time (the synchronous barrier) by localizing time stamps to each thread. Since each thread has its own time stamp, the OoO PDES scheduler relaxes the event and simulation time updates, allowing more threads (at different simulation cycles) to run in parallel and ahead of time. This results in a higher degree of parallelism and thus higher simulation speed. We are using advanced static compile-time analysis to identify all such potential conflicts. Based on this information (a simple table look-up is sufficient), the OoO PDES scheduler can then at run-time quickly decide whether or not a set of threads has any conflicts with each other.

C. RISC Analysis and Transformation Tools

As an example of other SystemC analysis tools built on top of RISC, `visual` [19] enables the user to visualize the SystemC module hierarchy. It supports a graphical user interface implemented with the Gtk API and renders a specified SystemC source file's module hierarchy, which is drawn using the Cairo API. The tool obtains module data from the SystemC IR in the RISC software stack which contains information about nested modules and thus can recursively iterate through nested lists of child modules in order to obtain enough information to visualize the hierarchy of the entire SystemC source file. The input SystemC source file may contain thousands of lines of code which can make manually drawing a representation of the modules, ports, and channels described by the code a difficult and time-consuming task. Thus the `visual` tool was created to address this issue. It can automatically generate a visual representation of a SystemC model in a very short period of time. Figure 3 shows the module visualization of a Canny edge detector application.

III. Experiments

We will now evaluate the performance of the RISC simulator. The following experiments show the speedup on an Intel Xeon Phi Coprocessor 5110P many-core architecture. The coprocessor contains 60 cores where each core has a vectorization unit of 512 bit. To obtain unambiguous measurements, we turn CPU frequency scaling off for all experiments.

A. Mandelbrot Renderer

The Mandelbrot renderer is a parallel video application to compute the Mandelbrot set. Basically, the Device under Test (DUT) hosts a number of renderer units. Each unit computes a different slice of the Mandelbrot image. At compile time, the user defines how many slices are available.

Figure 4 shows the simulation results [20]. Due to the minimal communication needs in this application, highest speedups are reached. The vectorization unit with 512 bit can execute up to eight double-precision floating-point operations in parallel. A speedup M of 6.9x is achieved. The thread-level parallelization increases strongly on the 60 cores with a speedup N of 50x. Afterwards, the speed slows down due to the 60 physical cores and use of hyper-threads. Notably, the combination of the thread and data level parallelization $N \times M$ generates a speedup of up to 212x.

IV. RISC Open Source Project

We make the Recoding Infrastructure for SystemC (RISC) described in this article freely available online as a software artifact. Generally, an artifact is a software program together with an applicable data set and test suite that accompanies a research publication for the purpose of independent evaluation[1]. The point here is that the proposed algorithms and data structures are made available as proof-of-concept implementation and can be used and evaluated by others. Experimental results may be replicated and validated. The proposed approach can also be compared against related work, and in the presence of source code, even be extended. Otherwise, great challenges are posed in repeatability [23].

Specifically, the presented RISC compiler and simulator are available as open source on the web [15] and can be used without restrictions (BSD license terms). RISC can be downloaded in both source code and binary format.

A. Open Source Code and Documentation

In its current version [16], the RISC open source package consists of approximately 162000 lines of code and includes the C++ source code for the RISC compiler and simulator, Linux build scripts and installation instructions, as well as comprehensive documention of the compiler and simulator APIs and tool manual pages. Example SystemC models, such as an abstract DVD player and the Mandelbrot renderer, and a regression test bench are included as well.

Given a suitable Linux platform[2], the RISC source code package can be easily installed and then tested. After downloading and adjusting the installation `Makefile`, a simple `make all` command builds and installs the RISC framework and runs several demo examples. The user can then fully evaluate the software with other SystemC examples and even extend our proof-of-concept implementation with new features.

B. Binary Image for "Plug-and-Play" Evaluation

For a quick test run without compilation and installation, we also provide a Docker container [17] for using RISC in "plug-and-play" fashion. The Docker image contains RISC (and all needed libraries) in binary format and allows the user to test it with just a few Linux commands, as shown in Figure 5.

```
bash# docker pull ucirvinelecs/risc
bash# docker run -it ucirvinelecs/risc
[dockeruser]# cd demodir
[dockeruser]# make test
```

Fig. 5: Linux commands to use RISC in a Docker container

V. Conclusion

The Recoding Infrastructure for SystemC (RISC) provides an automatic compiler-based framework to analyze and simulate IEEE SystemC models in parallel. In particular, we have introduced the RISC compiler and simulator. Using automatic conflict analysis based on Segment Graph (SG) abstraction, OoO PDES can execute threads safely in parallel and out-of-order (ahead of time) and thus achieves fastest simulation speed but nevertheless maintains the classic SystemC modeling semantics. In order to foster collaboration in the EDA community, we provide the RISC framework as a free open source artifact for full evaluation and possible extension.

For the future, we intend to expand our open source efforts and hope to involve other members of the EDA community to use, evaluate, and extend the RISC framework.

[1]Because of its importance, artifact evaluation has been adopted as integral part of the review process in several computer science areas, such as Software Engineering and Programming Languages [21], [22].

[2]RedHat Enterprise and CentOS Linux version 6 and 7 are verified to work.

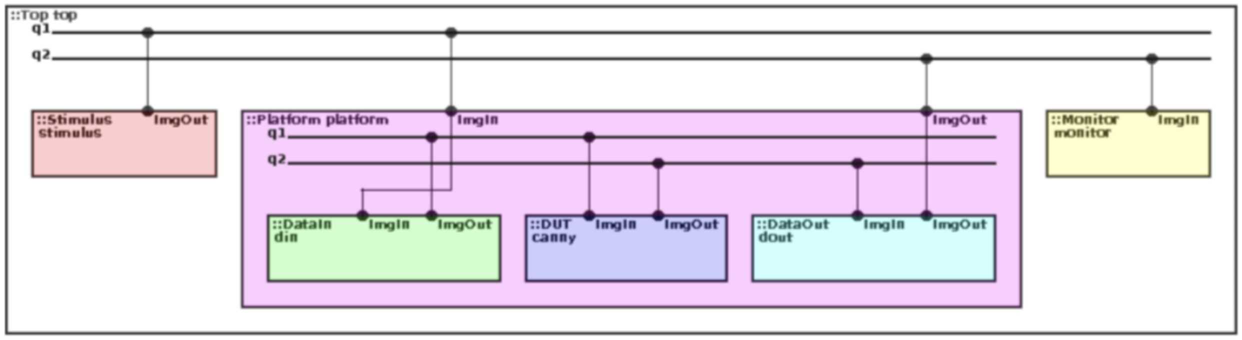

Fig. 3: Module hierarchy visualization of a SystemC model of a Canny edge detector [19]

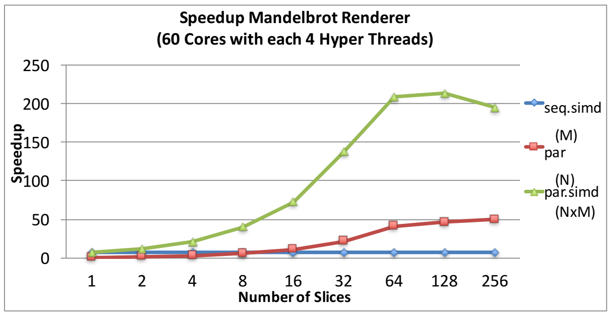

Fig. 4: Speedup of the Mandelbrot Renderer [20]

Acknowledgment

This work has been supported in part by substantial funding from Intel Corporation for two projects titled “Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures” and “Scaling the Recoding Infrastructure for Parallel SystemC Simulation”. The authors thank Intel Corporation for the valuable support.

References

[1] IEEE Computer Society, *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*. IEEE, New York, USA, 2011.

[2] Accellera Systems Initiative, “Core SystemC Language and Examples,” http://accellera.org/downloads/standards/systemc.

[3] R. Fujimoto, “Parallel Discrete Event Simulation,” *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct 1990.

[4] R. Dömer, “Seven obstacles in the way of standard-compliant parallel SystemC simulation,” *IEEE Embedded Systems Letters*, vol. 8, no. 4, pp. 81–84, Dec. 2016.

[5] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, “Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines,” in *PADS ’09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 80–87.

[6] D. Yun, J. Kim, S. Kim, and S. Ha, “Simulation Environment Configuration for Parallel Simulation of Multicore Embedded Systems,” in *Proceedings of the Design Automation Conference (DAC)*, 2011, pp. 345–350.

[7] R. Sinha, A. Prakash, and H. D. Patel, “Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs,” in *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[8] W. Chen, X. Han, and R. Dömer, “Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment,” *IEEE Design and Test of Computers*, vol. 28, no. 3, pp. 20–31, May/June 2011.

[9] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, “SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments,” in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2016.

[10] N. Ventroux and T. Sassolas, “A New Parallel SystemC Kernel Leveraging Manycore Architectures,” in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2016.

[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] W. Chen, X. Han, C.-W. Chang, G. Liu, and R. Dömer, “Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 33, no. 12, pp. 1859–1872, Dec. 2014.

[13] R. Dömer, G. Liu, and T. Schmidt, “Parallel simulation,” in *Handbook of Hardware/Software Codesign*, S. Ha and J. Teich, Eds. Dordrecht: Springer Netherlands, 2017, pp. 1–32. [Online]. Available: https://doi.org/10.1007/978-94-017-7358-4_19-1

[14] G. Liu, T. Schmidt, Z. Cheng, D. Mendoza, and R. Dömer, “RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset,” Center for Embedded and Cyber-physical Systems, University of California, Irvine, Tech. Rep. CECS-TR-18-03, Sep. 2018.

[15] Center for Embedded and Cyber-physical Systems, “Recoding Infrastructure for SystemC (RISC),” http://www.cecs.uci.edu/~doemer/risc.html.

[16] ——, “RISC Release version 0.5.0,” http://www.cecs.uci.edu/~doemer/risc.html#RISC050.

[17] ——, “RISC Docker Container,” https://hub.docker.com/r/ucirvinelecs/risc050/.

[18] D. J. Quinlan, “ROSE: Compiler support for object-oriented frameworks,” *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[19] D. Mendoza and R. Dömer, “A Tool for Visualization of SystemC Models,” Center for Embedded and Cyber-physical Systems, University of California, Irvine, Tech. Rep. CECS-TR-17-06, Nov. 2017.

[20] T. Schmidt, G. Liu, and R. Dömer, “Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation,” in *Proceedings of the Design Automation Conference (DAC)*, Jun. 2017.

[21] Shriram Krishnamurthi, “Artifact Evaluation Process,” http://www.artifact-eval.org/.

[22] Evaluate Collaboratory, “Artifact Evaluation,” http://evaluate.inf.usi.ch/artifacts.

[23] S. Krishnamurthi and J. Vitek, “The real software crisis: Repeatability as a core value,” *Commun. ACM*, vol. 58, no. 3, pp. 34–36, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2658987