UNIVERSITY OF CALIFORNIA,
IRVINE

Advancing Compiler and Simulator Techniques for Highly Parallel Simulation of Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Zhongqi Cheng

Dissertation Committee:
Professor Rainer Dömer, Chair
Professor Mohammad Al Faruque
Professor Aparna Chandramowlishwaran

2020

DEDICATION

To Lindsay, for her unconditional love and support.
To my parents, for everything they have done —and still do — for me.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

It would have been impossible for me to pursue the doctorate degree without the help and support from the amazing people around me. Here, I would like to give particular mentions to some of whom for encouraging and inspiring me in this long journey.

First and foremost, I would like to express my gratitude to my Masters and Doctorate advisor, Professor Rainer Dömer, for his guidance and support during my graduate study in University of California, Irvine. When I just joined his team, I was not aware of what academic research was about. His insights and broad knowledge of embedded system design methodologies helped me a lot in finding my own research directions. He also taught me how to do high quality research and how to balance between academic values and engineering efforts, which are invaluable for my whole career. I also appreciate most his kind and patient personalities, which created an relaxing and enjoyable environment in the lab. Such working environment was very helpful for increasing the efficiency of my research.

I would also like to thank Professor Mohammad Al Faruque, Professor Aparna Chandramowlishwaran and Professor Kwei-Jay Lin. I took their courses during my Masters and learned a lot about embedded systems and parallel computing knowledge and techniques. This built a solid foundation for my Doctorate study. They also spent their precious time being my committee members and provided me constructive advice and feedback on my dissertation.

In addition, I would like to thank my team members, Guantao Liu, Tim Schmidt, Daniel Mendoza and Emad Arasteh. Guantao and Tim provided me endless help on understanding the existing technologies of our group when I just joined and shared frequently their thoughts and ideas about research in this area. Daniel worked closely with me on several projects and I appreciate very much his cleverness and hardworking, which inspired me a lot. I am also grateful to Emad for his friendship and for all fruitful discussions with him.

All the works I have done in the five years have been supported by funding from Intel Corporation. I thank Intel for the valuable support.

Furthermore, I would like to thank my wife Lindsay Yan. She always stood by my side when I got lost during my research and cheered me up. She is the love of my life and I would not be what I am without her.

Finally, I owe my deepest gratitude to my parents for their unconditional love for me and continuous encouragements throughout my entire study. Their supports are beyond any description in words. My accomplishments would not have been possible without them. Thank you.

CURRICULUM VITAE

Zhongqi Cheng

EDUCATION

Doctor of Philosophy in Computer Engineering **2020**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2017**
University of California, Irvine *Irvine, California*

Bachelor of Science in Microelectronics **2015**
Shanghai Jiao Tong University *Shanghai, China*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2020**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Grader for Advanced C Programming **2016**
University of California, Irvine *Irvine, California*

Teaching Assistant for Embedded Systems Modeling and Design **2017**
University of California, Irvine *Irvine, California*

Teaching Assistant for Embedded Systems Modeling and Design **2018**
University of California, Irvine *Irvine, California*

Teaching Assistant for Embedded Systems Modeling and Design **2019**
University of California, Irvine *Irvine, California*

Teaching Assistant for Embedded Software **2020**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Zhongqi Cheng, Rainer Dömer. **Analyzing Variable Entanglement for Parallel Simulation of SystemC TLM-2.0 Models**. ACM Transactions on Embedded Computing Systems 18, 5s, Article 79 (October 2019).

REFEREED BOOK CHAPTERS

Zhongqi Cheng, Tim Schmidt, Rainer Dömer. **SystemC Coding Guideline for Faster Out-of-Order Parallel Discrete Event Simulation**. Chapter 6 in "Languages, Design Methods, and Tools for Electronic System Design" by T. Kazmierski, S. Steinhorst and D. Grosse, reprint of best papers at FDL 2018, Springer Nature, Switzerland, January 2020. (ISBN 978-3-030-31585-6)

REFEREED CONFERENCE PUBLICATIONS

Daniel Mendoza, Zhongqi Cheng, Emad Arasteh, Rainer Dömer. **Lazy Event Prediction using Defining Trees and Schedule Bypass for Out-of-Order PDES**. Proceedings of the Design, Automation and Test in Europe (DATE) Conference, Grenoble, France, March 2020.

Zhongqi Cheng, Emad Arasteh, Rainer Dömer. **Event Delivery using Prediction for Faster Parallel SystemC Simulation**. Proceedings of the Asia and South Pacific Design Automation Conference, Beijing, China, January 2020.

Daniel Mendoza, Ajit Dingankar, Zhongqi Cheng, Rainer Dömer. **Integrating Parallel SystemC Simulation into Simics(R) Virtual Platform"**. Proceedings of the Design and Verification Conference in Europe, Munich, Germany, October 2019.

Zhongqi Cheng, Tim Schmidt, Rainer Dömer. **Enabling IP Reuse and Protection in Out-of-Order Parallel SystemC Simulation**. Proceedings of the International Embedded Systems Symposium, Springer, Friedrichshafen, Germany, September 2019.

Zhongqi Cheng, Tim Schmidt, Rainer Dömer. **SystemC Coding Guideline for Faster Out-of-Order Parallel Discrete Event Simulation**. Proceedings of Forum on Specification and Design Languages, Munich, Germany, September 2018.

Tim Schmidt, Zhongqi Cheng, Rainer Dömer. **Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation**. Proceedings of Design, Automation and Test in Europe, Dresden, Germany, March 2018.

Zhongqi Cheng, Tim Schmidt, Rainer Dömer. **Thread- and Data-Level Parallel Simulation in SystemC, a Bitcoin Miner Case Study**. Proceedings of the International High Level Design

Validation and Test Workshop 2017, Santa Cruz, California, October 2017.

MASTER THESIS

Zhongqi Cheng. **Design and Evaluation of a Bitcoin Miner SystemC Model with Thread and Data-Level Parallelism**. Master's thesis, University of California, Irvine, California, June 2017.

TECHNICAL REPORTS

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza, Rainer Dömer. **RISC Compiler and Simulator, Release V0.6.0: Out-of-Order Parallel Simulatable SystemC Subset**. Center for Embedded and Cyber-Physical Systems, Technical Report 19-04, September 2019.

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza, Rainer Dömer. **RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset**. Center for Embedded and Cyber-Physical Systems, Technical Report 18-03, September 2018.

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Rainer Dömer. **RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset**. Center for Embedded and Cyber-Physical Systems, Technical Report 17-05, July 2017.

OPEN SOURCE SOFTWARE RELEASES

Docker image for RISC Compiler and Simulator, Release V0.6.0. `https://hub.docker.com/r/ucirvinelecs/risc060/`, September, 2019.

RISC Compiler and Simulator, Release V0.6.0. `http://www.cecs.uci.edu/~doemer/risc.html#RISC060`, September, 2019.

RISC Compiler and Simulator, Release V0.5.0. `http://www.cecs.uci.edu/~doemer/risc.html#RISC050`, September, 2018.

RISC Compiler and Simulator, Release V0.4.2. `http://www.cecs.uci.edu/~doemer/risc.html#RISC042`, June, 2018.

RISC Compiler and Simulator, Release V0.4.0. `http://www.cecs.uci.edu/~doemer/risc.html#RISC040`, July, 2017.

ABSTRACT OF THE DISSERTATION

Advancing Compiler and Simulator Techniques for Highly Parallel Simulation of Embedded Systems

By

Zhongqi Cheng

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2020

Professor Rainer Dömer, Chair

As an Electronic System Level (ESL) design language, the IEEE SystemC standard is widely used for testing, validation and verification of embedded system models. Discrete Event Simulation (DES) has been used for decades as the default SystemC simulation semantic. However, due to the sequential nature of DES, Parallel DES has recently gained an increasing amount of attention for performing high speed simulations on parallel computing platforms. To further exploit the parallel computation power of modern multi- and many-core platforms, Out-of-order Parallel Discrete Event Simulation (OoO PDES) has been proposed. In OoO PDES, threads comply with a partial order such that different simulation threads may run in different time cycles to increase the parallelism of execution. The Recoding Infrastructure for SystemC (RISC) has been introduced as a tool flow to fully support OoO PDES.

To preserve the SystemC semantics under OoO PDES, a compiler based approach statically analyzes the race conditions in the input model. However, there are severe restrictions: the source code for the input design must be available in one file, which does not scale. This disables the use of Intellectual Property (IP) and hierarchical file structures. In this dissertation, we propose a partial-graph based approach to scale the static analysis to support separate files and IP reuse. Specifically, the Partial Segment Graph (PSG) data structure is proposed and is used to abstract the behaviours and communication of modules within a single translation unit. These partial graphs

are combined at top level to reconstruct the complete behaviors and communication of the entire model.

We also propose new algorithms to support the static analysis for modern SystemC TLM-2.0 standard. SystemC TLM-2.0 is widely used in industrial ESL designs for better interoperability and higher simulation speed. However, it is identified as an obstacle for parallel SystemC simulation due to the disappearance of channels. To solve the problem, we propose a compile time approach to statically analyze potential conflicts among threads in SystemC TLM-2.0 loosely- and approximately-timed models. A new Socket Call Path (SCP) technique is introduced which provides the compiler with socket binding information for precise static analysis. Based on SCP, an algorithm is proposed to analyze entangled variable pairs for automatic and accurate conflict analysis.

Besides the works on the compiler side, we focus as well on increasing the simulation speed of OoO PDES. We observe that the granularity of the Segment Graph (SG) data structure used in static analysis has a high impact on OoO PDES. This motivates us to propose a set of coding guidelines for the RISC users to properly refine their SystemC model for a higher simulation speed.

Furthermore, in this dissertation, an algorithm is proposed to optimize directly the event delivery strategy in OoO PDES. Event delivery in OoO PDES was very conservative, which often postponed the execution of waiting threads due to unknown future behaviors of the SystemC model, and in turn became a bottleneck of simulation speed. The algorithm we propose takes advantage of the prediction of future thread behaviors, and therefore allows waiting threads to resume execution earlier, resulting in significantly increased simulation speed.

To summarize, the contributions of this dissertation include: 1) a scalable RISC tool flow for statically analyzing and protecting 3rd party IPs in models with multiple files, 2) an advanced static analysis approach for modern SystemC TLM-2.0 models, 3) a set of coding guidelines for RISC users to achieve higher simulation speed, and 4) a more efficient event delivery algorithm in OoO PDES scheduler using prediction information.

Together, these compiler and simulator advances enable OoO PDES for larger and modern model

simulation and thus improve the design of embedded systems significantly, leading to better devices at lower cost in the end.

Chapter 1

Introduction

In recent years, embedded systems become ubiquitous and essential. They are included in almost every kind of devices, e.g. cell-phones [1, 2, 3, 4], automobiles [5, 6, 7, 8], medical instruments [9, 10, 11, 12], missiles [13, 14, 15, 16] and so on. However, as people are demanding faster and more functional electronic devices, this leads to a rapid growing of the complexity and heterogeneity of embedded systems [17, 18, 19]. It becomes much more difficult and complicated for developers to consider all aspects of the entire design. To solve this problem, Electronic System Level (ESL) design methodology was proposed and widely studied [20, 21, 22, 23]. Developers first model and evaluate a design at a high abstraction level with a main focus on functionalities and algorithms. Before converting the system level design into Register-Transfer Level (RTL) designs, the developers have to carefully verify the correctness of the model. One common verification approach is Simulation-based Validation [24, 25, 26], which simulates the given model and produces an accurate result.

IEEE SystemC [27] is a standard ESL design language and is widely used for testing, validation and verification of system level models. To utilize the parallel computation power of modern multi-core processors in SystemC simulation, parallel simulation approaches attract a lot of attention in the community. In this dissertation, we aim at designing and improving more advanced algorithms for

parallel simulation of modern SystemC models.

1.1 Parallel Simulation of Embedded System Models

ESL design is a level above RTL design including both software and hardware of the whole system. Figure 1.1 illustrates the complexities of different levels of abstraction in a complete design flow [28]. It clearly illustrates the trade-off between complexity and model accuracy for different abstraction levels. ESL design is a promising approach at an early stage of the design flow.

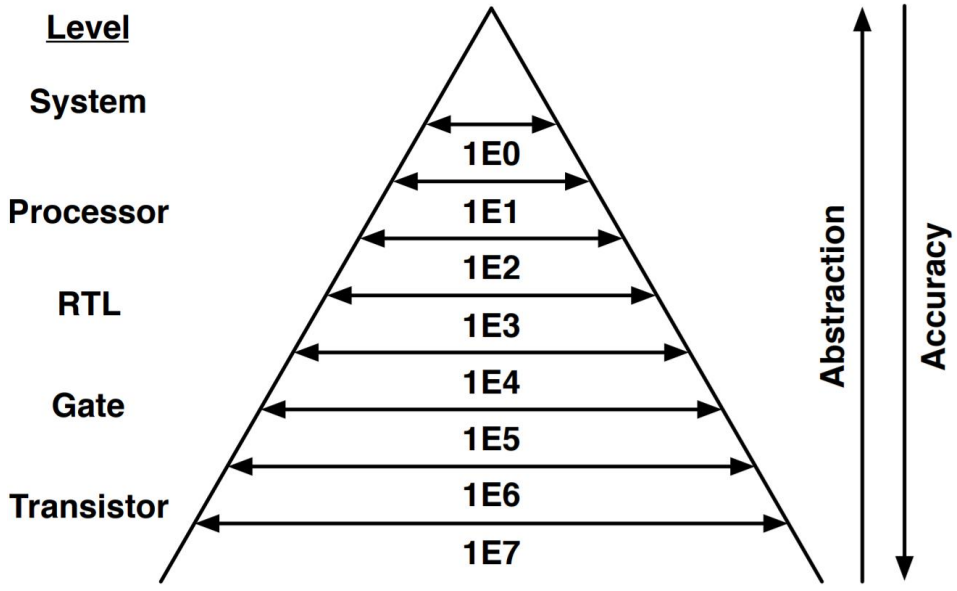


Figure 1.1: Level of abstraction in system design [28]

SystemC [27] is a de-facto language for ESL design in industry. With more than 20 years' development, SystemC has been widely applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis. SystemC is often associated with ESL design and with transaction-level modeling (TLM). Technically, SystemC is a set of C++ classes and macros. Developers can easily write SystemC models following the C++ syntax. Although SystemC is capable for RTL modeling and simula-

tion, this dissertation mainly focus on modeling and simulation for un-timed, loosely-timed and approximately-timed TLM and TLM-2.0 [29] SystemC models.
SystemC provides an event driven simulation interface which enables the simulation of concurrent SystemC processes. In the next sections, we are going to describe different approaches of Discrete Event Simulation (DES) algorithms.

1.1.1 Discrete Event Simulation

DES [30] is the inherent simulation approach for the SystemC language [31]. It utilizes a central scheduler to manage multiple concurrent threads, which results in temporal barriers (namely time and delta cycle) in the SystemC simulation. According to the cooperative multitasking semantics of the SystemC standard IEEE 1666-2011 [27], most SystemC simulator implementations have only one thread active at the same time and thus cannot utilize the parallel computing resources available on multi-core (or many-core) processor hosts. This significantly limits the execution speed of SystemC simulation. Figure 1.2 shows the algorithm for the traditional discrete event simulator [32].

Here, we formally define the data structures, states and operations used in DES:
1) In DES, we have the following thread queues:

1. **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITTIME**, **COMPLETE**}.
2. **READY** = $\cup th$ where thread th is ready to run.
3. **RUN** = $\cup th$ where thread th is running.
4. **WAIT** = $\cup th$ where thread th is waiting for an event.
5. **WAITTIME** = $\cup th$ where thread th is waiting for time advance.
6. **COMPLETE** = $\cup th$ where thread th has completed its execution.

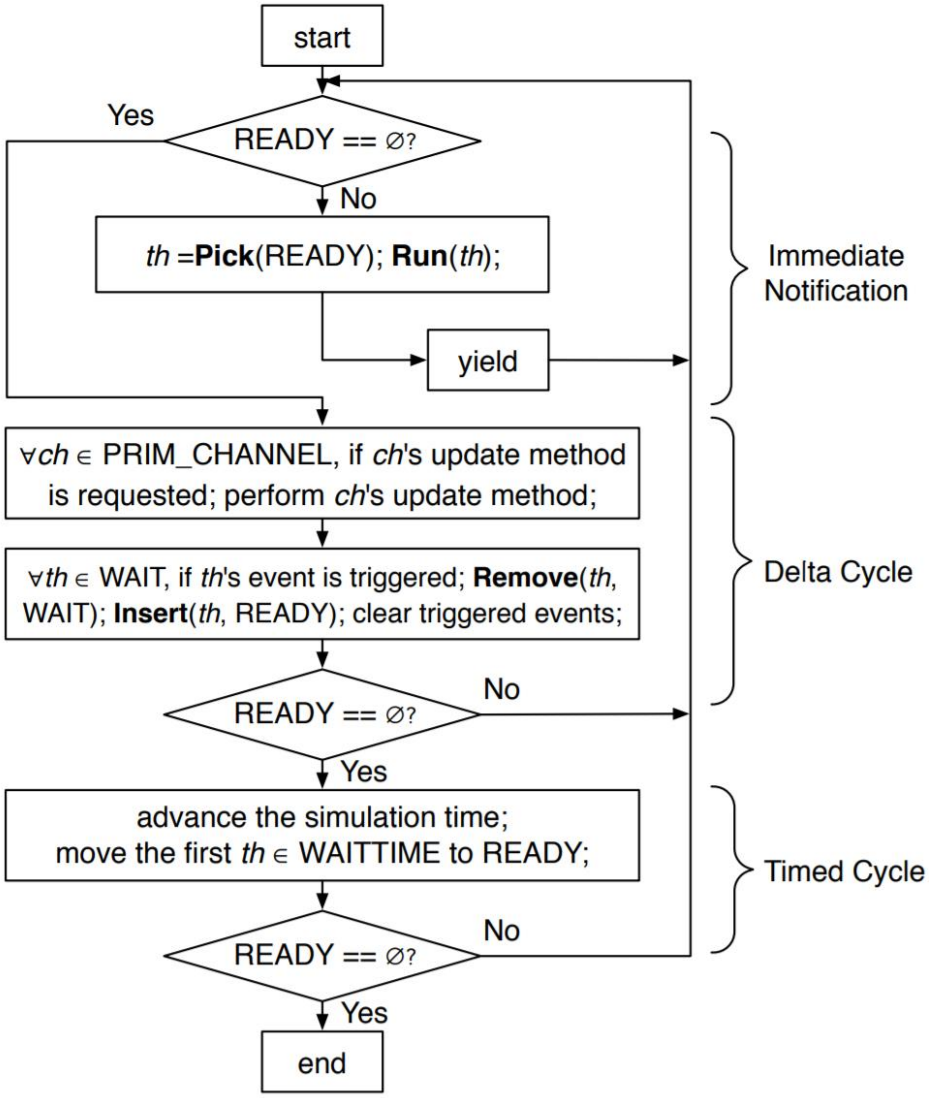


Figure 1.2: Traditional Discrete Event Simulation (DES) scheduler for SystemC[32]

2) In DES, we have the following simulation invariants:

1. **THREADS** = **READY** ∪ **RUN** ∪ **WAIT** ∪ **WAITTIME** ∪ **COMPLETE**.
2. $\forall A, B \in$ **QUEUES** and $A \neq B$, $A \cap B = \emptyset$.

3) In DES, we have the following operations:

1. RUN(th): Dispatch thread th.

2. th = PICK(**READY**): Pick a thread from the **READY** queue.

3. REMOVE(th, **WAIT**): Remove thread th from the **WAIT** queue.

4. INSERT(th, **READY**): Insert thread th to the **READY** queue.

4) In DES, we have the following initial states:

1. **THREADS** = $\{th_{root}\}$.

2. **RUN** = $\{th_{root}\}$.

3. **READY** = **WAIT** = **WAITTIME** = **COMPLETE** = $\emptyset$.

In DES, the simulated time contains two parts: the time cycle and the delta cycle. The time cycle represents the actual time advance during the simulation. The delta cycle is interpreted as the zero-delay semantics in digital systems.

The simulation is driven by events and time advances. At any time, the simulator runs a single thread from the **READY** queue. Once the **READY** queue is empty, the simulator checks the **WAIT** queue and move the threads that wake up to the **READY** queue. If the **READY** queue is still empty after event delivery, time will be advanced and corresponding threads in the **WAITTIME** queue are moved to the **READY** queue. If the **READY** queue is still empty, then the simulation reaches the end.

1.1.2 Synchronous Parallel Discrete Event Simulation

In order to provide faster simulation and due to the inexpensive availability of parallel processing on today's multi-core (and many-core) processors, Parallel Discrete Event Simulation (PDES) has recently gained significant attention [33]. The synchronous PDES simulator issues multiple threads (i.e. SC_METHOD, SC_THREAD and SC_CTHREAD) at the same time and dispatches them

onto the available cores in parallel. Specifically, once the **READY** queue is not empty and there are still cores idle, threads will be dispatched from the **READY** queue to run on the idle cores, until no more threads remaining in the **READY** queue or all cores are used. In turn, the simulation speed increases significantly.

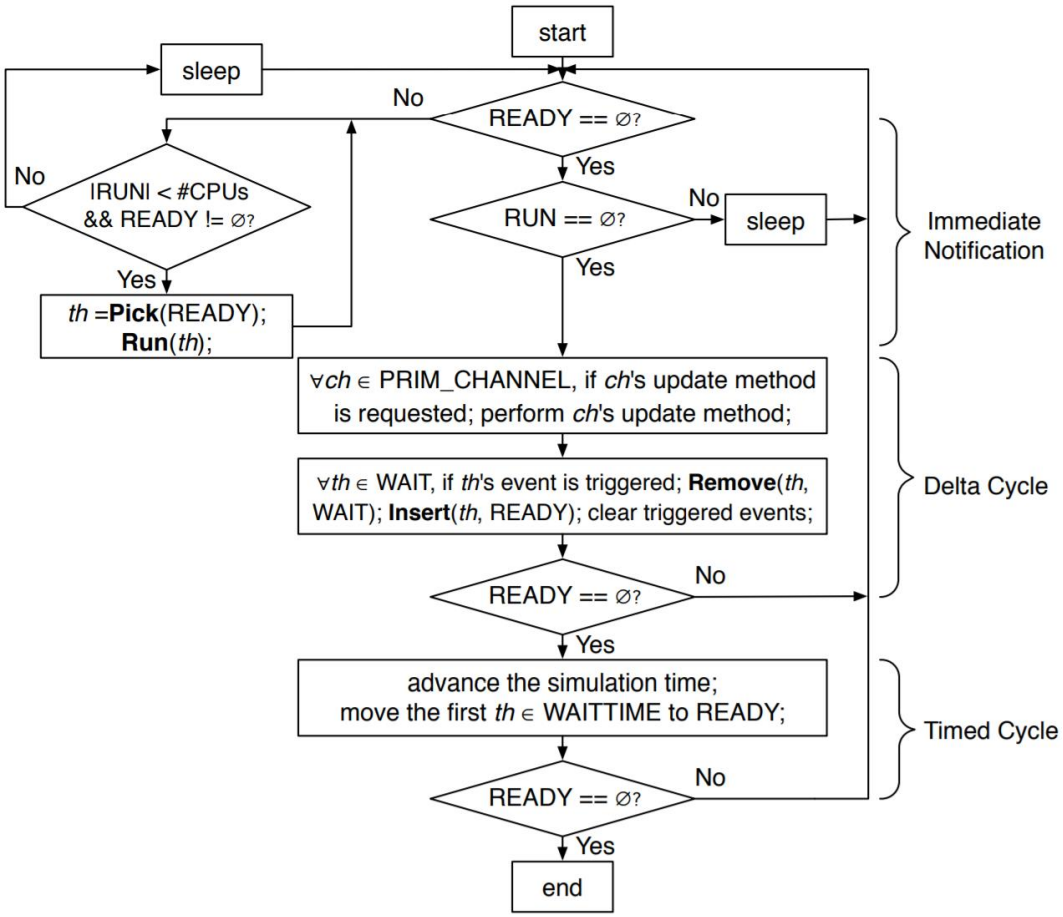


Figure 1.3: Synchronous Parallel Discrete Event Simulation (PDES) scheduler for SystemC [33]

The scheduler in the parallel simulator works in a similar way as the sequential simulator. There is still a main loop for handling event notifications. The main difference is that the parallel simulator picks multiple threads from the **READY** queue in each cycle, and runs them in parallel. The algorithm is shown in Figure 1.3. Note that in synchronous PDES, time advances happen globally. That is, earlier completed threads have to wait until all other running threads reach the same simulation

cycle, even if the threads do not have any conflict with each other in the future. The strict total order of time imposed by the synchronous PDES is still a limit to high performance parallel simulation.

1.1.3 Out-of-Order Parallel Discrete Event Simulation

To further utilize the parallel computation power of multi-core processors, Out-of-Order Parallel Discrete Event Simulation (OoO PDES) was proposed [34]. In OoO PDES, the simulation time is local to individual threads and events. It is formally defined as follows:

1. A thread th is assigned a local timestamp (t_{th},δ_{th}).
2. An event e is assigned a local timestamp (t_e,δ_e).

Timestamps have the following orders:

1. (t_1,δ_1) = (t_2,δ_2) iff $t_1 = t_2$ and $\delta_1 = \delta_2$.
2. (t_1,δ_1) < (t_2,δ_2) iff $t_1 < t_2$, or $t_1 = t_2$ and $\delta_1 < \delta_2$.

In OoO PDES, thread queues are separated into ones that correspond to different timestamps:

1. **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITTIME**, **COMPLETE**}.
2. **READY** = $\cup$**READY**$_{t,\delta}$, **READY**$_{t,\delta}$ = $\cup th$ where thread th is ready to run at (t, δ).
3. **RUN** = $\cup$**RUN**$_{t,\delta}$, **RUN**$_{t,\delta}$ = $\cup th$ where thread th is running at (t, δ).
4. **WAIT** = $\cup$**WAIT**$_{t,\delta}$, **WAIT**$_{t,\delta}$ = $\cup th$ where thread th is waiting for an event since (t, δ).
5. **WAITTIME** = $\cup$**WAITTIME**$_{t,\delta}$, **WAITTIME**$_{t,\delta}$ = $\cup th$ where thread th is waiting for a time advance at (t, δ).

6. **COMPLETE** = $\cup$**COMPLETE**$_{t,\delta}$, **COMPLETE**$_{t,\delta}$ = $\cup th$ where thread th has completed its execution at (t, δ).

In OoO PDES, initial states are refined as follows:

1. **THREADS** = $\{th_{root}\}$ where $(t_{root}, \delta_{root})$ = (0,0).
2. **RUN** = **RUN**$_{0,0}$ = $\{th_{root}\}$.
3. **READY** = **READY**$_{0,0}$ = **WAIT** = **WAIT**$_{0,0}$ = **WAITTIME** = **WAITTIME**$_{0,0}$ = **COMPLETE** = **COMPLETE**$_{0,0}$ = $\emptyset$.

Figure 1.4 depicts the algorithm of OoO PEDS. In OoO PDES, no global time order is imposed to the simulation threads. The scheduler aggressively moves any thread in **WAIT** that is notified by an event or any thread in **WAITTIME** to the **READY** queue. Note the NoConflicts(th) condition shown in Figure 1.4. Detailed dependency analysis is needed to avoid data or event conflicts for any shared variables among the parallel threads. Only if NoConflicts(th) is true, a new thread is issued for parallel execution (moved from the **READY** to the **RUN** queue). With the partial order of timing and advanced conflict analysis, the system model can be simulated without loss of accuracy with higher simulation speed.

We will be using advanced static compile-time analysis (and optionally dynamic run-time analysis) to identify all 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.

1.2 Recoding Infrastructure for SystemC

The Recoding Infrastructure for SystemC (RISC) [35] is essential to realize the OoO PDES approach. In this section, we briefly describe the basic concepts and notions in RISC.

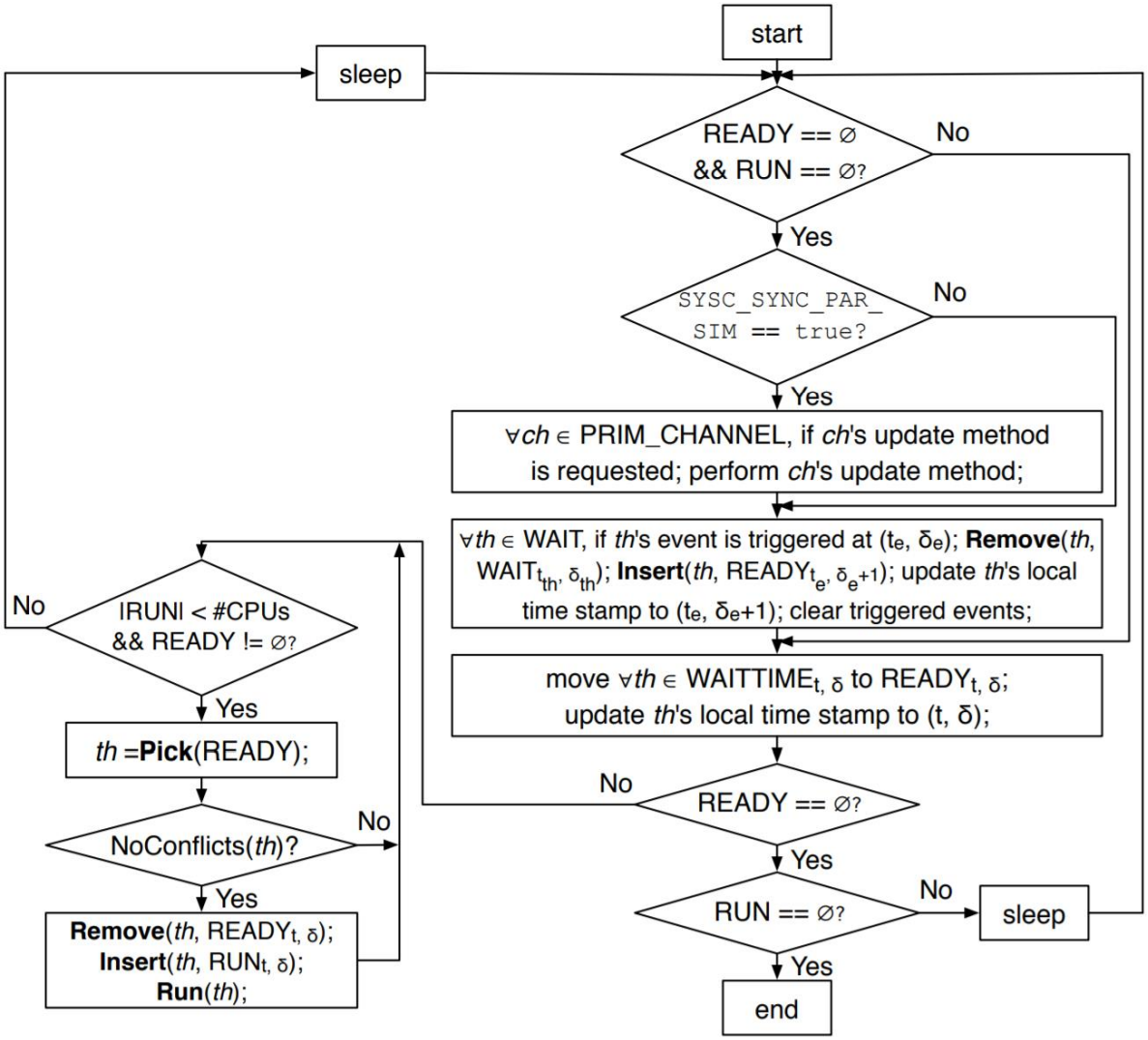


Figure 1.4: Out-of-Order Parallel Discrete Event Simulation (OoO PDES) scheduler for SystemC [34]

As described in Section 1.1.3, a dedicated SystemC compiler is implemented to provide essential information for the OoO PDES simulation library. Figure 1.5 shows the design flow using the RISC compiler and simulator [36]. As shown in this figure, the input SystemC model file is first sent to the RISC compiler, and the front-end RISC compiler generates an instrumented intermediate model. Then this model is linked against the parallel RISC SystemC library by the target compiler (a regular C++ compiler) to produce the final executable file. This is different from the conventional SystemC simulation where a regular C++ compiler includes the SystemC headers and links the input model directly against the SystemC library.

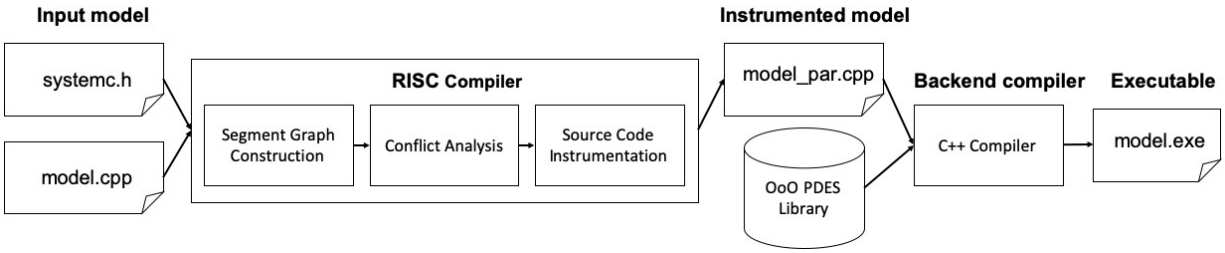


Figure 1.5: RISC Compiler and Simulator for Out-of-Order PDES of SystemC [35]

In RISC compiler four major tasks are performed. The compiler first constructs the Abstract Syntax Tree (AST) and builds the Segment Graph (SG) which describes the behaviors of SystemC processes. Next, it builds the Internal Representation (IR) of the SystemC model. Then, based on IR and SG, the compiler generates the conflict tables representing the data race conditions, segment dependencies and other information about the model. Finally, an intermediate model is created, containing all the conflict tables and other essential information for the OoO PDES library.

1.2.1 Segment Graph

RISC compiler relies on a complex software stack as its foundation, as shown in Figure 1.6. On top of the AST constructed using C/C++ standard libraries and ROSE infrastructure [37], RISC compiler generates the SG data structure.

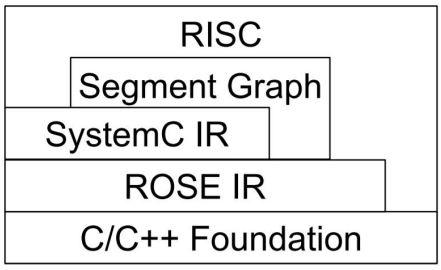
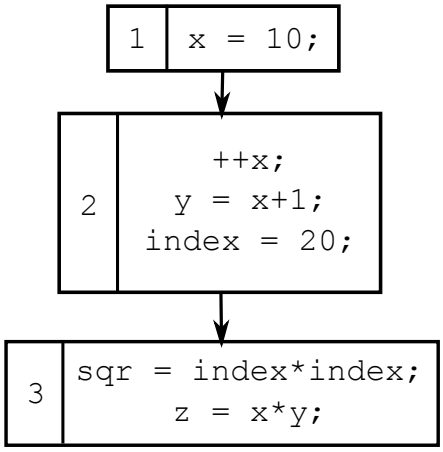


Figure 1.6: RISC software stack [35]

The SG is a directed graph where each node is a set of code statements executed between two scheduling steps [34]. A scheduling step is the entry to the scheduler domain from the application domain during execution of the model, which includes `wait` statement, start of a SystemC process and end of a SystemC process. An example of SystemC source code is shown in Figure 1.7a. The corresponding SG is shown in Figure 1.7b.

```
1  void foo(){
2      x = 10;
3      wait(1,SC_NS);
4      ++x;
5      y = x+1;
6      bar();
7      z = x*y;
8  }
9
10 void bar(){
11     index = 20;
12     wait(2,SC_NS);
13     sqr = index*index;
14 }
```

(a) Source code



(b) Segment Graph

Figure 1.7: Example SystemC Code and corresponding SG

A SG is automatically built by the RISC compiler as discussed in Algorithm 5 in [38]. The algorithm analyzes every source code statement and groups them into segments. For instance, `++x` in line 4 and `y=x+1` in line 5 are assigned both to segment 2 as they are executed after the same scheduling step: `wait(1,SC_NS)` in line 3. Function call is a special case because it may contain multiple statements and/or scheduling steps. When a function call is encountered, the algorithm first finds the definition of the function and constructs a temporary SG *sg_func* for the function. Then, *sg_func* is merged into the calling segment. In this example, `bar()` is first encountered while building segment 2. The RISC compiler identifies the definition of `bar()` in the AST and builds a SG *sg_func* for `bar()`. Then, *sg_func* is merged into segment 2.

1.2.2 SystemC Internal Representation

SystemC IR is built based on AST generated by ROSE infrastructure. It reflects the SystemC module and channel hierarchy, connectivity, and other SystemC-specific relations, as depicted in Figure 1.8. This is similar to the SystemC-clang representation [39]. More details are described in [36].

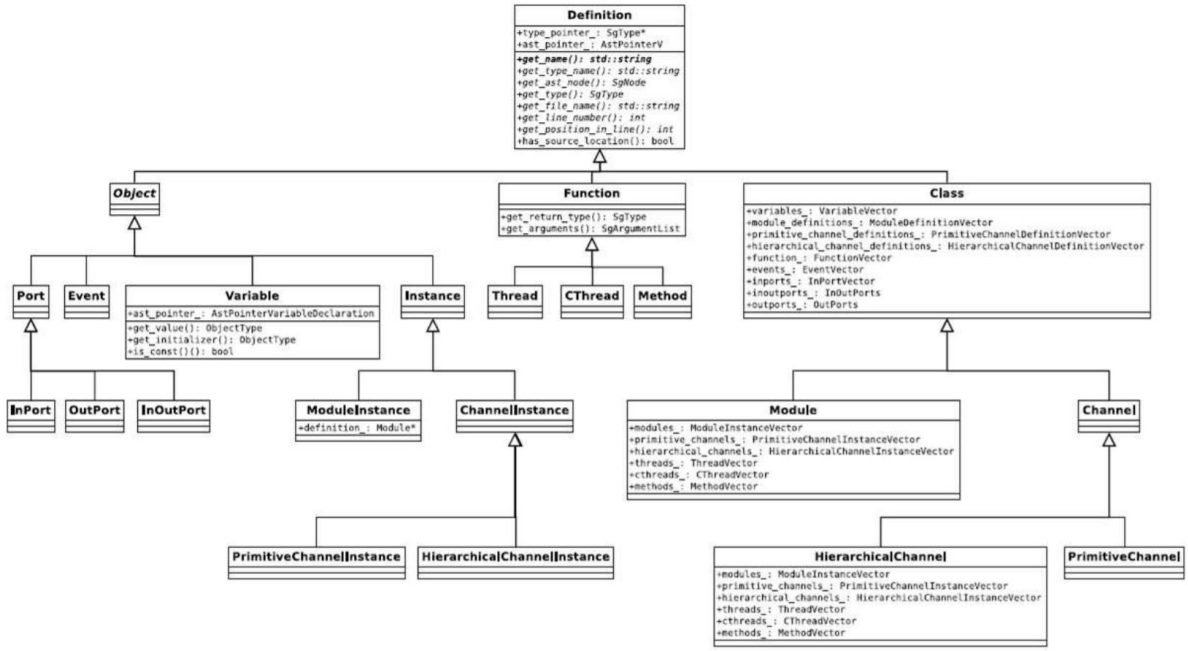


Figure 1.8: SystemC IR in RISC compiler

1.2.3 Conflict Analysis

Based on SG and SystemC IR, RISC compiler analyzes the conflicts at segment level. 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. The conflict information is used at runtime by the simulator to make sure that there is no thread dispatched if the thread has any potential conflicts against threads in the **RUN** or **READY** queue. The RISC compiler detects conflicts in two ways: static analysis at compile time and dynamic analysis in

the elaboration phase at run time. It should be emphasized that the accuracy of this analysis has significantly improved with the recent RISC release V0.6.0, which includes the advances presented in this work. As outlined in detail in [40], the RISC compiler now supports Port Call Path (PCP) sensitive conflict analysis which makes it aware of the actual channel instances used by threads from different modules. This much more precise analysis can avoid false positive conflicts in many cases and thus increases the efficiency of the simulation which, in turn, runs faster.

Static Conflict Analysis

Static analysis depends purely on the available information in the SystemC source code of the design model at input. In this case, the RISC compiler performs very conservative identification of the potential hazards in the model, as outlined in [36]. Identifying all possible hazards is a complex analysis task that requires the full understanding of the module hierarchy. Here we statically extract the module hierarchy and analyze the individual threads.

Dynamic Analysis

However, in most cases not all of the needed information can be gathered statically. For instance, design parameters may be passed via the command line to define the number of modules, certain channel characteristics, or other configuration information. In such SystemC models, the instantiated modules, channels, and ports are typically created through loops in a dynamic fashion. Thus, these exact parameters are only available at run time, so they cannot be statically analyzed. In these cases, dynamic analysis is needed.

In dynamic analysis, the compilation flow is extended by a preprocessing step. The input SystemC model is fed into the RISC elaborater which produces an executable model that only performs the SystemC elaboration phase. At the end of the elaboration phase, the executable model automatically traverses the created module hierarchy via the SystemC introspection API and dumps this detailed

structural design information into an instance connectivity file. This file is in turn provided as an input to the RISC compiler, so that the dynamically created design hierarchy and specific instance connectivity can be used for precise conflict analysis. The instance connectivity data file includes the actual module hierarchy, the specific port mapping, and the actual target variable mapping of references.

1.2.4 Source Code Instrumentation

As a result of the conflict analysis (static, dynamic, or hybrid [36]), the RISC compiler generates several conflict tables that describe all possible conflicts between threads in any two segments. 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 any given thread in parallel or ahead of time.

As shown above in Figure 1.5, 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 model instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) source code which the simulator can then rely on. At the same time, the RISC compiler also instruments user-defined SystemC channels with automatic protection against race conditions among communicating threads.

In total, the RISC source code instrumentation includes four major components:

- Segment and instance IDs: Individual threads are uniquely identified by a creator instance ID and their current code location (segment ID). Both IDs are passed into the simulator kernel as additional arguments to scheduler entry functions, including wait and thread creation.

- Data and event conflict tables: Segment concurrency hazards due to potential data conflicts,

event conflicts, or timing conflicts are provided to the simulator as two-dimensional tables indexed by a segment ID and instance ID pair. For efficiency, these table entries are filtered for scope, instance path, and reference and port mappings.

- Current and next time advance tables, and thread state prediction tables: The simulator can make better scheduling decisions by looking ahead in time if it can predict the possible future thread states. This optimization is discussed in detail in [41] and is available in the RISC Compiler and Simulator in versions 0.4.0 and later. Since thread state prediction for most models requires only little additional compile time but results often in higher simulation speed, it is enabled by default.

Note that the source code instrumentation is performed automatically by the RISC Compiler and no user interaction is necessary. However, the interested user may inspect the instrumented source code. It is stored in a file named risc_model_name.cpp which serves as the input file to the compiler back-end which in turn then generates the final executable.
With RISC version 0.6.0, source code instrumentation is optimized for large design models with many segments. Here, the conflict, time, and prediction tables can become fairly large, which unnecessarily slows down the code generation step during compilation. To avoid such inefficiency, a separate file (model_name.risc) is automatically generated with binary images of the tables. This file is then read at run time (automatically, just like a shared library) to fill the conflict, time, and prediction tables needed by the simulator.

1.2.5 Simulation Library

Same as the classic Accellera proof-of-concept implementation, the RISC simulator is not an explicit tool, but a run-time library that the generated executable SystemC model is linked against. Thus, simulation is performed by execution of the compiled model, the same way as in the classic tool flow (just faster).

By default, the simulation library performs OoO PDES scheduling. However, when primary channels are detected in the model, or specified by the user, the RISC simulation library can fall back to Synchronous PDES or even DES.

1.3 Related Work

DES is the inherent approach for SystemC simulation. However, due to its sequential nature, PDES is studied to take advantage of the parallel computing capabilities on multi-core and many-core hosts. The main idea of PDES is to execute a simulation program on a parallel computer by decomposing the simulation application into a set of concurrently executing processes [33]. In the past few decades, PDES has attracted tremendous attention and interest and grown fast.

In [39] SystemC-clang is proposed. It analyzes SystemC models with a mixture of transaction-level and register-transfer level components. The authors in [42] studied the distributed parallel simulation, where SystemC models are organized into small executable units and distributed onto different host machines to run in parallel. A parallel SystemC simulation kernel is proposed in [43] which requires the user to manually translate the sequential design into a safe parallel design. [44] takes a survey about existing SystemC simulation approaches and concludes that most of these works do not fully support the parallel simulation of TLM-2.0 LT models due to shared variables. [45] proposes a tool that addresses this problem. A set of primitives is provided to the user to manually express tasks with duration such that parallelism in the model can be exploited and LT models are executing in parallel. In [46], a parallel SystemC simulation kernel is developed by reducing synchronization overheads of parallel threads. [47] proposes an efficient multi-threaded memory allocator named HMalloc which eliminates false sharing and lock contention and supports lock-free shared memory allocation and deallocation. [48] identifies the available parallelism between event scheduling and execution, highlights points of contention between the two, provides an algorithm to take advantage of the parallelism. [49] exploits hardware profiling facilities to build a hardware-

supported incremental check-pointing solution that enables the reduction of the event-execution cost in speculative PDES compared to the software-based counterpart. [50] proposes a system for predicting future situations through parallel simulations based on observation data. An optimistic simulation engine is introduced to support parallel simulation of many possible situations and dynamic simulation modification based on observation data acquired by sensors. [51] presents an approach that exploits reinforcement learning techniques for speculative Time Warp-based PDES. Rather than assuming an optimal control strategy, the authors seek to find the optimal strategy through parameter exploration. A value function that captures the history of system feedback is used, and no a-priori knowledge of the system is required. [52] presents a runtime support for speculative parallel processing of discrete event simulation models on multi-core architectures. It exploits Hardware-Transactional-Memory (HTM) facilities for the purpose of state recoverability, which can hence host conventionally developed discrete event models relying on the concept of event-handlers to be dispatched by an underlying simulation engine. [53] optimizes PDES by implementing a thread-based version of the ROSS simulator. The multi-threaded implementation eliminates multiple message copying and significantly minimizes synchronization delays. [54] presents a PDES scheme that enables cost- and time-efficient execution of large scale parameter studies on GPUs. Two orthogonal levels of parallelism are exploited: external parallelism among the inherently independent simulations of a parameter study and internal parallelism among independent events within each individual simulation of a parameter study.

In comparison with the above approaches for PDES, the work we propose in this dissertation is based on a specific PDES approach: OoO PDES. To achieve accurate results, Out-of-Order PDES relies on a static analysis approach to collect the context information of each simulation thread. Note that SystemC models are written in C++ code, and C++ based static analysis has been widely studied. [55] proposes three algorithms for statically analyzing virtual functions in C++ to reduce compiled code size and program complexity. [56] introduces a tool called ITS4 for scanning vulnerability in C and C++ programs based on static analysis. [57] presents the Parallel Pattern Analyzer Tool which aids the discovery and annotation of parallel patterns in source codes. It

supports for identifying Map, Farm, and Pipeline parallel patterns and evaluate the quality of the detection for a set of different C++ applications. SystemC-specific static analysis has also been studied.

OoO PDES has been studied for years. In [34], OoO PDES is first introduced to increase the simulation speed over the synchronous PDES. [58] proposes a dynamic load-profiling and segment-aware scheduling algorithm with optimized thread dispatching to maximize parallel SystemC simulation speed, which generally can be applied to all work-sharing PDES approaches for better multi-core scheduling. [59] defines the concept of core distance for many-core architectures and proposes an approach to optimize thread-to-core mapping in order to minimize on-chip communication overhead. In [60], the authors achieve a high simulation speed of SystemC models by exploiting data-level parallelization together with thread-level parallelization. An algorithm was proposed to automatically apply data-level parallelization to the source code. This is orthogonal to the focus of this dissertation and thus could be applied here as well. Port Call Path [40] is an advanced technique that helps the compiler to gain more specific context information about non-pointer variables in channels. This can reduce false positive conflicts in channels and result in significantly increased simulation speed. [61] integrates OoO PDES library into the Simics virtual platform.

1.4 Goals

As outlined in Section 1.2, RISC provides a tool chain for highly parallel simulation of SystemC models. However, there are several noticeable restrictions that limit the use of RISC in large designs. In this section, we briefly introduce these limitations, and then set corresponding goals to overcome them. In other words, we scale the OoO PDES approach beyond academic examples to real-world industrial design models.

In industrial designs, a SystemC model often comes as separate files and involves heavy use

of third-party libraries and Intellectual Properties (IP). The reuse-based design methodology is actively applied to achieve high design productivity to meet the demand for shorter design cycles and relax the time-to-market pressure. However, RISC compiler needs the complete source code of a SystemC model to build AST for further static analysis. Complete source codes from the IP providers can hardly be obtained because IPs are carefully protected. This limits the use of RISC in SystemC models using third-party libraries and Intellectual Properties (IP).

Another limitation is that current static analysis does not support the modern SystemC TLM-2.0 standard. SystemC TLM-2.0 standard is widely used in industrial SystemC models. It exploits the benefits of both temporal decoupling and traditional TLM standard for higher simulation speed. In SystemC TLM-2.0, communication between SystemC processes is done by passing transactions, where each transaction consists of a generic payload object, a timing object and a transaction phase (for non-blocking transportation). The SystemC TLM ports and channels no longer exist in SystemC TLM-2.0, instead, an initiator passes the reference of a transaction to a target by calling a registered callback method on the target side. RISC compiler was not able to handle communications outside channels and cannot analyze references and pointers, therefore, SystemC TLM-2.0 models were not able to be statically analyzed by RISC compiler, which was a large limitation.

In this dissertation, we are going to scale the RISC compiler and simulator so that industrial SystemC designs can be simulated correctly and fast. Specifically, our goals include:

1. **Static analysis for 3rd party libraries and IPs**: For those SystemC models that use third-party libraries and IPs, we aim to enable OoO PDES to effectively perform static analysis to build Segment Graphs and analyze conflicts.

2. **Static analysis for SystemC TLM-2.0 standard**: For those SystemC models that use SystemC TLM-2.0 standard, we would like to allow our RISC compiler to efficiently recognize the communication between modules and statically analyze the conflicts.

3. **Improving simulation speed of OoO PDES**: We aim to further increase simulation speed of OoO PDES by direct and indirect optimizations in the simulator.

In the following chapters, we present our approaches to achieve our goals. Chapter 2 realizes the first goal. Chapter 3 realizes the second goal. The third goal is addressed in Chapter 4 and 5 respectively. Figure 1.9 summarizes our contributions with respect to the tool flow of RISC .

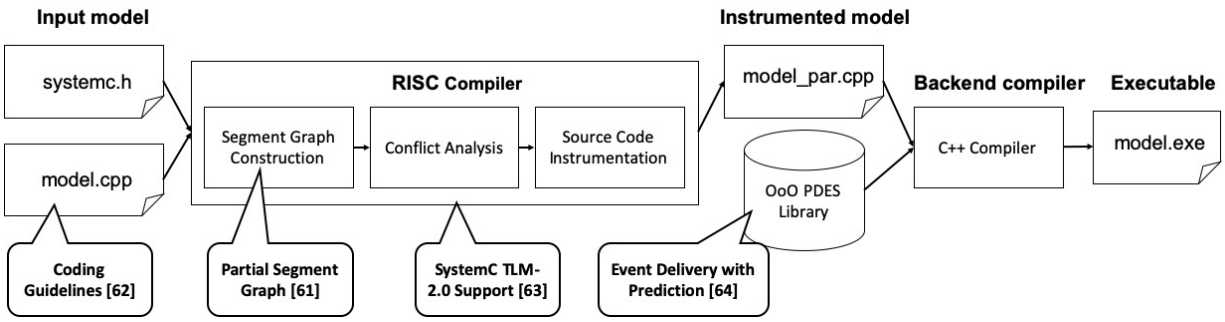


Figure 1.9: Contributions in RISC tool flow

The rest of the dissertation is organized as follows:

In Chapter 2, we extend the static analysis design flow to support separate files and IP reuse by introducing Partial Segment Graph (PSG) abstraction [62]. We also propose approaches to prevent IP security leakage by using PSG. Experiments demonstrate the effective design flow and sustained speedup with parallel simulation.

In Chapter 3, we propose a compile time approach to statically analyze potential conflicts among threads in SystemC TLM-2.0 loosely- and approximately-timed models [63]. We introduce a new Socket Call Path (SCP) technique which provides the compiler with socket binding information for precise static analysis. We also propose an algorithm to analyze entangled variable pairs. Experimental results show that our approach is able to support automatically safe parallel simulation of SystemC models with TLM-2.0 Blocking Transport Interface, Direct Memory Interface and Non-blocking Transport Interface, resulting in impressive simulation speeds.

In Chapter 4, we propose for RISC users coding guidelines that increase the granularity of segments [64], so that the level of parallelism in the design increases and higher simulation speed becomes possible. The experimental results show the simulation speed of OoO PDES increases significantly by applying the coding guidelines.

In Chapter 5, we introduce a novel event delivery strategy that allows waiting threads to resume execution earlier based on predicted behaviors of all threads, resulting in significantly increased simulation speed [65]. Experimental results show that the proposed approach increases the OoO PDES simulation speed by a large amount.

Finally, Chapter 6 summarizes this dissertation with an outline of contributions. Future works are also discussed.

Chapter 2

Extending Static Analysis for Large Models

In this chapter, we first propose an algorithm that builds Partial Segment Graph for each individual SystemC translation unit, we then propose an algorithm that combines all PSGs to reconstruct the complete Segment Graph [62]. Based on the two algorithms, the RISC compiler is able to support SystemC models with multiple source file inputs, especially for the models that use Intellectual Properties.

2.1 Introduction

As described in Chapter 1, the complexity of system design has been growing with the increasing functionalities of modern embedded systems. Out-of-Order Parallel Discrete Event Simulation (OoO PDES) [34] was proposed to exploit the parallel computation of modern multi- and many-core platforms, and the Recoding Infrastructure for SystemC (RISC) [35] has been developed to implement OoO PDES for SystemC. The RISC compiler first builds the Abstract Syntax Tree (AST) of the input file and then derives from the AST the *behavior model (BM)* of the input SystemC design. BM is an abstraction of the execution of the SystemC processes in the design. The RISC

compiler represents BM with a statically built Segment Graph (SG) data structure. Based on SG, the RISC compiler is able to analyze the data conflicts, timing conflicts and event hazards in the design.

2.2 Problem Definition

To completely build the BM of the input SystemC design, the RISC compiler needs the entire AST for the input model. Thus the user has to provide all the source code in one single translation unit. In other words, the RISC compiler cannot build BM for SystemC designs whose source code are separately structured in multiple source files or third party Intellectual Properties (IP). With the wide use of IP, this requirement severely restricts the RISC compiler to meet industrial system level design needs.

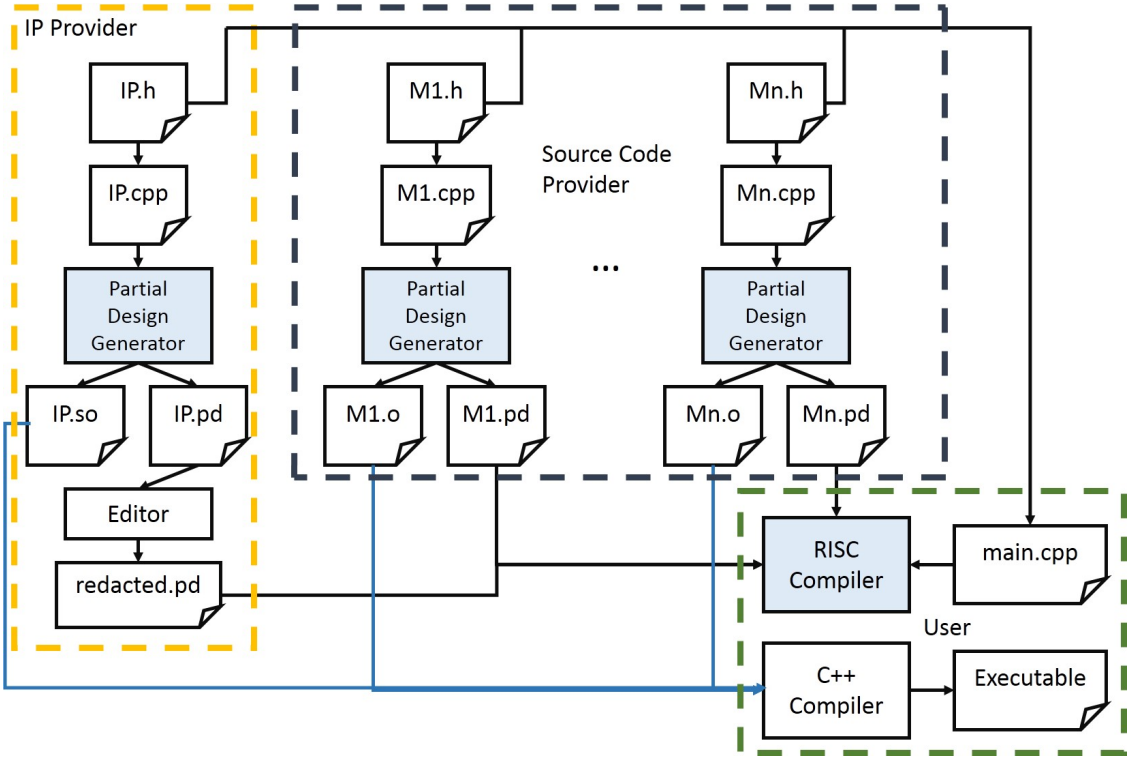


Figure 2.1: Scaled RISC tool flow with IP components

In this chapter [62], we propose a solution that scales the RISC compiler to support multiple file inputs, especially for the integration of IPs, as shown in Figure 2.1. In the new design flow, the construction of BM no longer relies on the complete AST. Besides the usual object and header files, component providers supply a *partial design (PD)* file that abstracts the BM of integrated design components. Specifically, in the PD file, the BM is abstracted by a *Partial Segment Graph (PSG)*. IP providers can inspect and redact the PD file, in order to further minimize the PSG, which protects the security of their IP. On the user's side, by combining all the received PSGs, the RISC compiler is able to reconstruct the BM of the whole design.

2.3 Related Work

IP reuse and protection have received a lot of attentions. In [66], the authors describe an effective methodology for IP reuse in SOC design. They studied the IP enhancement and also proposed a framework for the reuse of customer IP. [67] proposes a new preprocessing approach that embeds watermarks as constraints into the input of a black-box design tool and a new postprocessing approach that embeds watermarks as constraints into the output of a black-box design tool. [68] introduces a new technique for protecting the IP of both processor cores and application software in hardware/software systems. The approach is based on public-key cryptography and it has been implemented as a package in the JavaCAD distributed design and simulation environment.
In [69], the authors proposed a way to use pre-defined graphs to represent the BM of IP components. However, this simple approach requires the users to manually analyze the design and insert pragmas where needed. Furthermore, there are only three kinds of predefined graphs, which is insufficient. In contrast, we propose PSG as the data structure to represent the BM of IP components, which is accurate and is automatically built by a compiler.

2.4 Partial Segment Graph

We now describe the PSG technique that represents the BM in each separate translation unit.

2.4.1 Behavior Model and Segment Graph

The behavior model of a SystemC design can be described by the Segment Graph, which provides a way to analyze threads and their position during execution. The SG is a directed graph where each node is a sequence of code statements executed between two scheduling steps, i.e., `wait` statements [34]. Details about SG are described in Section 1.2.1

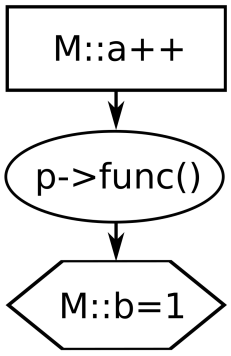
2.4.2 Concept of PSG

In our proposed design flow, we store the BM specified in each translation unit as a PSG in a PD file and when the PSGs are loaded and integrated together, they reconstruct the complete SG.

The main difference between PSG and SG is that PSG is built based on an incomplete AST, where

```
1  SC_MODULE(M){
2    ...
3    sc_port<C> p;
4    void th(){
5      M::a++;
6      p->func();
7      M::b=1;
8    }
9    ...
10 }
```

(a) Example Source Code



(b) PSG of Fig. 2.2a

Figure 2.2: SystemC Code and PSG

definitions of function calls may be unknown. An example is shown in Figure 2.2a. It contains only the definition and implementation of module `M`. Function `p->func()` is called in `M::th()`,

but it is not defined in this translation unit. We refer to a function call that lacks the definition as a *non-defining function call*. Because the compiler cannot determine from the current AST if a non-defining function call contains scheduling steps or not, the simulation cycle of the code statements following the non-defining function call cannot be statically determined. In the example, we cannot know if line 5 and line 7 execute in the same cycle.
To deal with this uncertainty incurred by the non-defining function calls, we introduce three types of PSG nodes:

- *Segment node* contains a sequence of code statements executed in the same determined simulation cycle. In Figure 2.2a, `M::a++` belongs to a segment node because its simulation cycle is determined, which is the first cycle of the `sc_thread M::th()`. A segment node becomes a segment after the integration of PSGs.

- *Partial segment node* contains a sequence of code statements executed in the same non-determined simulation cycle. In Figure 2.2a, `M::b=1` belongs to a partial segment node because it is executed after the non-defining function call `p->func()`. Later during the PSG integration phase, the partial segment node will be merged with other segment nodes.

- *Partial function call node* is created as a place holder for the non-defining function call in the PSG such that during the PSG integration phase, the partial function call node can be replaced by the sub-PSG corresponding to the function's definition. In Figure 2.2a, node 3 is a partial function call node for the non-defining function call `p->func()`.

2.4.3 Create PSG

A PSG is recursively built by traversing the AST of the current translation unit, as shown in Algorithm 1. If the current statement `CurrStmt` is a scheduling entry point (`wait` statement), then an empty segment node is created and connected to the nodes in the `CurrNodes`. On the

other hand, if `CurrStmt` is not a scheduling point, then it is added to all the nodes in `CurrNodes`. This is similar as in the *BuildSG* in [69]. The main difference is that to build PSG, the compiler also needs to deal with non-defining function calls. If `CurrStmt` contains a non-defining function call, for example `f()`, the compiler first builds a partial function call node `NewNode` and stores the qualified name `M::f()`. Next, the compiler connects `NewNode` to all the nodes in `CurrNodes`. Then, a partial segment node `NextNode` is created and connected to `NewNode`, and the compiler sets `NextNode` as the only node in the `CurrNodes`.

Algorithm 1 Partial Segment Graph Generation

```
 1: function BuildPSG(CurrStmt, CurrNodes)
 2:     if isBoundary(CurrStmt) then
 3:         NewNode ← new segmentNode
 4:         for Node ∈ CurrNodes do
 5:             AddEdge(Node, NewNode)
 6:         end for
 7:         return CurrNodes ∪ { NewNode }
 8:     else if isNonDefiningFunctionCallStmt(CurrStmt) then
 9:         NewNode ← new partialFuntionCallNode
10:         Mark(NewNode, getFuncName(CurrStmt))
11:         for Node ∈ CurrNodes do
12:             AddEdge(Node, NewNode)
13:         end for
14:         NextNode ← new partialSegmentNode
15:         AddEdge(NewNode, NextNode)
16:     else if isControlFlow(CurrStmt) then
17:         BuildSG(CurrStmt, CurrNodes)
18:         ...
19:     end if
20: end function
```

2.4.4 Store and Load PSG

The PD file stores an abstraction of the PSG. For each node, we omit the detailed code statements and store only the access types (R,W,RW) to non-local variables. This is sufficient for the RISC compiler to analyze the data and event conflicts. In addition, some meta-data is stored for each node, which is needed for the integration of PSGs, as listed in Table 2.1. Note that the PD file is compatible with dot format and the PSG therefore can easily be visualized. An example of PSG is

later shown in Figure 2.7.

A PSG is loaded from the PD file with a dot file parser. The parser reads the attributes of each PSG node, and reconstructs the data in memory. For example, a node has a variable access attribute `(W)M::a`, which indicates that `M::a` is been written in the node. To load the node into memory, the PSG parser locates the symbol of `M::a` in the AST and puts it into the *variable_write_list* of the node. PSG edges are constructed according to the connections specified in the PD file. After the loading of individual PSGs, the compiler integrates them together to construct the complete SG.

Table 2.1: PSG meta-data node attributes

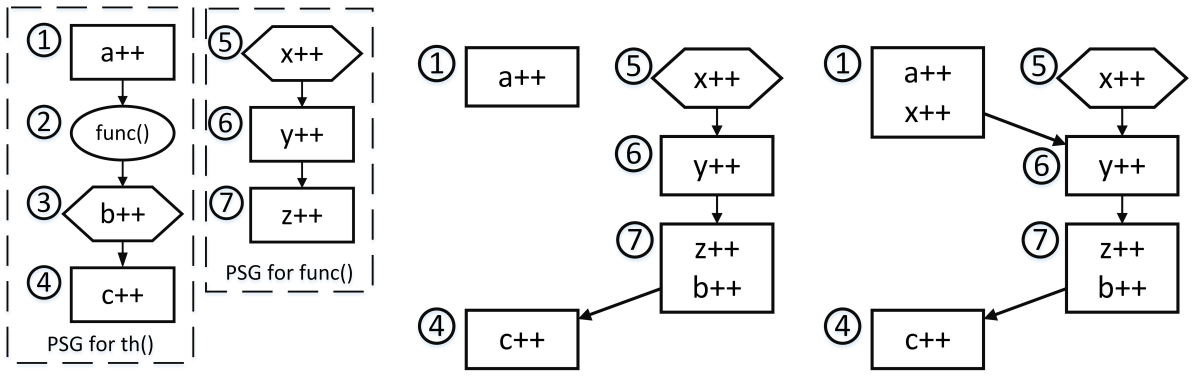
Attribute	Description
Node type	Segment node, Non-segment node, Function call node
Written variables	Qualified name of variables written
Read variables	Qualified name of variables read
Notified events	Qualified name of events notified
Dependent events	Qualified name of events waiting for
Hosting function name	Qualified name of function belonging to
Hosting module name	Qualified name of module belonging to
Is entry node	Marker for function entry point
Is exit node	Marker for function exit point
Is simulation process	Marker for simulation process
Non-defining function name	Qualified name of non-defining function

2.4.5 Integration Phase

A complete segment graph is the basis for accurate static analysis. After loading all the PSGs, first the partial function call nodes are recursively replaced with the corresponding sub-PSG. Second, all the partial segment nodes are merged with segment nodes they follow. All remaining nodes in the graph are segment nodes (with underlying `wait` boundaries) and belong to determined simulation

cycles, such that the integrated graph by definition becomes a proper segment graph. With the reconstructed SG, the RISC compiler has the complete the BM and can perform the needed static analysis of the design.

We illustrate the merging process of two PSGs in Figure 2.3a, 2.3b and 2.3c. In this example, *node_2* is a partial function call node that holds the non-defining function call `func()`, and *node_5*, *node_6* and *node_7* are loaded from the psg in func.pd and forms the sub-PSG of `func()`. *node_5* and *node_7* are respectively the entry and exit node of `func()`. First, *node_3* is merged into *node_7* because they belong to the same simulation cycle. After merging, *node_4* is connected to *node_7* since it was connected to *node_3*. Then, *node_5* is merged into *node_1* because it is the starting node of `func()`. *node_6* is connected to *node_1* since it was connected to *node_5*.



(a) Two PSGs (b) Phase 1 (c) Phase 2

Figure 2.3: Integration of PSGs

2.5 IP Protection and Security

IP reuse is an important feature in semiconductor industry. Basically, an IP consists of two parts: a header file that describes the interfaces and protocols, and a binary file that implements the IP component. Since no implementation source code is provided, the IP is protected. The internal BM of the IP is hidden from the users.

However, static analysis cannot be performed without the BM. We need the IP provider to supply an abstract PSG of the IP to the user via PD files. For the IP provider not to reveal too much implementation detail and to solve this IP security leakage problem, we allow the IP provider to redact the PSG in the PD file, so that the implementation details remain hidden. If desired, misleading information can even be added. This way the users will not be able to obtain the inner implementation, while still maintaining the correctness of BM.

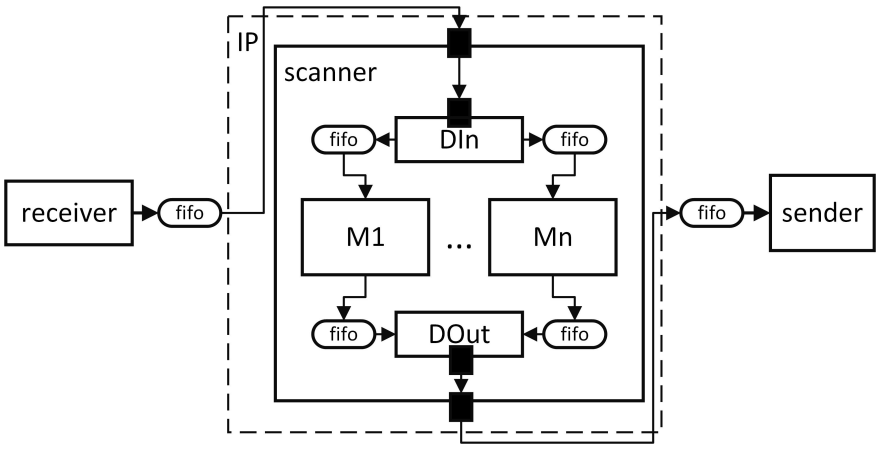
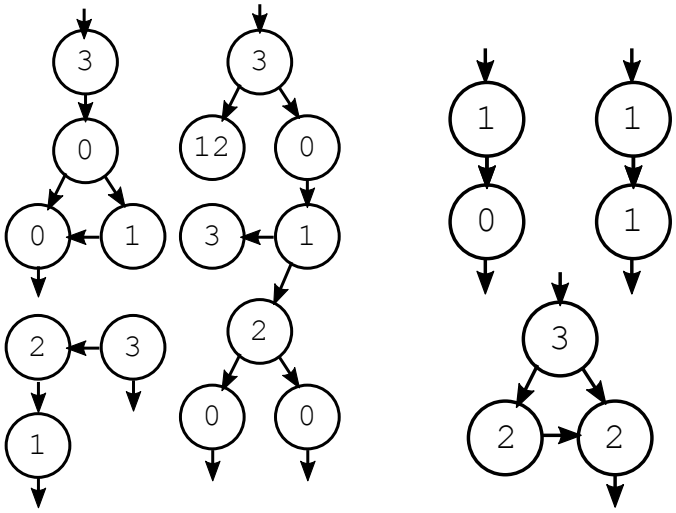


Figure 2.4: SystemC model of Bitcoin miner



(a) Original (b) Redacted

Figure 2.5: Original and redacted scanner.pd

Figure 2.4 shows the SystemC model of a Bitcoim miner [70]. It has several user defined modules

(receiver and sender) for data input and output, and uses an IP module (scanner) for number crunching. Three PD files (*receiver.pd*, *sender.pd*, *scanner.pd*) contain the corresponding BM of each module.

By default, in each PD file the RISC compiler stores (1) the qualified name of variables accessed and access types, (2) qualified name of events and dependencies, (3) PSG structure and timing advance. For *receiver.pd* and *sender.pd*, it is fine to have such information transparent because the two modules are user-defined. However, for *scanner.pd*, exposing the internals is risky from the perspective of IP protection.

The original and redacted versions of scanner.pd are shown in Figure 2.5. Here the numbers in the PSG nodes indicate the number of variable accesses stored. Compared to the original, the revised file has fewer nodes, and each node has fewer variable accesses stored. These modifications are carefully performed such that during the simulation, the model still executes correctly.

There are several possible changes that can be performed to redact the PSG:

1. Reduce the amount of variable accesses: If two nodes share more than one internal variable accesses, only one of them needs to be kept and others can be removed from the two nodes. This does not change the data conflict of the two segments. Only externally visible variables need to be retained. Furthermore, if a variable is read only in any node, it can be removed because it cannot lead to data conflicts.

2. Add fake variable access: By adding extra variables to a node, the IP provider can further obscure the IP and inject misleading details.

3. Hide nodes: An entire node that contains no variable accesses or event notifications can be hidden from PSG because it does not affect the static analysis. To maintain the timing correctness, the incoming and outgoing edges need to merge.

4. Merge segment nodes: Segment nodes can be merged to form an aggregation. This effectively hides the detailed PSG structure. The downside is that merging may pollute conflict-free

segment pairs. Two code statements that actually can run in parallel after merging run only sequentially because they now belong to two conflicting aggregated nodes.

In general, there is a trade-off between the amount of IP protection versus the analysis accuracy, which may affect simulation performance.

2.6 Experiments and Results

We first show the correctness of the proposed design flow using a simple producer-consumer example and a more complex Canny edge detector model. Then we demonstrate our IP protection using a SystemC model of Bitcoin miner, where we designed an IP for the parallel data crunching module and redacted the PD file to hide details in the PSG. Our experiments were executed on an Intel Xeon E3-1240 multicore processor with 8 CPU cores. The CPU frequency scaling was turned off so as to provide accurate and stable results.

2.6.1 Producer-Consumer Example

In the producer-consumer model, we have defined and implemented each module/channel in individual files. According to the tool flow in Figure 2.1, we first generate the PD file for each translation unit. At top level, RISC integrated the PSGs. The regular RISC tool flow without PD support cannot handle this multi-file design and generates an error message. Now with the proposed approach, RISC is able to correctly perform static analysis and generates a functioning parallel executable.

2.6.2 Canny Edge Filter

The Canny edge detector algorithm [71] is a multi-stage operator that detects edges in an image. Our SystemC model has a pipeline structure with five stages, and each stage communicates with the next via user-defined channels. We have defined and implemented all stage modules and channels in different translation units. In this experiment, we have a total of 15 implementation files and corresponding PD files. The sequential simulation runs for 280.90 seconds, and the parallel one runs for 116.48 seconds, achieving a speedup of 2.41x.

Without the proposed technique, the RISC compiler generates an error and cannot compile. With the proposed design flow, the RISC compiler is able to construct the BM of the complex design and correctly perform static analysis, and gains speedup for the simulation.

2.6.3 Bitcoin Miner

We have implemented a Bitcoin miner model in SystemC to demonstrate the IP protection capability [72]. Our model consists of 3 stages: data preparation, scanning and result output, as shown in Figure 2.6. The scanning stage involves the use of multiple parallel scanners, which runs *SHA256* algorithm are provided as IP. In order to protect the IP, we have inspected *scanner.pd* and carefully redacted the PSG. The graphical view of the default and the redacted PD files of scanner are shown in Figure 2.7. We have removed several variable accesses because they do not actually result in conflicts. Furthermore, We have redacted the structure of PSG by removing an empty node. The new PSG is smaller and hides information about the detailed implementation of the scanner.

Table 2.2 shows the simulation speed of Bitcoin miner model with different number of scanners under both sequential and parallel simulations, using the default and redacted PD files of the IP. We have performed this experiment on a Xeon E1240 8-core processor. The speedup of the parallel simulation grows about linearly with the number of scanners. When there are 8 scanners, we get

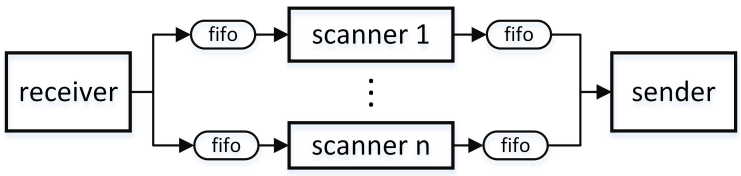


Figure 2.6: SystemC model of Bitcoin miner

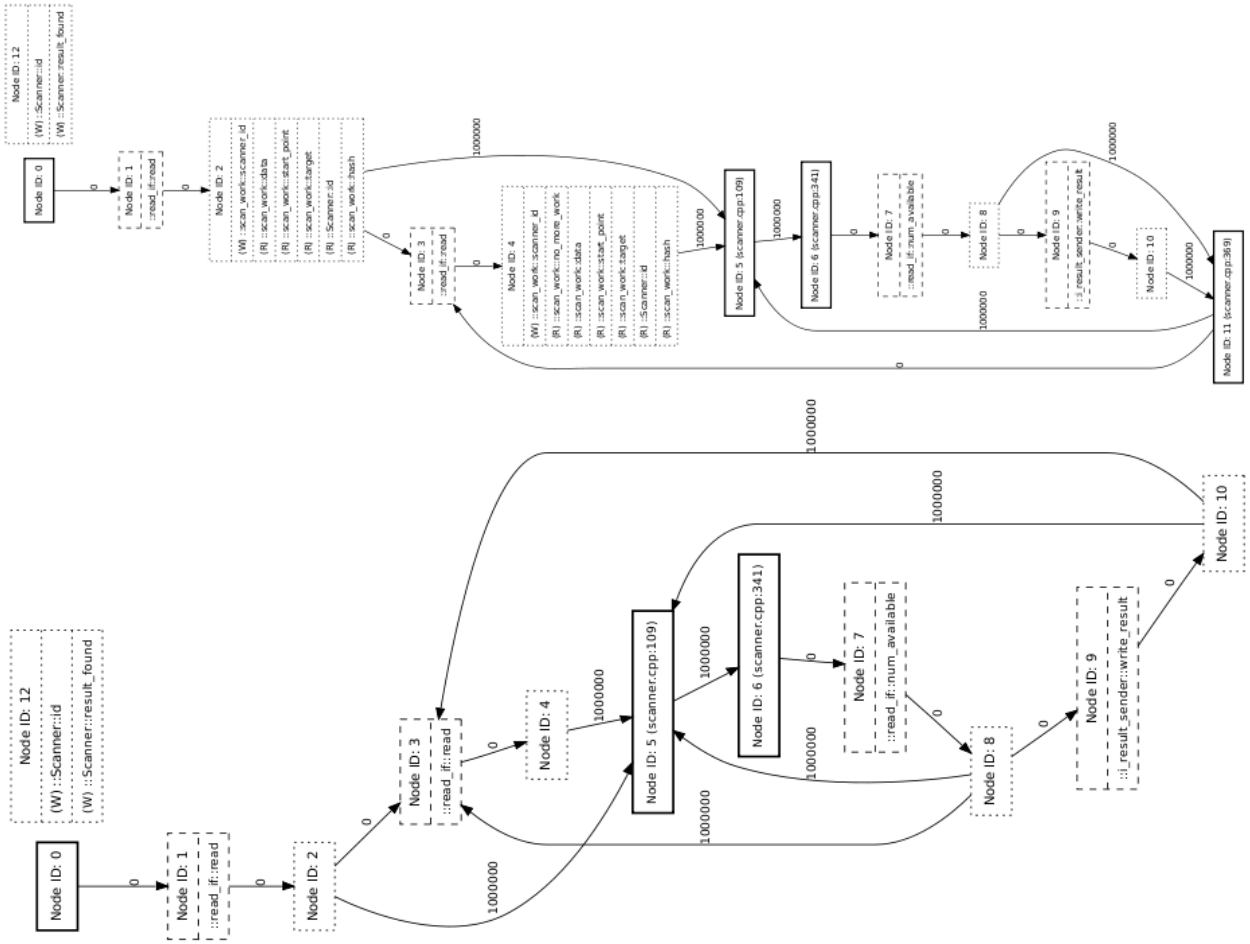


Figure 2.7: Original and modified PSG for scanner

the maximum speedup of 6.76x. A full speedup of 8 cannot be achieved because of the sequential part in the model and the scheduling overhead of OoO PDES. Another important observation is that the simulation result and speed does not change with the IP scanner. This demonstrates that our information hiding approach works well for IP protection.

Table 2.2: Simulation of Bitcoin Miner SystemC Model: runtime (secs) /speedup (%)

#scanner	SEQ	Original	Modified
1	117.68 / 1	117.69 /1.00	117.51/1.00
2	114.96 / 1	86.90 / 1.32	87.2/1.32
4	158.00 / 1	49.08 / 3.22	50.11/3.15
8	164.50 / 1	24.91 / 6.60	24.32/ 6.76

2.7 Conclusion

The work introduced in this chapter removes two scaling limitations of static-analysis based parallel SystemC simulation. The new PSG techniques enable the use of hierarchical input models with multiple translation units and the reuse of SystemC IP components. Third party IP is protected from security leakage by high abstraction from the source code, using automatically generated behavior model that the IP provider can further redact to meet trust expectations. The IP-enabled design flow is effective and sustains the speedup of advanced parallel simulation.

Chapter 3

Extending Static Analysis for Modern Transaction Level Models

In addition to scaling the RISC compiler in Chapter 2 for industrial SystemC designs, we investigate the support of SystemC TLM-2.0 standard in static analysis in this chapter [63]. We first introduce the new Socket Call Path technique and then propose an algorithm to statically analyze entangled variables.

3.1 Introduction

The Open SystemC Initiative (OSCI) TLM-2.0 [27] is a transaction level modeling standard released in 2008. The typical use of it is building virtual platforms to simulate today's large system on chip (SOC) models with processors, buss and other components [73, 74, 75]. SystemC TLM-2.0 consists of a set of core interfaces, sockets, generic payload and so on. These facilities are used to increase (a) interoperability between models, that is, the plug-and-play ability to take transaction level models from different sources and connect them together and (b) simulation speed.

As described in Section 1.1.3, Out-of-Order Parallel Discrete Event Simulation (OoO PDES) [34] has been designed for highly parallel SystemC simulation. Compared to traditional PDES, OoO PDES utilizes a finer grained task scheduler to allow suitable threads running in parallel even when they are in different cycles. This significantly increases the execution speed. In order to preserve the simulation semantics and timing accuracy, the input model is statically analyzed by a dedicated compiler to prevent potential race conditions between threads [34].

Conceptually, SystemC TLM-2.0 and OoO PDES are both library-based approaches and can be applied together. However, no analysis technique has been designed until today that is capable of determining safe parallelism between threads in SystemC TLM-2.0 models.

In this chapter, we propose and implement a static analysis for TLM-2.0 loosely-timed (LT) and approximately-timed (AT) SystemC models with multiple threads. Our approach is able to precisely and automatically analyze potential conflicts between threads that are communicating using the standard TLM-2.0 interfaces, and helps the model to achieve reasonable execution speedup given its parallelism potential. In this chapter, we describe in detail the analysis of the two mostly used APIs, namely the Blocking Transport Interface (BTI) and Direct Memory Interface (DMI). These two interfaces use the main TLM-2.0 features, such as generic payload, DMI objects and sockets. We also outline the support of the Non-Blocking Transport Interface (NBTI) and Debug Transport Interface (DTI) using the same techniques proposed in this chapter. Note that Register-Transfer Level (RTL) and TLM-1.0 models that do not use sockets are not targeted in this work. Given the results of the novel static analysis, a SystemC TLM-2.0 model can then be simulated safely and fast with OoO PDES, as demonstrated with extensive experiments and results in Section 3.8.

3.2 Background

In this section, we first introduce the motivation that lead to this work and then briefly review the TLM-2.0 interfaces [77, 29].

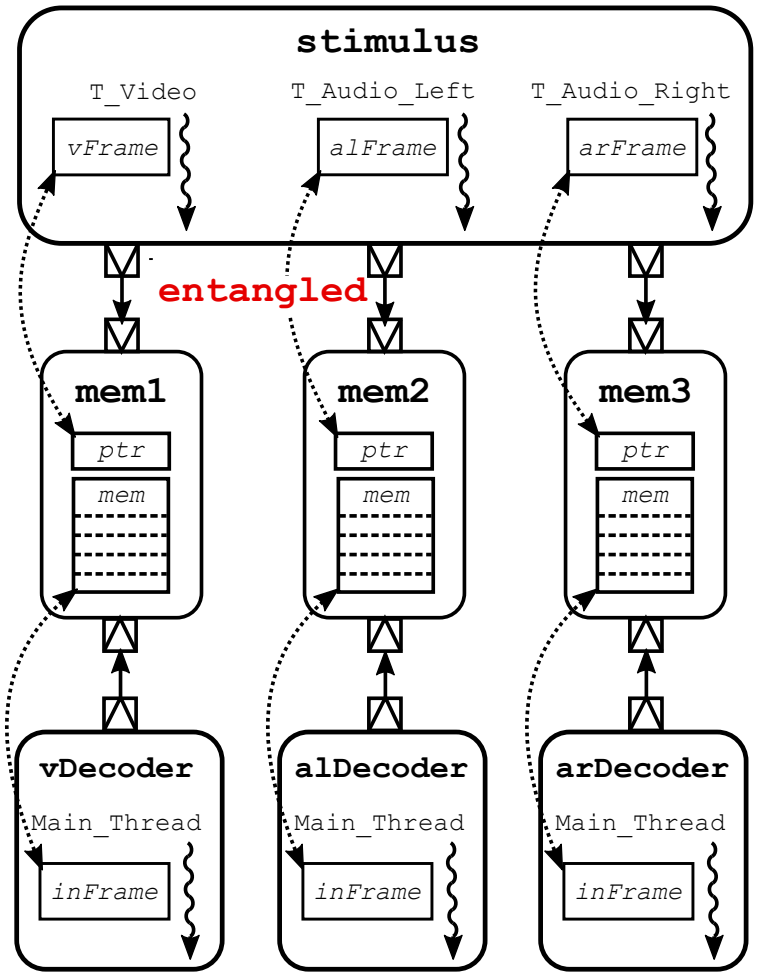


Figure 3.1: SystemC TLM-2.0 model of a DVD player

3.2.1 Motivation

OoO PDES has shown excellent results for faster simulation of SystemC TLM-1.0 based models. However, as pointed out in [76], TLM-2.0 is an obstacle for parallel simulation. In contrast to TLM-1.0, TLM-2.0 lacks the concept of channels. Instead, a module uses *pointers* to access memory locations in other modules. Since pointer analysis is difficult and communications are not encapsulated in containment constructs, no multi-thread access synchronization can be offered and race conditions are likely to occur, violating the execution semantics. [76] also proposed a conceptual solution for this problem, namely the idea of reintroducing channels into TLM-2.0 and protect the communication with locks. However, this would reduce simulator speed and violate

interoperability of the TLM-2.0 IEEE standard [27].

In order to simulate TLM-2.0 models safely and correctly with OoO PDES, we propose a new static analysis-based approach that protects communication without the need to modify the TLM-2.0 standard nor the application model. Our technique builds on top of an analysis algorithm for SystemC TLM-1.0 that is implemented in the Recoding Infrastructure for SystemC (RISC) [35]. It is a compiler based approach that automatically analyzes data, timing and event hazards among threads. However, the TLM-1.0 oriented static analysis is not able to analyze *variable entanglement* and is thus insufficient for the TLM-2.0 standard.

Definition 3.2.1. Variable Entanglement: Variable Entanglement occurs when one variable points to the memory location of another variable in other modules through the SystemC TLM-2.0 communication interfaces. Because two entangled variables refer to the same memory location, access (read/write) to one is also applied to the other.

As an example, Figure 3.1 shows a SystemC TLM-2.0 model of a DVD player which decodes a stream of H.264 video and MP3 audio data using separate decoders. All communications are modeled using TLM-2.0 sockets and APIs. Three parallel threads `T_Video`, `T_Audio_Left` and `T_Audio_Right` in the initiator `stimulus` store data into corresponding target memories `mem1`, `mem2` and `mem3`. Decoders `vDecoder`, `alDecoder` and `arDecoder` fetch the data from the memories and decode it. In this example, there are three parallel lanes. Take the `stimulus-mem1-vDecoder` lane as an example. Through the TLM-2.0 interfaces, variable `vFrame` of `stimulus` and pointer `ptr` of `mem1` are entangled (indicated by the dashed arrow in Figure 3.1), meaning that the two variables are pointing to the same memory location. Similarly, `inFrame` of `vDecoder` and `mem` of `mem1` are entangled as well. Figure 3.2 lists partial code of this simple SystemC model and shows in detail how the variables are entangled.

```
1 SC_MODULE(Stimulus){
2   void T_Video(){
3     tlm::tlm_generic_payload rgp;
4     rgp.set_data_ptr(vFrame);
5     ...//initilize rgp
6     sc_time delay;
7     while(true){
8     //keep sending vFrame
9       ...//generate vFrame
10      out1->b_transport(rgp, delay);
11      ...
12    }
13  }
14  tlm_utils::simple_initiator_socket
      out1; //socket
15  unsigned char vFrame[FRAMESIZE];
16  ...//other functions and variables
17 };

18 SC_MODULE(Memory){
19   void custom_b_transport(tlm::tlm_generic_payload &pgp, sc_time &delay)
20   {
21     unsigned char *ptr = pgp.get_data_ptr();
22     unsigned int len = pgp.get_data_length();
23     ... //get other pgp fields
24     memcpy(&mem[OFFSET], ptr, len);
25     ... //error handling, response
26   }
27   bool custom_get_dmi(tlm::tlm_generic_payload& pgp, tlm::tlm_dmi& pd)
28   {
29     pd.allow_read_write();
30     pd.set_dmi_ptr(&mem[OFFSET]);
31     ... //set other pd fields and pgp fields
32     return true;
33   }
34   Memory(){
35     in.register_b_transport(this, &Memory::custom_b_transport);
36     out.register_get_direct_mem_ptr(this, &Memory::custom_get_dmi);
37   }
38   //sockets and data
39   tlm_utils::simple_target_socket in;
40   tlm_utils::simple_target_socket out;
41   unsigned char mem[SIZE];
42   ...
43 };

44 SC_MODULE(VideoDecoder){
45   void Main_Thread(){
46     unsigned char inFrame[FRAMESIZE];
47     tlm::tlm_generic_payload rgp;
48     tlm::tlm_dmi rd;
49     ... // initialize rgp, rd
50     bool DMI_allowed =
         in->get_direct_mem_ptr(rgp, rd);
51     inFrame = rd.get_dmi_ptr();
52     while(DMI_allowed){
53     //keep decoding inFrame
54       decode(inFrame);
55     }
56   }
57   tlm_utils::simple_initiator_socket
       in; //socket
58   ...//other functions and variables
59 };
```

Figure 3.2: Partial SystemC Code for Figure 3.1

Blocking Transport Interface

Analyzing the variable entanglement information is critical for an accurate static analysis. It helps the SystemC compiler to handle the behavior of a pointer in a well-defined TLM-2.0 model. In general, statically analyzing the entanglement of general pointers is very difficult. However, we observe that the pointers for TLM-2.0 communication do not point to arbitrary memory locations. Through the well-defined TLM-2.0 interface methods, a pointer points to a determined memory

location and thus can be analyzed statically. In the above example, for instance, `inFrame` of `vDecoder` points only to `mem` of `mem1` through the TLM-2.0 DMI interface. The prior TLM-1.0 oriented static analysis does not take this variable entanglement information into account and treats `inFrame` as a pointer potentially pointing to any memory location in the entire model. Consequently, thread `vDecoder::Main_Thread` that accesses `inFrame` is not allowed to run in parallel with any other threads to prevent race conditions. Such too-strict pointer-involved data conflict analysis causes false conflicts and may reduce the simulator speed to sequential levels. Now by analyzing variable entanglement, the compiler is able to identify that `inFrame` of `vDecoder` and `mem` of `mem1` are pointing to the same memory location, accessing `inFrame` only causes data hazard with the concurrent access to `mem`. With the precise and correct data conflict analysis, the three lanes in the example in Figure 3.1 are able to run in parallel with no race conditions, rather than sequentially due to pointer conflicts imposed by the TLM-1.0 oriented static analysis.

The proposed approach in this chapter allows the SystemC compiler to analyze TLM-2.0 interfaces and socket connections. Based on this information, our approach then precisely analyzes the entangled variables. Note that our work does not support the analysis of general pointer operations, which is known to be hard for static analysis.

The work described in this chapter is inspired by [40] but different from it in several aspects. First, our Socket Call Path (SCP) technique is novel as it enables the compiler to analyze entangled pointers through TLM-2.0 interface APIs, which is not done in [40] (nor has been done in any other work to the best of our knowledge). Second, [40] focuses on the analysis of SystemC TLM-1.0 channels, whereas our work targets SystemC TLM-2.0. Note that TLM-1.0 and TLM-2.0 are entirely different in their communication modeling, because the channel concept has disappeared [76]. Note that TLM-2.0 was first identified as an obstacle for parallel SystemC simulation in [76]. Inter-thread communication is considered unsafe in a multi-thread environment when not encapsulated in a channel. To overcome the obstacle, the author proposed a conceptual solution that wraps the TLM-2.0 communication methods into actual channels (similar to TLM-1.0), so that locks could be implemented for protection. In contrast, our approach does not require extra

communication containment nor locks. We protect simulation semantics by precise static analysis only.

3.2.2 TLM-2.0 Background

This section briefly reviews the usage of TLM-2.0 BTI and DMI with a partial code that describes the `stimulus-mem1-vDecoder` lane in Figure 3.1. The code snippet is shown in Figure 3.2. In the code, `Stimulus` is the module type of `stimulus`, `Memory` is the module type of `mem1` and `VideoDecoder` is the module type of `vDecoder`. TLM-2.0 focuses mainly on the communication between processes rather than computation, so the details of encoding and decoding algorithms are not shown in the code. We first examine the Blocking Transport Interface (BTI). It is used for the communication between `stimulus` and `mem1` in Figure 3.1. Basically, it takes four steps to use BTI. On the initiator's side:

1. prepare a generic payload object and a timing annotation. In this example, `rgp` is defined in line 3 and `delay` is defined in line 6. `rgp` stands for *Referred Generic Payload* and will be described in Section 3.3.3. In line 4, the video frame `vFrame` is wrapped inside `rgp` by `set_data_ptr`. Note that initializations of other `rgp`'s fields (such as data length, streaming width and so on) are omitted in this demo code.

2. call `b_transport` via the initiator socket, passing the prepared generic payload object and timing annotation as the arguments. In this example, `b_transport` is called on socket `Stimulus::out1` with `rgp` and `delay` as arguments, in line 10.

On the target's side:

1. implement a callback for `b_transport`. In this example, we implement `Memory::custom_b_transport` in line 19 as the callback method. The callback method takes a reference

of generic payload pgp and the timing annotation delay as parameters. pgp stands for *Parametric Generic Payload* and will be described in Section 3.3.3. In line 21, the pointer wrapped inside pgp is extracted by get_data_ptr and assigned to ptr. The data length is extracted in line 22. Other extractions of pgp's fields are omitted in this demo code. Line 24 copies ptr's value to Memory::mem at offset OFFSET.

2. register the callback on the target socket. In the Memory's constructor, Memory::
custom_b_transport is registered as the b_transport callback method on socket Memory::in, in line 35.

A corresponding callback is executed on every b_transport call [1]. Consequently, the actual behavior of a b_transport is fully defined by its callback function. In the stimulus-mem1-vDecoder lane, stimulus is connected to mem1 via socket binding stimulus.out1→mem1.in. When stimulus.
T_Video executes out1->b_transport, it invokes mem1.custom_b_transport. In this case, the behavior of this b_transport is represented by Memory::custom_b_transport.

Direct Memory Interface

The Direct Memory Interface (DMI) is used for the communication between vDecoder and mem1 in Figure 3.1. Similar to BTI, it takes five steps to use DMI. On the initiator's side,

1. prepare a generic payload object and a dmi object. In this example, thread VideoDecoder::
Main_Thread declares rgp at line 47 and rd at line 48. rd stands for *Referred DMI object* and will be described in Section 3.4.

2. call get_direct_mem_ptr via the initiator socket, passing the prepared generic payload object and DMI object as the arguments. In this example, get_direct_mem_ptr is called on socket

[1] Section 3.3.1 explains how we match a b_transport call to the registered callback method(s)

VideoDecoder::in with rgp and rd as arguments, in line 50. The *Boolean* flag DMI_allowed indicates if the DMI access is permitted by the target.

3. extract the pointer of the directly accessed memory location from the dmi object via get_dmi_ptr. In line 51, inFrame is the extracted pointer from rd, through which videoDecoder::Main_Thread is able to access the memory location in the connected target module instance directly.

On the target's side:

1. implement a callback for get_direct_mem_ptr. In this example, we implement Memory::custom_get_dmi in line 27. The callback method takes a reference of generic payload pgp and a reference of dmi object pd as parameters. pd stands for *Parametric DMI object* and will be described in Section 3.4. In line 30, the pointer to Memory::mem at offset OFFSET is wrapped inside pd by set_dmi_ptr. Other initializations of pd's fields are omitted in this demo code.

2. register the DMI callback on the target socket. In the Memory constructor, Memory::custom_get_dmi is registered as the DMI callback method on socket Memory::out, in line 36.

In the stimulus-mem1-vDecoder lane, mem1 and vdecoder are connected via socket binding vDecoder.in→mem1.out. Through get_direct_mem_ptr, vDecoder gets the direct memory access to mem1.mem.

3.3 Static Analysis for Blocking Transport Interface

The BTI is appropriate when an initiator wants to complete a transaction with a target during the course of a single function call, to be specific, b_transport. In this section, we first discuss the approach to build the needed SG for b_transport call. Then, we introduce our new SCP technique that provides context information about socket bindings. Finally, we propose an approach for analyzing variable entanglement.

3.3.1 Segment Graph for `b_transport`

The TLM-2.0 BTI is intended to support the loosely-timed coding style. The source code in Figure 3.2 demonstrates the use of BTI. As discussed in Section 3.2.2, the behavior of a `b_transport` is represented by its registered callback method on the target's side. Thus, to build the SG for a `b_transport` call, the RISC compiler must be able to find the function definition of the corresponding callback method. In our proposed approach, this is done in three steps:

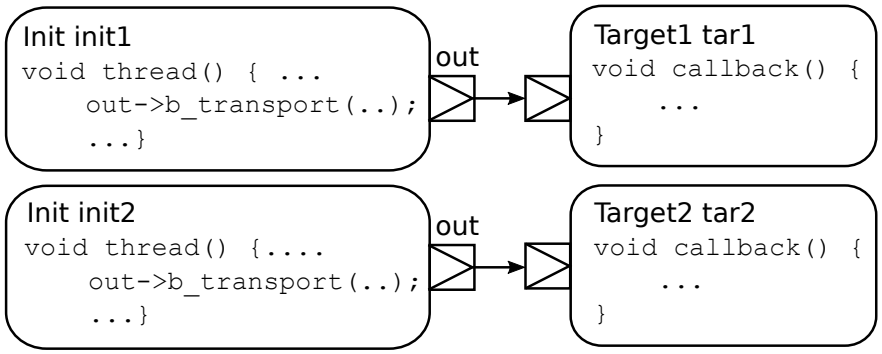
1. for each target socket, identify the registered `b_transport` callback method.

2. use a recursive approach to obtain the socket binding information. The approach is recursive because an initiator socket may bind to another intermediate initiator/target socket, and we need to get an end-to-end binding information.

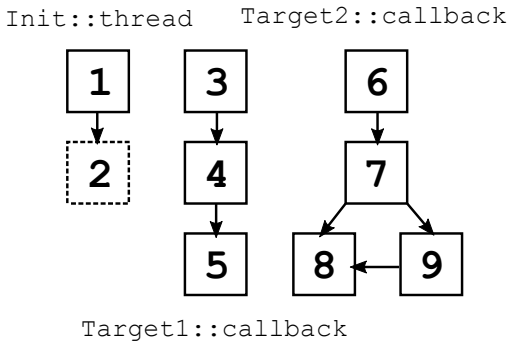
3. when a `b_transport` call is encountered in a calling segment, the compiler first identifies the bound target socket according to the socket binding information, then builds the SG of the callback method registered on this target socket. Finally, the compiler merges the SG into the calling segment.

Note that a single `b_transport` can potentially have multiple callback methods. An example is shown in Figure 3.3a. In this example, `init1` and `init2` are module instances of the same module type `Init`, whereas `tar1` and `tar2` are of type `Target1` and `Target2` respectively. When `out->b_transport` is executed in `Init::thread`, either `Target1::callback` or `Target2::callback` may be called depending on which `Init` module instance the `b_transport` belongs to. Note that an SG represents threads at module level but not instance level. This means that when building the SG for `Init::thread`, the compiler cannot determine if the current `Init::thread` is called from module instance `init1` or `init2`. Thus, our proposed static analysis takes this flexible binding behavior into account. We illustrate processing of callback methods in Figure 3.3b and 3.3c. Initially, `Init::thread` has two segments: segment 1 and 2. Segment 2 is

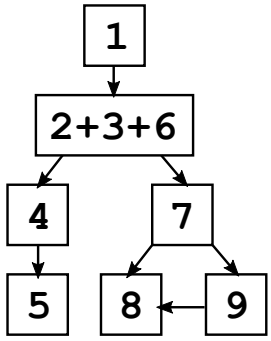
being constructed (depicted as dashed border) and here `out->b_transport` is called. Then, the compiler builds temporary SGs for `Target1::callback` and `Target2::callback`, where segments 3, 4 and 5 belong to `Target1::callback` and segments 6, 7, 8, and 9 belong to `Target2::callback`. Since `Target1::callback` and `Target2::callback` are both potential callbacks of `out->b_transport`, their temporary SGs are joined into the complete SG of `Init::thread` in Figure 3.3c. Specifically, segment 3 and segment 6 are both merged with segment 2, represented by 2+3+6 in Figure 3.3c.



(a) SystemC model with two initiators and two targets



(b) Initial SG of `Init::thread` and temporary SGs of callback methods



(c) Complete SG of `Init::thread` after merging

Figure 3.3: Example of merged SGs of multiple callbacks

3.3.2 Socket Call Path

SCP is a new advanced technique in our approach. It provides the SystemC compiler with the information regarding how a target is reached by the initiator through the TLM-2.0 interface. The idea is similar to the Port Call Path (PCP) analysis proposed in [40]. One main difference is that PCP is based on port-to-channel connections whereas SCP is for analyzing socket-to-module connections. Also different from PCP, a SCP is represented by a list of sockets.

When used together with SG, SCP helps the SystemC compiler to perform instance-aware conflict analysis, which provides similar benefits as to the use of PCP in [40]. Figure 3.4 shows a SCP-included SG of `Stimulus::T_Video` in Figure 3.2. Segment 1 contains code statements in lines 3-10 from `Stimulus::T_Video` and lines 21-25 from the callback method `Memory::custom_b_transport`. On one hand, lines 3-10 are local to `Stimulus::T_Video` and thus have an empty SCP. On the other hand, lines 21-25 are accessed through socket `Stimulus::out1` and their corresponding SCP is [Stimulus::out1]. Given the SCP information, the SystemC compiler looks up the socket binding information and identifies that when lines 21-25 are executed by thread `T_Video` of `stimulus`, they belong to `mem1`. The SCP-included SG captures an instance-aware behavior of threads and thus increases the precision of conflict analysis.

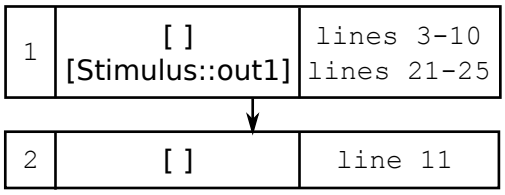


Figure 3.4: SCP-included SG of `Stimulus::T_Video` in Figure 3.2

3.3.3 Variable Entanglement Analysis

Variable entanglement is defined in Definition 3.2.1. Figure 3.1 in Section 3.2.1 shows an example of variable entanglement through TLM-2.0 interfaces. Without precise analysis, entangled variables result in overwhelming false conflicts and thus compromise the simulation. In this section, we propose a three-step approach to analyze variable entanglement and variable access for entangled variables.

Identify Original and Alias Variable

In this step, the compiler identifies (a) the original variable encapsulated in a generic payload by `set_data_ptr`, and (b) the alias variable extracted from a generic payload by `get_data_ptr`. In Figure 3.1, `vFrame` in line 4 is an original variable and `ptr` in line 21 is an alias variable. The original and alias variables are stored in a table data structure for later use.

Reference Analysis for Generic Payload with SCP

In the second step, the compiler analyzes the mapping between *parametric generic payload* (PGP) and *referred generic payload* (RGP). PGP is a reference parameter of a `b_transport` callback method, for instance, `pgp` in line 19 of Figure 3.2. RGP is the generic payload passed to a `b_transport` call, for instance, `rgp` in line 3 of Figure 3.2. A PGP refers to RGP(s) through BTI.

Because a `b_transport` callback method may serve as the callback of multiple `b_transport` calls from various initiators, its PGP can potentially refer to multiple RGPs. In the example in Figure 3.5, `Target::callback()` is the callback method for both `b_transport` calls in `Init1::thread` and `Init2::thread`. At runtime, `pgp` of `Target::callback()` refers at different times to the two RGPs: `Init1::rgp` and `Init2::rgp`. However, the static compiler is not able to further

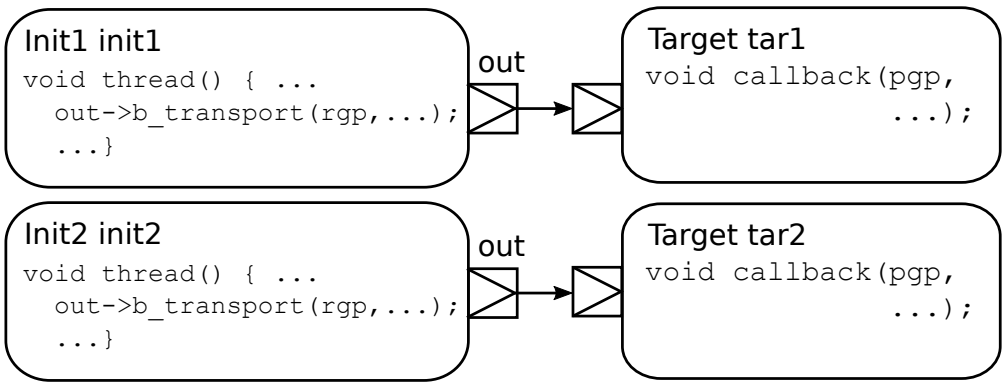


Figure 3.5: SystemC model with two initiators and two targets

identify the exact RGP that a PGP refers to in a given context. For instance, the compiler cannot figure out that `pgp` actually refers to `Init1::rgp` but not `Init2::rgp` when `Target::callback` is invoked by the `b_transport` in `Init1::thread`. Such ambiguity causes many false conflicts if a PGP refers to a large number of RGPs.

To solve this problem, we attach SCP as a context information to each RGP. As described in Section 3.3.2, a SCP represents the path from a `b_transport` call to its callback method. Since RGP is the argument of a `b_transport` call and PGP is the parameter of the callback method, the SCP also represents the connection between RGP and PGP. For the example in Figure 3.5, when `Target::callback` is invoked by the `b_transport` in `Init1::thread`, `pgp` refers to `Init1::rgp` via SCP [Init1::out]. On the other hand, when `Target::callback` is invoked by the `b_transport` in `Init2::thread`, `pgp` refers to `Init2::rgp` via SCP [Init2::out]. The SCP-included PGP-RGP reference mappings are stored in a table data structure for later use.

Variable Access Analysis for Entangled Variables

Through the PGP-RGP reference mappings, the corresponding alias and original variables are entangled. Algorithm 2 shows the algorithm to analyze variable accesses for entangled variables within a segment. It is divided into two parts. First, two sets $\mathbb{R}$ and $\mathbb{W}$ are created in lines 3-5 to store read and written variables in the segment. In particular, we traverse each expression of the

segment and identify the accessed variables with AnalyzeExpression. Second, in lines 8-33, the algorithm checks for each accessed variable if it has an entangled variable. If so, the entangled variable is also added to the corresponding set.

Take segment 1 in Figure 3.4 and Figure 3.2 as an example. When expression `memcpy( &mem[OFFSET],ptr,len)` in line 24 is encountered, Algorithm 2 first analyzes the variables accessed in this expression and assigns them to corresponding variable access sets. In this example, `mem` with SCP [Stimulus::out1] is assigned to $\mathbb{W}$, `ptr` with SCP [Stimulus::out1] and `len` with SCP [Stimulus::out1] are assigned to $\mathbb{R}$. Next, Algorithm 2 identifies alias variables in the two sets according to the information collected in the step in Section 3.3.3. The only alias variable is `ptr` in $\mathbb{R}$. It is extracted from `pgp` in line 21 of Figure 3.2. A PGP can refer to multiple RGPs as described in the step in Section 3.3.3. We collect them all in a set $\mathbb{RGP}$. Since `ptr` is reached through SCP [Stimulus::out1], given this context information, `pgp` should refer to an RGP that has the same SCP, [Stimulus::out1]. In line 27 of Algorithm 2, such RGP is identified, which is `rgp` in `Stimulus::T_Video`. It encapsulates an original variable `vFrame`. Finally, `vFrame` is assigned to $\mathbb{R}$.

While a detailed theoretical complexity analysis is beyond the scope of this chapter, we note that the size of the analysis tasks performed by the compiler is proportional to the model size. Specifically for Algorithm 2, all function calls (e.g., IsAliasVar, IdentifyPGPs) performed inside have constant time complexity[2]. Thus, the overall time complexity of Algorithm 2 is $O(N^2)$ due to the nested `for` loops, where N is the total number of read and written variables in the segment.

3.4 Static Analysis for Direct Memory Interface

In this section, we discuss the static analysis for TLM-2.0 DMI. It is consistent with the approach for BTI with a few key differences.

[2]Technically, these functions are implemented to look up hash tables that are constructed in previous steps

Algorithm 2 Variable access analysis for entangled variables

```
 1: function AnalyzeVariableAccess(seg)
 2:     //build the variable read and write sets
 3:     for all exprWithScp ∈ seg do
 4:         AnalyzeExpression(exprWithScp, ℝ, 𝕎)
 5:     end for
 6:
 7:     //add entangled variables to variable write set
 8:     for all varWithScp ∈ 𝕎 do
 9:         if IsAliasVar(varWithScp) then
10:             pgp ← IdentifyPGP(varWithScp)
11:             ℝ𝔾ℙ ← IdentifyRGPs(pgp)
12:             for all rgp ∈ ℝ𝔾ℙ do
13:                 if GetSCP(rgp) == GetSCP(varWithScp) then
14:                     originalVar ← GetOriginalVar(rgp)
15:                     AddVariableAccess(originalVar, 𝕎)
16:                 end if
17:             end for
18:         end if
19:     end for
20:
21:     //add entangled variables to variable read set
22:     for all varWithScp ∈ ℝ do
23:         if IsAliasVar(varWithScp) then
24:             pgp ← IdentifyPGP(varWithScp)
25:             ℝ𝔾ℙ ← IdentifyRGPs(pgp)
26:             for all rgp ∈ ℝ𝔾ℙ do
27:                 if GetSCP(rgp) == GetSCP(varWithScp) then
28:                     originalVar ← GetOriginalVar(rgp)
29:                     AddVariableAccess(originalVar, ℝ)
30:                 end if
31:             end for
32:         end if
33:     end for
34: end function
```

The TLM-2.0 DMI is designed to speed up simulation by giving an initiator a direct pointer to an area of memory in a target. It involves two directions of communication paths:

1. *Forward path* lets the initiator request a direct memory pointer from a target with the `get_direct_mem_ptr` method call. In the example in Figure 3.1, `inFrame` of `vDecoder` is entangled with `mem` of `mem1` through the forward DMI path.

2. *Backward path* lets the target invalidate a DMI pointer previously given to an initiator with the `invalidate_direct_mem_ptr` method call.

To support the DMI in the proposed approach, we only need to consider the forward path. The backward path is not analyzed because a static analysis must be conservative and consider variable entanglement at all times, not only between `get_direct_mem_ptr` and `invalidate_direct_mem_ptr` function calls. Besides, the backward path does not entangle variables.
We utilize a similar approach as for BTI to statically analyze DMI. The approach involves two main steps:

1. Construct SCP included SG for `get_direct_mem_ptr` call. The same approach as in Section 3.3.1 and 3.3.2 is used. The compiler first identifies the callback method of a `get_direct_mem_ptr` call using the socket binding information. Then, the SG of the callback is built and merged to the calling segment with a correct SCP.

2. Analyze variable entanglement through DMI. A similar approach as the three steps in Section 3.3.3 is used. The SystemC compiler first analyzes the original variable wrapped inside a *parametric DMI object* (PD) and the alias variable extracted from a *referred DMI object* (RD). In Figure 3.2, `Memory::mem` is the original variable wrapped inside `pd` in line 30. `inFrame` of `VideoDecoder::Main_Thread` is an alias variable extracted from `rd` in line 51. Next, the compiler analyzes the reference mapping between PD and RD. In Figure 3.1, `vDecoder` is connected to `mem1`. With this module interconnect information, the SystemC compiler identifies that in Figure 3.2, `Memory::custom_get_dmi` is the DMI callback method of `get_direct_mem_ptr` in line 50, and thus `pd` refers to `rd` via SCP [VideoDecoder::in]. Finally, the variable accesses for entangled variables are analyzed. From the previous two steps, the compiler obtains the information that `pd` wraps `Memory::mem` and `rd` releases `inFrame`, and `pd` refers to `rd` via SCP [VideoDecoder::in]. Consequently, `inFrame` is entangled with `Memory::mem` via SCP [VideoDecoder::in]. Since `inFrame` is read in line 54 in Figure 3.2, `Memory::mem` with SCP [VideoDecoder::in] is thus

assigned to $\mathbb{R}$ of the corresponding segment.

3.5 Static Analysis for Non-blocking Transport Interface

Non-blocking Transport Interface (NBTI) is intended to support the approximately-timed coding style. It breaks down a transaction into multiple timing points, and typically requires multiple function calls with phase information for a single transaction. Two interface methods `nb_transport_fw` and `nb_transport_bw` are used for forward and backward communications between initiators and targets. Similar to `b_transport`, the behaviors of `nb_transport_fw` and `nb_transport_bw` are represented by their corresponding registered callback methods.

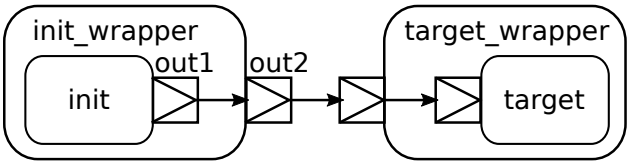
`nb_transport_fw` and `nb_transport_bw` take three parameters: a generic payload, a timing annotation and a phase object. The generic payload and timing annotation are of the same use as for `b_transport`. The phase object is used to indicate the current phase of a transaction. Since a static analysis is conservative and considers variable entanglement at all times, not only between certain phases, transaction phase updates need not to be treated specially in a static analysis. With all the above observations, NBTI can be analyzed the same way as for BTI. The SystemC compiler first builds the SCP-included SG for `nb_transport_fw` and `nb_transport_bw` according to the registered callback methods and socket binding information. Then, it analyzes the variable entanglement through NBTI. BTI and NBTI both entangle variables through the PGP-RGP reference mappings, and thus variable entanglement by NBTI can be analyzed using the approach proposed in Section 3.3.3. One main difference between NBTI and BTI is that NBTI contains backward paths (form target sockets to initiator sockets) used by `nb_transport_bw`, whereas in BTI there are only forward paths (from initiator sockets to target sockets). To support the `nb_transport_bw` API, we augment our SystemC Internal Representation with backward socket mapping information.

3.6 Static Analysis for Debugging Transport Interface

Debugging Transport Interface (DTI) is used for debug access which gives an initiator the ability to read or write memory in the target without delays or side-effects. `transport_dbg` is the corresponding interface method and takes only a generic payload as parameter. Conceptually, DTI can be statically analyzed the same way as for BTI. SCP-included SG for `transport_dbg` is firstly built and variable entanglement through DTI is analyzed using the approach proposed in Section 3.3.3.

3.7 Static Analysis for Indirect Communication

In the previous sections, all examples involve only direct communication between initiators and targets. Nevertheless, our proposed solution also works for indirect TLM-2.0 communication, for instance, communication through hierarchical modules and interconnect components such as routers and bridges. Two corresponding examples are shown in Figure 3.6a and 3.6b.



(a) Hierarchical communication

(b) Interconnected communication

Figure 3.6: Example of indirect communications

3.7.1 Hierarchical Communication

In Figure 3.6a, an initiator `init` is connected to a target `target` across wrappers `init_wrapper` and `target_wrapper`. With wrappers around, a target is reached by an initiator through multiple socket bindings. As described in Section 3.3.2, a SCP is a list of sockets from an initiator to a target. In this example, `init` reaches `target` through two initiator socktets: `out1` and `out2`. Thus the SCP from `init` to `target` is [out1→out2]. With the correct SCP information, variable accesses for entangled variables are analyzed the same way as for direct communication.

3.7.2 Interconnected Communication

Interconnections such as routers and buses are common in SystemC models. In the example in Figure 3.6b, `interconnect` forwards the communication from `init` to `target`. With an interconnecting module, a target is reached by an initiator through multiple initiator sockets, and thus the SCP contains multiple sockets. In Figure 3.6b, the SCP from `init` to `target` is [out1→out2]. With the correct SCP information, variable accesses for entangled variables are analyzed the same way as for direct communication.

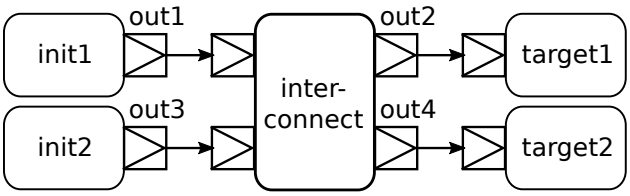


Figure 3.7: Interconnected communication with multiple initiators and targets

It is common that an interconnect module connects to multiple initiators and targets. An example is shown in Figure 3.7. Initiators `init1` and `init2` connect to targets `target1` and `target2` via `interconnect`. Because the compiler cannot statically identify which target a TLM-2.0 API call in the initiator `init1` is routed to during run time, our approach has to analyze both possibilities. Specifically, `init1` reaches `target1` via SCP [out1→out2] and reaches `target2` via

SCP [out1→out4], `init2` reaches `target1` via SCP [out3→out2] and reaches `target4` via SCP [out3→out4]. Variable accesses for entangled variables are analyzed the same way as for direct communication based on such SCP information.

3.8 Experiments and Results

We have implemented the proposed static analysis approach as an extension of the SystemC compiler download from [35]. The project is written in C++ and uses the ROSE infrastructure [37] to build an AST of the input SystemC model. Based on the AST and the RISC internal representations, we are able to analyze sockets mappings, TLM-2.0 APIs and variable entanglements. We have evaluated our approach with demonstration and real-world examples. All the examples are SystemC TLM-2.0 approximately- or loosly-timed models with multiple threads and some parallelism potential. For evaluation, we measure the number of conflicts and the execution times under the sequential Accellera simulator (Seq) and the parallel RISC simulator (Par). Note that with the previous RISC compiler and simulator, none of these examples would actually compile. The compiler would output an error message about the use of TLM-2.0 constructs, such as unknown sockets and interface methods. Even if the model would pass the compiler, all pointers would result in conflicts with all other segments, leaving no threads available for parallel execution in the simulator. Thus, the examples would run only sequentially, similar as in the Accellera simulator. Our experiments execute on an Intel Xeon E3-1240 multi-core processor with 4 cores, 2-way hyperthreaded. The CPU frequency-scaling was turned off so as to provide accurate and repeatable results.

3.8.1 Demonstration Examples

The examples in this experiment are derived from the loosely-timed `LT` models in the Accellera SystemC library [78]. 36 examples are used with the following configurable variations:

1. TLM-2.0 interface type: BTI, DMI or NBTI.

2. communication type: Direct, Hierarchical (Hier) or Interconnected (Inter).

3. varying number of lanes.

A generic block diagram is shown in Figure 3.8. The dotted lines indicate varied modules or socket connections. Despite the configurable variations, the number of threads in each initiator and target remain fixed. An initiator has two SC_THREADs where `th1` performs pure computation and `th2` performs communication, and a registered callback method for the backward NBTI path. A target has one SC_THREAD `th` and three registered callback methods for BTI, DMI and the forward NBTI path. The interconnect module has registered callbacks that route the communication. During communication, `Initiator::th2` accesses memory locations in both the initiator and the target and thus has data conflicts with `Initiator::th1` and `Target::th`. The initiators, targets, wrappers and interconnects are respectively different instances of the same module types.

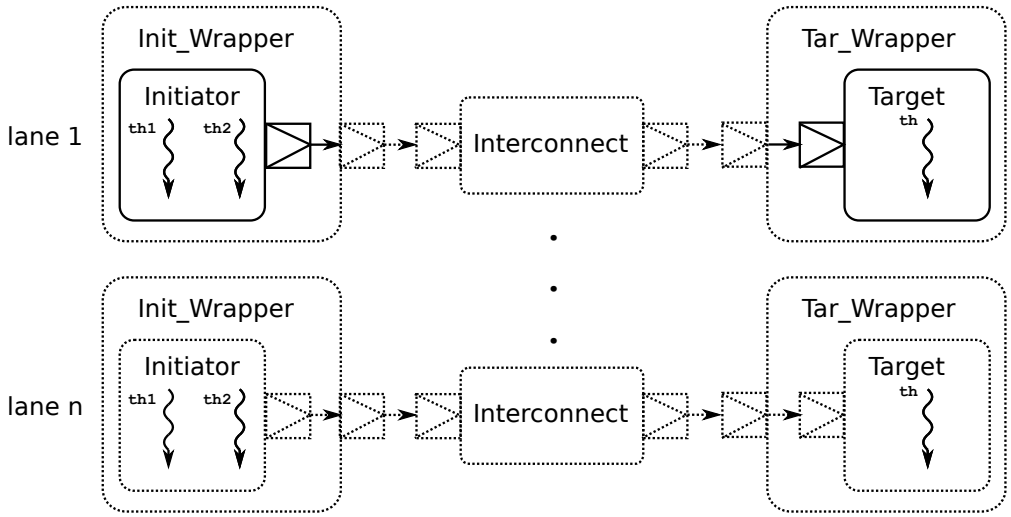


Figure 3.8: Block diagram for the demonstration example

The sequential (Seq) and Out-of-Order parallel (Par) simulator run-times and speedups under different model configurations are shown in Table 3.1, 3.2 and 3.3. Table 3.4 shows the percentage

Table 3.1: Results of BTI examples from Accellera : run-time (secs) and speedup (%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	41.0	21.5	191%	41.2	21.5	192%	41.3	21.7	190%
2	80.8	22.5	359%	81.1	21.4	379%	81.7	22.5	363%
4	160.6	29.6	543%	162.0	29.9	542%	161.3	30.4	531%
8	320.8	59.2	542%	320.7	57.3	560%	320.7	58.0	553%

Table 3.2: Results of DMI examples from Accellera: run-time (secs) and speedup (%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	41.7	21.6	193%	41.3	21.8	190%	41.4	21.7	191%
2	81.3	23.0	353%	81.1	22.4	362%	81.3	22.7	358%
4	161.2	30.8	523%	161.1	29.8	541%	161.2	29.5	546%
8	320.0	57.6	556%	322.0	58.4	551%	319.5	57.3	558%

Table 3.3: Results of NBTI examples from Accellera: run-time (secs) and speedup (%)

	Direct			Hierarchical			Interconnect		
Lanes	Seq	Par		Seq	Par		Seq	Par	
1	40.2	20.1	200%	40.6	20.3	200%	40.5	20.3	200%
2	81.1	21.2	383%	80.7	20.8	388%	80.6	21.5	375%
4	159.2	28.7	554%	159.1	28.3	561%	161.2	28.9	558%
8	320.0	56.2	569%	320.1	57.2	559%	317.7	56.0	567%

Table 3.4: Percentage of reduced false variable entanglements in the demonstration examples

	BTI			DMI			NBTI		
Lanes	Direct	Hier	Inter	Direct	Hier	Inter	Direct	Hier	Inter
1	33%	33%	33%	33%	33%	33%	33%	33%	33%
2	73%	73%	73%	73%	73%	73%	71%	71%	71%
4	88%	88%	88%	88%	88%	88%	87%	87%	87%
8	94%	94%	94%	94%	94%	94%	94%	94%	94%

of reduced false variable entanglements over all entanglements. Take the model with configuration *BTI+Inter+2-Lanes* as an example. An `Initiator` uses one pointer (in local module scope), `Interconnect` uses no pointer, and `Target` uses two pointers (one in local scope and the other in module scope). Since there are two lanes, the total number of pointers is six, and therefore there are

15 pairs of pointers in combination. Without the analysis for variable entanglement, all 15 pairs are considered as total conflicts. Now with our approach, the compiler is able to use context information and identify that only variables that belong to the same lane are entangled. Specifically, only four pairs of pointers are truly entangled. Thus 73% of entangled variable pairs are false entanglements and reduced by our approach. This results in much fewer conflicts in the data conflict table. Note that the numbers of pointers and truly entangled pointer pairs do not change with communication types. Therefore, the percentages of reduced false variable entanglements are the same for different communication types. On the other hand, the numbers of pointers and truly entangled pointer pairs are affected by interface types. Due to the backward path, the NBTI models have one more pointer in the callback method in `Initiator` than the BTI or DMI models. Therefore, the percentages of reduced false variable entanglements are the same for BTI and DMI, but slightly different for NBTI.

The data conflict table is shown in Figure 3.9. The indexes of the table have the form `(Segment ID, module Instance ID)`. For instance, `(2,0)` represents a segment with ID 2 and belongs to a thread of a module instance in the first lane, whereas `(2,1)` is the same segment with ID 2 but belongs to a thread of a module instance in the second lane. A red entry represents data conflicts, a green entry represents conflict free and a yellow entry represents the eliminated false conflicts by applying our SCP technique and variable entanglement analysis. The data conflict table shows that with our proposed static analysis, the number of data conflicts is reduced from 144 to 56. More than 60% of data conflicts are pointer-related false conflicts and are eliminated. The experimental results allow the following observations:

1. Variable entanglement analysis largely reduces false conflicts due to pointers. Without the variable entanglement analysis, the pointers impose data conflicts with other segments and thus cause overwhelming false conflicts, as indicated by the yellow entries in Figure 3.9. Our approach is able to identify the exact memory location a pointer points to through the TLM-2.0 interfaces and thus enables precise data conflict analysis.

2. The SCP technique allows instance-aware conflict analysis. Our SCP technique provides context information to the compiler to distinguish threads that belong to different module instances. Without the SCP technique, thread `th1` in the first `Initiator` module instance cannot run in parallel with `th2` in the second `Initiator` module instance due to false conflicts. In the data conflict table, this for instance results in a false conflict between (1,0) and (3,1).

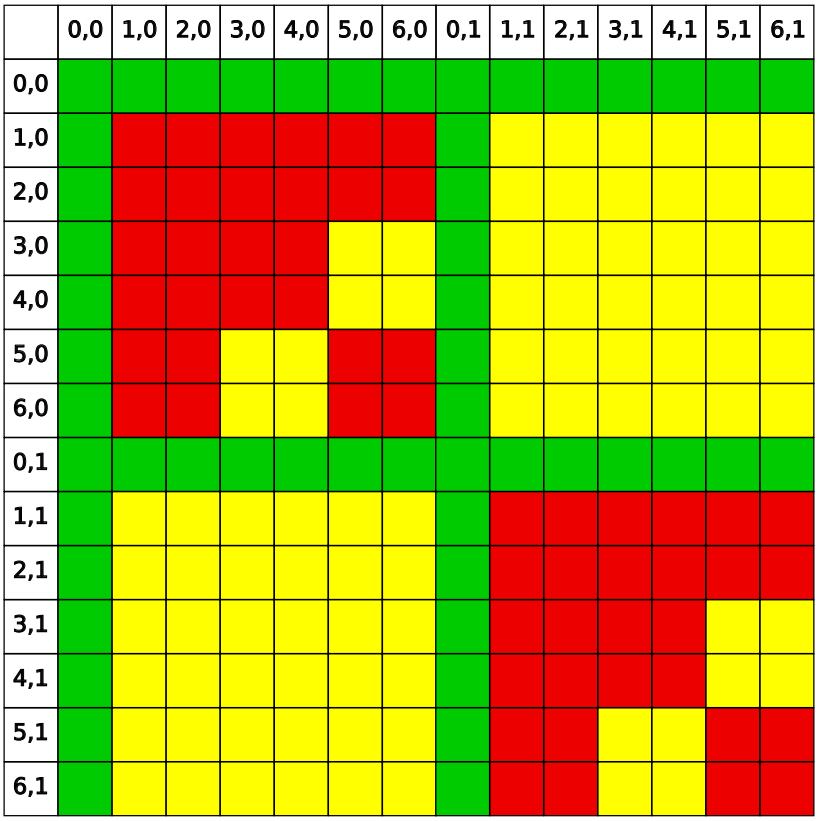


Figure 3.9: Data conflict table for BTI+Inter+2-Lanes

Given the novel analysis results, the OoO PDES simulator is able to execute all computation threads in the `Initiator` and `Target` in parallel and achieve a speedup of about 360% compared with the sequential execution. The ideal speedup of 400% is not achievable because of two main reasons: (a) the communication thread `Initiator::th2` cannot run in parallel with other threads due to data conflicts and causes a sequential bottleneck. According to Amdahl's law, this reduces the overall parallelism of the model (b) the OoO PDES simulator has a run-time overhead of dynamic checking. Other examples with different model configurations also demonstrate impressive speedups. A

maximum speedup of over 550% is achieved for the 8-lane models. Overall, the demonstration examples show the correctness and effectiveness of the proposed static analysis for SystemC TLM-2.0 models.

3.8.2 DVD Player

Now we evaluate our approach using a real world DVD Player example which is similar to the one in Figure 3.1. The block diagram of the model is shown in Figure 3.10. `Stimulus` has three parallel threads that feed data into a memory `Memory` for the decoders to fetch. After decoding, the decoded results are passed to monitors to verify the correctness. One main difference from the model in Figure 3.1 is that there is only one memory module. This does not affect the parallelism of the model because different data flow lanes have their own memory locations inside `Memory`. Similar to the demonstration examples in Section 3.8.1, the DVD Player model is also configurable with the combination of following options: BTI/DMI/NBTI and Direct/Hierarchical/Interconnect. Note that the varied modules are represented by dashed lines. The two wrapper modules `Decoder` and `Monitor` are enabled under the "Hierarchical" configuration. `Router` is enabled under the "Interconnect" configuration. The results of run-time and speedup comparing to sequential simulation are shown in Table 3.5.

The results demonstrate that the model gets a speedup of around 280% under OoO PDES and is consistent over all model configurations. The consistency is reasonable because the TLM-2.0 communication and connection do not have impact on the threads' behavior and thus will not affect speed much. It is also notable that the theoretical maximum speedup of 300% is not achieved for this three-lane model. This is explainable because the OoO PDES scheduler needs to perform the scheduling and to decide thread dispatching order and thus incurs overhead. Nevertheless, 280% is an impressive value for a three-lane model.

The results confirm the correctness and effectiveness of the proposed static analysis for SystemC TLM-2.0 models.

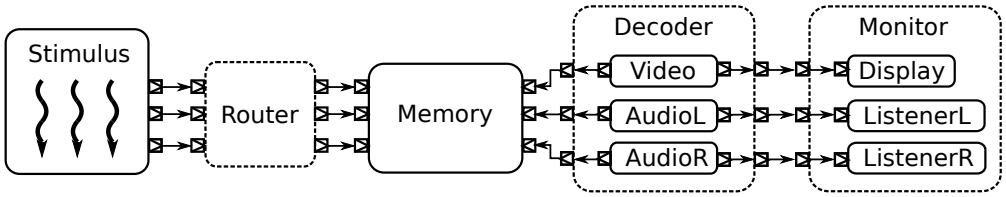


Figure 3.10: Block diagram for the DVD Player example

Table 3.5: Results of DVD Player: run-time (secs) and speedup (%)

Interface	Direct			Hierarchical			Interconnect		
	Seq	Par		Seq	Par		Seq	Par	
BTI	208.1	73.8	282%	208.1	75.7	274%	208.4	74.8	278%
DMI	208.2	73.7	282%	208.5	75.5	276%	208.4	7.47	279%
NBTI	209.3	74.9	279%	209.4	75.6	277%	209.5	7.57	277%

3.8.3 Mandelbrot Renderer

The Mandelbrot renderer is a parallel image rendering application to compute the Mandelbrot set. The platform architecture is shown in Figure 3.11. The `DUT` module hosts eight parallel renderer threads. Each renderer thread computes a Mandelbrot image in a given area and sends the result to the controller thread of `DUT`. The controller merges all the results and sends it out, then starts to wait for new frames. Again, three experiments are performed on the the Mandelbrot Renderer example with different communication types: BTI, DMI and NBTI. The results of run-time and speedup comparing to sequential simulation are shown in Table 3.6.

As shown in the results, the speedup between OoO PDES and sequential simulation under both communication types are around 420%. The naive maximum speedup of 800% is not achieved because there are only 4 floating point units (FPU) in the processor and thus the eight computationally intensive renderer threads are not able to run totally in parallel [3]. Considering the restriction of hyperthreading, the 420% speedup is impressive on a 4 core machine. The results of this experiment

[3]In the BTI example, the user time of sequential simulation is 77.59 seconds which is smaller than 92.04 seconds for parallel simulation. This shows that each hyperthread during parallel simulation is running longer due to the contention in the FPU

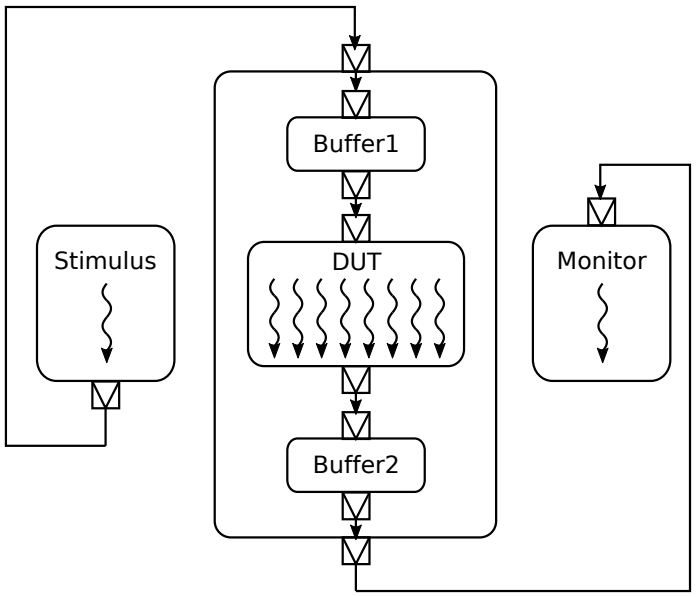


Figure 3.11: Block diagram for the MandelBrot Renderer example

demonstrate again that the proposed static analysis is effective and correct to support OoO PDES of SystemC TLM-2.0 models using BTI, DMI and NBTI.

Table 3.6: Results of Mandelbrot Renderer: run-time (secs) and speedup (%)

Interface	Seq	Par	Speedup
BTI	77.59	18.66	416%
DMI	77.64	18.45	421%
NBTI	77.60	18.50	419%

Table 3.7: Results of Bitcoin miner: run-time (secs) and speedup (%)

	BTI			DMI			NBTI		
Scanners	Seq	Par		Seq	Par		Seq	Par	
1	1903.02	1902.17	100%	1902.64	1908.29	100%	1889.93	1899.13	99%
2	1680.54	858.06	196%	1675.68	855.61	196%	1677.73	850.95	197%
4	1885.32	508.24	371%	1890.30	507.47	372%	1895.60	506.56	373%
8	1944.04	424.02	458%	1948.76	420.77	464%	1934.75	423.59	457%
16	2282.64	506.33	450%	2274.40	507.76	448%	2283.62	498.07	458%
64	3169.93	696.75	455%	3161.10	698.49	452%	3169.84	688.85	460%
256	6827.77	2315.08	295%	6824.44	2056.17	332%	6818.38	2236.09	305%

3.8.4 Bitcoin Miner

Our third real-world example is a SystemC TLM-2.0 model of a Bitcoin miner. Bitcoin miners create Bitcoins by solving math problems using a computation-intensive cryptographic hashing algorithm. Figure 3.12 shows the structure of the Bitcoin miner model. `Stimulus` prepares the math problems and sends them to `Dispatcher`. `Dispatcher` divides the received problems into multiple sub-problems and stores them into `Memory`. From there, a `Scanner` picks a corresponding sub-problem and tries to solve it using a hashing algorithm. Once done, the `Scanner` stores the result into `Memory` and waits for a new sub-problem. `Dispatcher` fetches the result from `Memory` and sends it to `Monitor`. Note that in the Bitcoin miner model all `Scanner`s are independent and their threads are able to run in parallel with our approach.

We perform experiments with different number of `Scanner`s: 1, 2, 4, 8, 16, 64, 256 and different

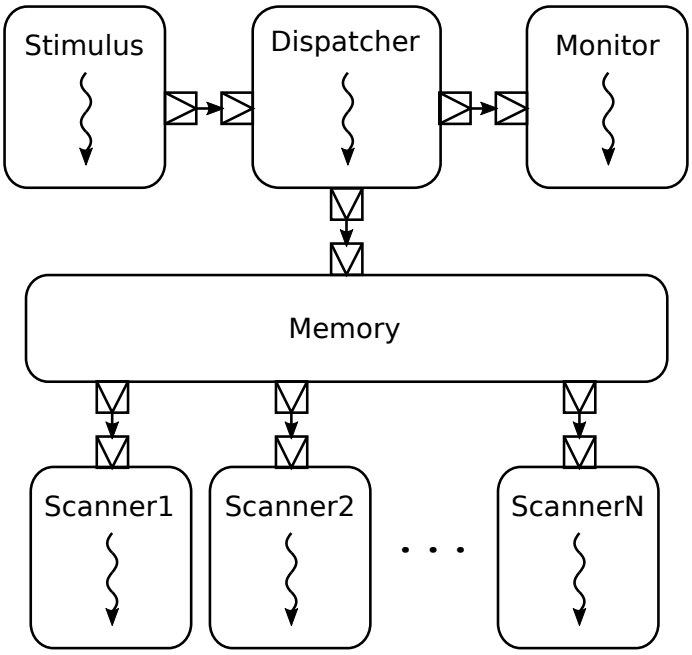


Figure 3.12: Block diagram for the Bitcoin miner example

communication types: BTI, DMI, NBTI. The results are shown in Table 3.7. Similar to previous experiments, the communication type does not significantly affect the simulation speeds. Because of the high parallelism level of Bitcoin miner, the speedup between OoO PDES and sequential simulation keeps increasing and reaches a maximum of around 460% when there are 8 `Scanner`s.

Although the host processor has eight hyperthreads, a naive speedup of 800% cannot be achieved due to the contention in the FPU. The speedup remains at the same level with more `Scanner`s and even drops when there are 256 `Scanner`s. This is because of the dramatically increased context switching between threads. The context switching is also a major factor that significantly affects the sequential and parallel simulation time when there are more than 8 scanners. Considering all the hardware restrictions, this experiment still demonstrates the effectiveness and correctness of our proposed static analysis for supporting OoOPDES of SystemC TLM-2.0 models using BTI, DMI and NBTI.

3.9 Conclusion

In this chapter, we propose a compiler-based approach to statically analyze SystemC TLM-2.0 loosely-timed (LT) and approximately-timed (AT) models. The analysis is essential to simulate the model under OoO PDES. In the proposed approach, an accurate SG is first built with SCP information for TLM-2.0 interface function calls. Then, the compiler analyzes the variable accesses for entangled variables, precisely identifying potential conflicts. Our experiments demonstrate the correctness and effectiveness of the approach with demonstration examples from Accellera and three real world examples: DVD Player, Mandelbrot Renderer and Bitcoin Miner.

Chapter 4

Improving Coding Guidelines For Faster OoO PDES

In this chapter, we propose for the RISC users coding guidelines that decrease the granularity of segments [64], so that the level of parallelism in the design increases and higher simulation speed becomes possible.

4.1 Introduction

As described in Section 1.2.1, a Segment Graph (SG) is fundamental for both static analysis in the RISC compiler and dynamic checking in the OoO PDES library. On one hand, in the RISC compiler, the data, event and time conflicts are analyzed at segment level. On the other hand, in the OoO PDES simulator, a segment is the basic unit that is been executed by each simulation thread. A segment in a SG contains a sequence of code statements that is supposed to run in the same simulation cycle. Since the conflicts are analyzed at segment level, once two segments have conflict, the code statements that they contain must no run in parallel. Correspondingly, the granularity of

a SG has a high impact on the conflict analysis and the parallelism level of simulation.

In this chapter, we will study the impact of the SG granularity on the simulation speed. We propose coding guidelines for SystemC users to build models with higher parallel potential that can be executed faster by the OoO PDES simulator. This is different from coding guidelines that focus more on code readability and maintainability [79, 80, 81]. Specifically, the guidelines suggest for users to insert extra **`wait`** statements into the model, so as to increase the granularity of the SG. With the finer granularity SG, variable and event conflicts can be constrained into shorter segments, thereby reducing the time of sequential execution, which is necessary, for example, during communication between modules in the system.

Our contributions in this chapter [64] are summarized as follows:

1: We propose a formal metric ψ to estimate the level of parallelism of the model under OoO PDES.

2: We propose coding guidelines for the SystemC model designers to optimize the model for faster simulation.

3: We demonstrate that the proposed coding guidelines enable significant speedup of OoO PDES.

4.2 Proposed Coding Guideline

In this section, we propose a new coding guideline for the SystemC model designers to write SystemC models with higher parallel simulation potential. Before describing the guideline, we first define a metric to estimate the level of parallelism of a SystemC model under OoO PDES.

4.2.1 SG Granularity and Simulation Speed

In OoO PDES, models are simulated at segment level. As shown in Figure 4.1, module `M` has two sc_threads `th1` and `th2`, and a member variable `a`. `f()` and `g()` are data crunching functions which

work on local variables. The corresponding SG is shown in Figure 4.2. Due to the data hazard over a, the two segments are not allowed to run in parallel. More details about data conflicts are described in Section 1.2.3. Figure 4.3 shows the scheduling of execution of the two sc_threads.

```
 1  SC_MODEULE(M){
 2      ...
 3      int a;
 4      void th1(){
 5          a=1;
 6          f();
 7      }
 8
 9      void th2(){
10          g();
11          a=2;
12      }
13      ...
14  }
```

Figure 4.1: Coarse Grained Source Code

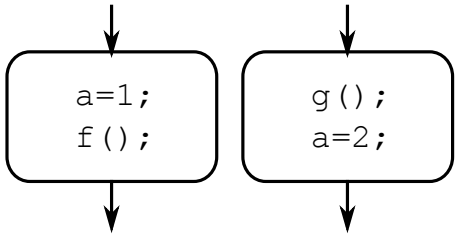


Figure 4.2: SG of Fig. 4.1

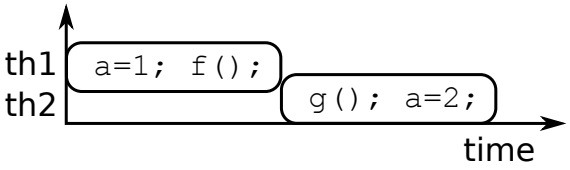


Figure 4.3: Scheduling of Fig. 4.1

By inserting two new **wait** statements into the sc_threads, as shown in Figure 4.4, the SG becomes Figure 4.5. In this model, functions f() and g() are no longer in the same segment of the statements that access the shared variable a. Because f() and g() are conflict free, they can now be executed in parallel as shown in Figure 4.6, which significantly speeds up the simulation.

This leads to the conclusion that by increasing the granularity of SG, more code statements can run in parallel, and consequently increase the level of parallelism of a model and further speedup the simulation. In the following sections, we will show more details to confirm this idea and propose a coding guideline for the model designer to increase the parallel potential of the SystemC models under OoO PDES.

```
1  SC_MODEULE(M){
2      ...
3      int a;
4      void th1(){
5          a=1;
6          wait(
       SC_ZERO_TIME);
7          f();
8      }
9
10     void th2(){
11         g();
12         wait(
       SC_ZERO_TIME);
13         a=2;
14     }
15     ...
16 }
```

Figure 4.4: Fine Grained Source Code

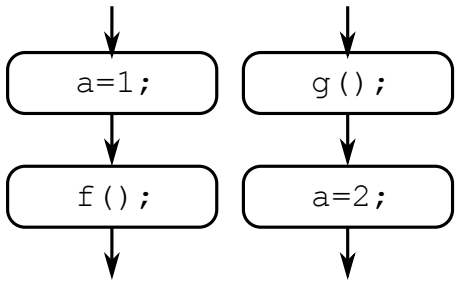


Figure 4.5: SG of Fig. 4.4

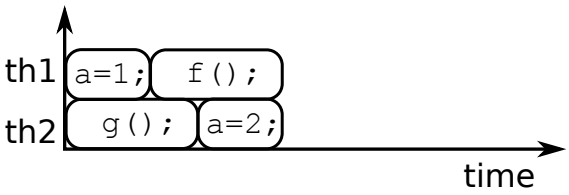


Figure 4.6: Scheduling of Fig. 4.4

4.2.2 Estimation for Level of Parallelism

The level of parallelism ψ is estimated as the amount of code statement pairs that can potentially execute in parallel. In OoO PDES, only code statements that belong to conflict-free segments can run in parallel, and hence our estimation is expressed as:

$$\psi = \sum_{i} \sum_{\substack{j>i \\ \text{th}_i \neq \text{th}_j}} \textsc{HasNoConflict}(\text{seg}_i, \text{seg}_j) \tag{4.1}$$

Where i and j are the index of code statements in the model. seg_n is the segment that includes the n^{th} code statement. And similarly, th_i is the thread that executes the n^{th} code statement. Each single thread executes sequentially, and code statement i and j cannot execute in parallel if they belong to the same thread. HasNoConflict(seg_i,seg_j) returns 1 if seg_i and seg_j are conflict free, otherwise it returns 0.

If two segments are in conflict, then any pair of code statements that belong to the two segments are not allowed to execute in parallel, which would reduce ψ. Thus, the larger ψ is, the higher is the parallelism level of the input model.

4.2.3 Motivation

Our idea is motivated by the following observation:

Consider we have two segments: seg_1 and seg_2, which are executed by two different threads. There are respectively p and q statements in seg_1 and seg_2. ψ for this model is simply $\psi_1 = p \times q \times \textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2)$.

Now, if a **`wait`** statement is inserted into seg_1, such that seg_1 is partitioned into two non-overlapping segments: seg_{11} and seg_{12}. After the partitioning, seg_{11} includes the first p_1 statements of seg_1, and seg_{12} includes the other $p_2 = p - p_1$ statements of seg_1. ψ for the new model becomes $\psi_2 = p_1 \times q \times \textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) + p_2 \times q \times \textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2)$. seg_{11} and seg_{12} are executed by the same thread, and hence they must run sequentially and ψ_2 does not increase.

When comparing ψ_1 and ψ_2, we get four different scenarios:

1. The conflict between seg_1 and seg_2 is only incurred by certain statements in the first p_1 statements of seg_1, and the last p_2 statements are conflict free. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 0$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 1$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this scenario, $\psi_1 = 0$ and $\psi_2 = p_2 \times q$. ψ_2 is larger than ψ_1.

2. The conflict between seg_1 and seg_2 is only incurred by certain statements in the last p_2 statements of seg_1, and the other p_1 statements are conflict free. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 1$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 0$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this scenario, $\psi_1 = 0$ and $\psi_2 = p_1 \times q$. ψ_2 is larger than ψ_1.

3. The conflict between seg_1 and seg_2 is incurred both by certain statements in the first p_1 statements and the other p_2 statements of seg_1. This indicates that $\textsc{HasNoConflict}(\text{seg}_{11}, \text{seg}_2) = 0$, $\textsc{HasNoConflict}(\text{seg}_{12}, \text{seg}_2) = 0$ and $\textsc{HasNoConflict}(\text{seg}_1, \text{seg}_2) = 0$. Under this sce-

nario, $\psi_1 = 0$ and $\psi_2 = 0$. ψ_2 is equal to ψ_1.

4. seg_1 and seg_2 are conflict free. This indicates that HasNoConflict$(seg_{11}, seg_2) = 1$, HasNoConflict$(seg_{12}, seg_2) = 1$ and HasNoConflict$(seg_1, seg_2) = 1$. Under this scenario, $\psi_1 = p \times q$ and $\psi_2 = p_1 \times q + p_2 \times q = p \times q$. ψ_2 is equal to ψ_1.

The four scenarios suggest that

1. Partitioning a segment does not decrease the parallel potential of a model.

2. If the user carefully selects the place to insert the extra segment boundary, i.e., **`wait`** statement, ψ can be increased significantly and results in a model with higher parallelism level.

4.2.4 Overhead Consideration

One may deduce that it is always beneficial to insert as many extra **`wait`** statements as possible, because by doing this the ψ of the model keeps increasing. Although the deduction is correct, it is not a good practice.

Each extra **`wait`** statement will increase the number of segments in the SG by one. And the size of conflict tables is to the square of the segment count. Thus, if too many extra **`wait`** statements are inserted, the time cost for static analysis and dynamic checking will grow dramatically, which would rather decrease the simulation performance. Besides, too many extra **`wait`** statements may also make the model incomprehensible.

Last but not least, each new **`wait`** statement creates an extra scheduler entry point into the simulator kernel which incurs high overhead.

4.2.5 Suggestions

Motivated by the above observations and considerations, we propose the following suggestions for the SystemC model designers to properly place extra **wait** statements in the source code, so as to increase the parallel potential of the model under OoO PDES.

Use the Wait-for-delta-cycle Primitive as the Extra Segment Boundary

There are six different kinds of **wait** primitives in the SystemC standard [27]:

1. **wait**() : Wait for the sensitivity list event to occur.
2. **wait**(int) : Wait for n clock cycles in SC_CTHREAD.
3. **wait**(event) : Wait for the event mentioned as parameter to occur.
4. **wait**(double,sc_time_unit) : Wait for specified time.
5. **wait**(double,sc_time_unit, event) : Wait for specified time or event to occur.
6. **wait**(SC_ZERO_TIME): Wait for one delta cycle.

The event related **wait** primitives shall not be used because they require proper events to be notified. For the wait-for-time primitive, it is likely to change the simulation time cycle, which is not desirable. Thus, in order to maintain the semantics and timing accuracy of the original SystemC model, we suggest to the designers to use wait-for-delta-cycle primitive, i.e., **wait**(SC_ZERO_TIME) as extra segment boundaries. [1]

[1]Note that the timing accuracy of a robust model will not be affected by extra delta cycles.

```
1  SC_MODEULE(M)
2  {
3      int c;
4      void th1()
5      {
6          int x=1;
7          wait(10,SC_NS);
8          c=42;
9
10         for(int i=0;
               i<100;
11             i++) x++;
12     }
13     void th2()
14     {
15         int y=100;
16         wait(1,SC_NS);
17         c=0;
18
19         for(int j=0;
               j<100;
20             j++) y--;
21     }
22     ...
23 }
```

Figure 4.7: Source Code for Module M

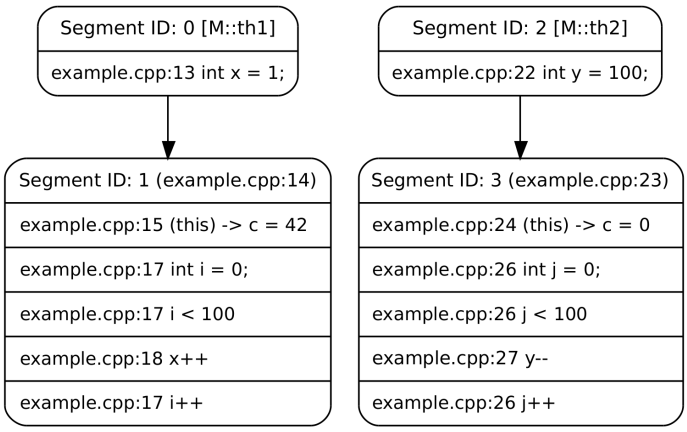


Figure 4.8: SG for Figure 4.7

(Seg,Inst)	(0,0)	(1,0)	(2,0)	(3,0)
(0,0)				
(1,0)		M::c, 0		M::c, 0
(2,0)				
(3,0)		M::c, 0		M::c, 0

Figure 4.9: DCT for Figure 4.7

Partition the Heavy Segments

As mentioned in Section 4.2.4, the cost for one extra **`wait`** statement is independent of where it is inserted. Thus, in order to maximize the gain of ψ of the model, we suggest the users to partition computational intensive segments, which we refer to as *heavy segments*.

Unfortunately, it is not obvious to identify heavy segments directly from the model code. However, the RISC compiler is able to dump the statically generated SG and the DCT into files by turning on the **`-risc:dump`** command line option. The SG is then dumped into a **`.dot`** file which can be viewed graphically using the **`xdot.py`** tool. Also, the DCT is dumped into an HTML file which the designer can easily view in any browser. An example SystemC source code is shown in Figure 4.7. The dumped SG and DCT are shown in Figure 4.8 and Figure 4.9. The level of parallelism ψ for this model is $\psi 1 = 6 + 5 = 11$

```
1  SC_MODEULE(M)
2  {
3      int c;
4      void th1()
5      {
6          int x=1;
7          wait(10,SC_NS);
8          c=42;
9          wait(SC_ZERO_TIME);
10         for(int i=0;
               i<100;
11             i++) x++;
12     }
13     void th2()
14     {
15         int y=100;
16         wait(1,SC_NS);
17         c=0;
18         wait(SC_ZERO_TIME);
19         for(int j=0;
               j<100;
20             j++) y--;
21     }
22     ...
23 }
```

Figure 4.10: Source Code for Module M after partitioning

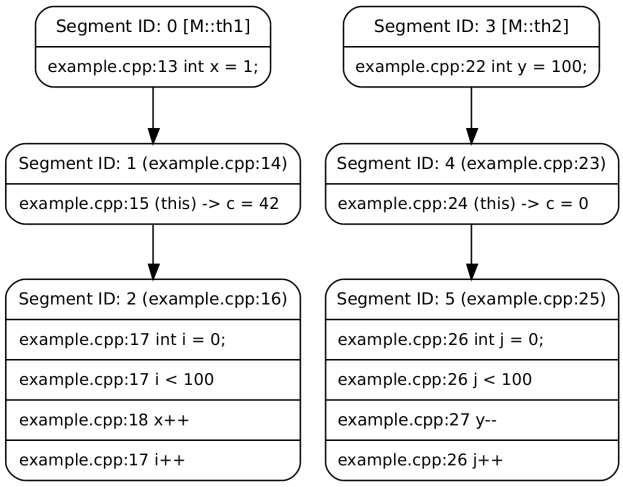


Figure 4.11: SG for Figure 4.10

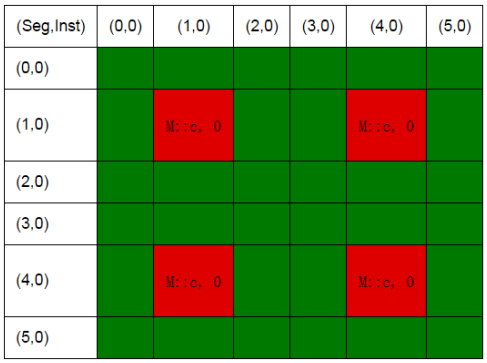


Figure 4.12: DCT for Figure 4.10

From the SG, it is apparent that segment 1 and segment 3 are heavy segments which both contain loops. In order to increase the parallelism level of the model, we wish to partition the conflict-free statements from the conflicting ones in the segment, as described in the first and second scenarios in the previous section. To locate the conflicting statement, the user can refer to the dumped Data Conflict Table. In the ((1,0),(3,0)) entry of the table, it shows that the data conflict is over the variable **`M::c`**, and so the conflict is between statement line 8 and 17 in Figure 4.7[2]. In this example, the conflicting statements are not inside the computationally intensive code pieces, that are, the for loops. So we can partition the segments by inserting **`wait`** statements after line 9 and 18. The optimized model is shown in Figure 4.10. The dumped SG and DCT are shown in Figure 4.11 and Figure 4.12. Now, the level of parallelism ψ becomes $\psi 2 = 6 + 5 + 4 \times 6 = 35$. The

[2]The instance id is shown here, which is not of interest in this chapter

parallel potential is further intensified during the simulation due to the two conflict free **for** loops.

4.3 Experiments and Results

We have applied the proposed coding guideline to several SystemC model examples. We first tested it on the synthetic benchmarks generated by the TGFF tool to validate the effectiveness of our coding guideline. Then, we evaluate the guideline with two real world designs, Canny Edge Detector and Audio/Video Decoder, to demonstrate the performance. The experiments are performed on an Intel E3-1240 host machine, which has a total of 8 cores (4 cores with 2-way hyperthreading each). The CPU frequency scaling is turned off so as to obtain repeatable results.

4.3.1 TGFF benchmarks

We first examine the performance of the proposed coding guideline on a synthetic benchmark, which is automatically generated by the TGFF tool with SystemC extension [58]. Figure 4.13 shows the data flow block diagram of the generated model. It has a source and a sink, and multiple parallel lanes of nodes in between. Figure 4.14 shows the source code for each node. Each node module first gets an input from a channel, and then does data crunching which is computationally intensive. The data crunching accesses only local variables and thus is conflict-free. After the computation the module outputs the result to another channel. In such model, data conflicts are incurred only by channel communications, which are caused by the parallel accesses to the shared variables in the channels. To optimize the model, we apply the proposed coding guideline and put **wait**(SC_ZERO_TIME) statements around the data crunching parts. The source code for the optimized module is shown in Figure 4.15.

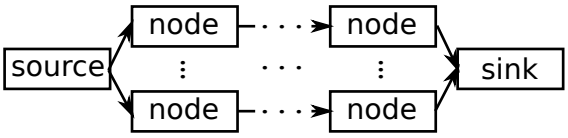


Figure 4.13: Block Diagram of TGFF Models

```
1  SC_MODEULE(Node)
2  {
3      sc_port<input> in;
4      sc_port<output> out;
5      void th1()
6      {
7          int a = in.read();
8          for(int i=0;
9              i<WORKLOAD;
10             i++) a++;
11         out.write(a);
12     }
13     ...
14 }
```

Figure 4.14: Original Source Code of Generated Testbench Model

```
1  SC_MODEULE(Node)
2  {
3      sc_port<input> in;
4      sc_port<output> out;
5      void th1()
6      {
7          int a = in.read();
8          wait(
           SC_ZERO_TIME);
9          for(int i=0;
10             i<WORKLOAD;
11             i++) a++;
12         wait(
           SC_ZERO_TIME);
13         out.write(a);
14     }
15     ...
16 }
```

Figure 4.15: Optimized Source Code of Generated Testbench Model

Table 4.1: Performance of TGFF Benchmarks, Simulator run times (sec) and CPU utilization

Benchmark	SEQ	PAR	GDL
1	63.55 (99%)	17.85 (377%)	10.48 (690%)
2	63.54 (99%)	17.63 (379%)	10.91 (663%)
3	134.41 (99%)	88.41 (155%)	81.55 (172%)
4	349.86 (99%)	165.41 (214%)	93.44 (400%)
5	493.02 (99%)	169.12 (301%)	99.17 (552%)
6	134.40 (99%)	92.00 (155%)	81.10 (173%)
average	206.46 (99%)	91.74 (263.5%)	62.77 (441%)

Through a parameter to the TGFF generator, we are able to control the total number of lanes as well as nodes per lane, and each lane may consist of various number of nodes. The data crunching workload of each node is controlled by the number of iterations of the **`for`** loop.

We studied 6 test cases with different data flow configurations in this experiment. Table 4.1 shows

the performance of the simulations before and after applying the coding guideline. The first column SEQ refers to the sequential simulation with the reference Accellera SystemC simulator. Under the sequential simulation, the CPU utilization is always below 100% because only one thread is running at any time during the simulation. The second column PAR refers to the OoO PDES before applying the coding guideline. It shows that on average, the simulation of the original models is 2.3x faster than SEQ. The third column GDL refers to the OoO PDES after applying the coding guideline. It is 3.2x faster than SEQ, and 1.4x faster than PAR. For the first benchmark, GDL achieved a maximum speedup of 1.7x over PAR, and the latter one is 3.5x faster than SEQ. Note that the CPU utilization is larger than the speedup over SEQ. This is because in OoO PDES there is some overhead for checking conflict tables. The results confirm that our coding guideline can be very effective in achieving higher speedup under OoO PDES.

4.3.2 Real world examples

We then evaluate the proposed coding guideline with two real world examples, namely Canny Edge Detector and Audio/Video Decoder modeled similarly to the benchmarks used in [58] and [40].

Canny Edge Detector

Our first real world example is the Canny edge detector, which filters edges in an image. The edge detector is a structurally five-stage pipeline, and each stage has a communication-computation-communication code structure. Communication between two pipeline stages is via a user-defined channel in which the read and write functions access the shared channel variable. In this experiment, a sequence of 20 images is fed into the pipeline and correspondingly generates 20 outputs. The outputs are verified to ensure a correct simulation.

Table 4.2 shows the simulation time and CPU utilization before and after applying the coding guideline. By using the original model, a CPU utilization of 127% is achieved, which is due to

Table 4.2: Performance of Canny Edge Detector

	SEQ	PAR	GDL
simulation time (sec)	248.51	199.62	172.33
CPU utilization	100%	127%	149%
speedup	1.00	1.24	1.44

the conflicts among communications. With the optimized model, the CPU utilization is increased to 149%, and the OoO PDES speed is increased by 1.2x. The speedup is not as impressive as in the TGFF test cases. This is because the workload of each pipeline stage varies greatly, and the bottleneck of the simulation speed is determined by the longest stage. However, this experiment still confirms the effectiveness of the proposed coding guideline.

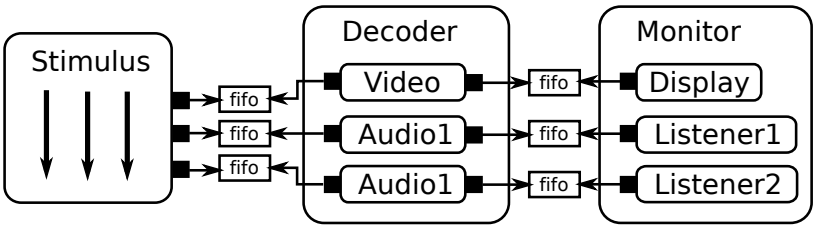


Figure 4.16: Block Diagram of Audio/Video Decoder

A/V Decoder

The second real world test case is an Audio/Video decoder. The model structure is shown in Figure 4.16. The stimulus sends the encoded stream to one video decoder and the left and right audio decoders. Then, the video decoder outputs the result to a monitor, and the audio decoders output the results to two speakers. The results for this test case are shown in Table 4.3. The execution times cost for OoO PDES before and after applying the coding guideline are 48.24 secs and 26.67 secs, which suggest the optimized model executes 1.8x faster. The speedup is reasonable because the encoding and decoding stages have similar computation loads. The result again confirms the effectiveness of the proposed coding guideline.

Table 4.3: Performance of Audio/Video Decoder

	SEQ	PAR	GDL
simulation time (sec)	73.41	48.24	26.67
CPU utilization	100%	152%	247%
speedup	1.00	1.52	2.75

4.4 Conclusion

In this chapter, we proposed a coding guideline for the SystemC model designers who use OoO PDES parallel execution enabled by the Recoding Infrastructure for SystemC. By applying the coding guideline, the granularity of the Segment Graph becomes larger, and thus results in a faster execution speed. Our experiments show that by applying the proposed coding guideline, the optimized SystemC model is able to achieve a speedup of up to 1.7x on a 8 core machine, on top of the 3.5x speedup due to PDES.

Chapter 5

Optimizing Event Processing in Out-of-Order Parallel Simulation

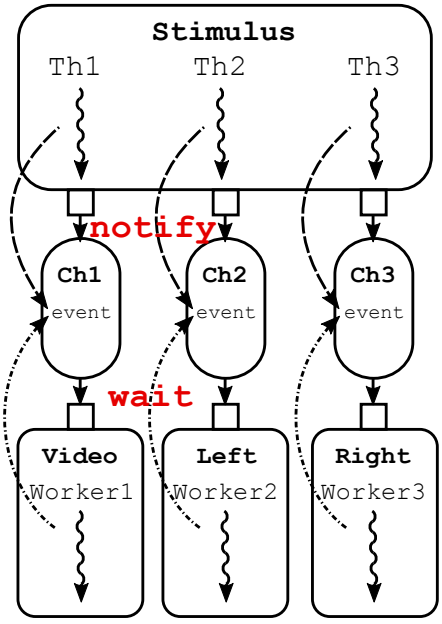
Besides the researches on static analysis on the RISC compiler side, we also investigate approaches to improve the OoO PDES library [65], specifically, the central scheduler. In this chapter, we introduce a novel event delivery strategy that allows waiting threads to resume execution earlier, resulting in significantly increased simulation speed.

5.1 Introduction

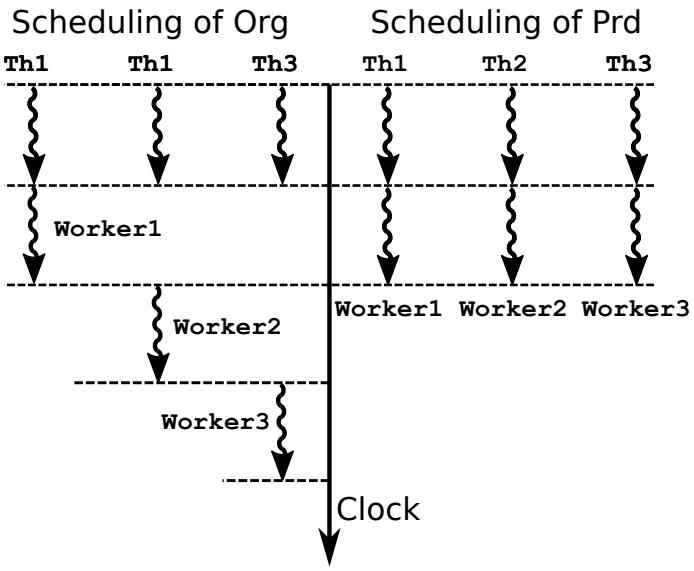
As described in Section 1.1.3, Out-of-Order Parallel Discrete Event Simulation (OoO PDES) [34] is studied to increase the multi-core CPU utilization. Compared to traditional Parallel Discrete Event Simulation (PDES), OoO PDES has a higher parallelism level as it allows threads to run in parallel even if they are in different cycles. Two techniques, namely static analysis and dynamic checking, are performed to preserve the simulation semantics and timing accuracy of OoO PDES. One bottleneck of current OoO PDES that limits the simulation speed is its event delivery strategy.

The OoO PDES scheduler is conservative and only delivers the earliest event notifications on every scheduling step. This limits the number of threads to be waked up and and reduces the parallelism level of simulation.

To resolve this limitation, in this chapter we propose a prediction-based event delivery algorithm. The new approach looks ahead in time to predict the earliest possible wake-up time of each waiting thread. This allows the OoO PDES scheduler to make aggressive but safe thread dispatching decisions. The approach relies on the statically generated event notification table with prediction (ETP) by the RISC compiler, as introduced in [41].



(a) SystemC model of a DVD player



(b) Scheduling of Org and Prd

Figure 5.1: SystemC model and OoO PDES scheduling

As a motivating example, Figure 5.1a shows a simplified high-level SystemC model of a DVD player. The `Stimulus` has three parallel threads and each sends data to a corresponding channel. When the data is sent, an event is notified inside the channel for synchronization purpose. `Video`, `Left` and `Right` are three decoder modules and each has a single worker thread that waits for the corresponding channel event. When the event is delivered, the worker thread wakes up and starts processing the received data.

Figure 5.1b shows the scheduling of threads under the original event delivery strategy (Org) and the

optimized event delivery strategy using prediction (Prd) proposed in this chapter. Under Org, only one worker thread can wake up each scheduling step due to the in-order event delivery strategy. While under Prd, the scheduler is able to get more context information provided by the predictions of future thread behaviors. More events are delivered every scheduling step and more threads are allowed to wake up in parallel. As a result, the simulation speed is increased significantly.

5.2 Related Work

[58] proposes a scheduling algorithm that predicted thread run time at segment level for better multi-core scheduling. [60] exploits data-level parallelization on top of OoO PDES for faster SystemC simulation. [41] introduces a static approach to predict future behaviors of threads, and used the information for advanced data conflict analysis of threads. Although our work reuses ETP proposed in [41], it is totally different from [41] because in our work the prediction information is used for optimized event delivery strategy. Note also that the approaches in [58], [60] and [41] are orthogonal with ours and can be applied together for parallel SystemC simulation. Other works on PDES are discussed in Section 1.3.

5.3 Background

SG is the data structure that is fundamental to this research, which is briefly described in Section 1.2.1. In this section, we review the ETP data structure that provides the prediction information about event notifications and the original event delivery strategy.

5.3.1 Event Notification Table with Prediction

ETP was first introduced in [41] for optimized data conflict analysis in OoO PDES, which is a table that stores the prediction information about the time advance for a segment to wake up another segment. ETP is formally defined in Equation 5.1. It is automatically built by the compiler with the algorithm proposed in [41].

$$ETP[i,j] = \begin{cases} (t_\Delta, \delta_\Delta) & \text{if a thread in } seg_i \text{ may wake up a thread in } seg_j \text{ with least time advance of } (t_\Delta, \delta_\Delta) \\ (\infty, 0) & \text{if a thread in } seg_i \text{ will never wake up a thread in } seg_j \end{cases} \tag{5.1}$$

Take the SystemC code in Figure 5.2a as an example. Its SG is shown in Figure 5.2b. Segment 4 is directly waked up by segment 2 via event `e1`, ETP[2, 4] is thus one delta cycle, denoted as (0,1) in this chapter[1]. Indirect event notifications are also considered in ETP. Segment 1 does not notify any event, however, it is followed by segment 2 with time advance of `1 SC_NS` and segment 2 directly wakes up segment 4. Therefore, segment 1 indirectly wakes up segment 4 with a minimum time advance of (1,1). Segment 1 can also indirectly wake up segment 6 by first indirectly wakes up segment 4, then segment 4 directly wakes up segment 6. In this case, EPT[1,6] = (1,2). The corresponding ETP is shown in Figure 5.3. Note that there are several (∞,0) entries in the table. For instance, ETP(2,3) is (∞,0). This is because segment 3 does not wait for any event and thus no other segment may wake it up.

In our approach, ETP is used to calculate the predicted wake-up time of a waiting thread by another thread. Take Figure 5.2b as an example. Suppose `th3` is waiting and `th1` is running at timestamp (0,0) in segment 1. According to ETP, the scheduler predicts that thread 3 will probably wake up

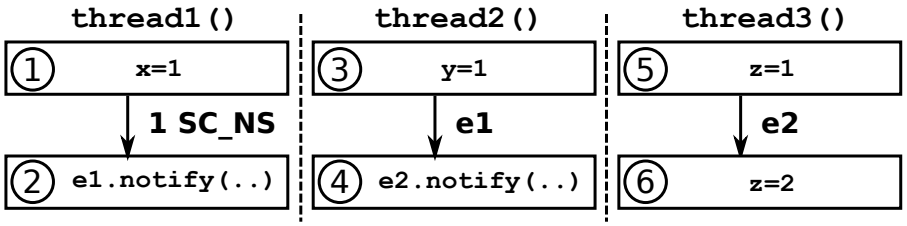
[1]the first element in the tuple is the time count and the second one is the delta count

```
1 void thread1(){
2   x = 1;
3   wait(1,SC_NS);
4   e1.notify(
    SC_ZERO_TIME);
5 }
```

```
6 void thread2(){
7   y = 1;
8   wait(e1);
9   e2.notify(
    SC_ZERO_TIME);
10 }
```

```
11 void thread3(){
12   z = 1;
13   wait(e2);
14   z = 2;
15 }
```

(a) SystemC Source code



(b) SG of 5.2a

Figure 5.2: Example of SG

at timestamp (0,0) + (1,2) = (1,2).

Seg	1	2	3	4	5	6
1	∞,0	∞,0	∞,0	**1,1**	∞,0	**1,2**
2	∞,0	∞,0	∞,0	**0,1**	∞,0	**0,2**
3	∞,0	∞,0	∞,0	∞,0	∞,0	**0,2**
4	∞,0	∞,0	∞,0	∞,0	∞,0	**0,1**
5	∞,0	∞,0	∞,0	∞,0	∞,0	∞,0
6	∞,0	∞,0	∞,0	∞,0	∞,0	∞,0

Figure 5.3: ETP for 5.2b

5.3.2 Original Event Delivery Strategy

We now briefly introduce the original event delivery strategy and discuss its limitations.

In OoO PDES, an event notification is not delivered immediately when notified. Instead, it is first

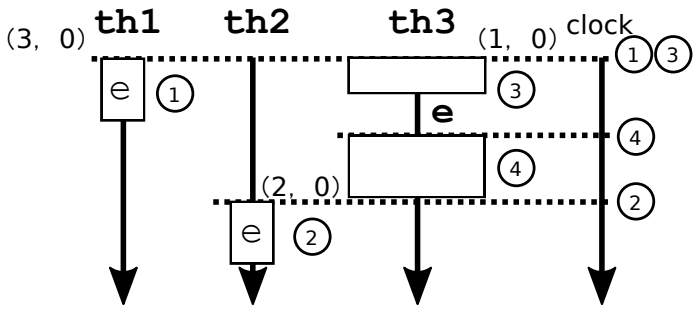
stored into an *event notification set* Σ that keeps all event notifications during the simulation. Then, on every scheduling step, the scheduler checks every event notification in Σ to determine if the event notification satisfies the following two requirments:

1. Earlier than all **RUN** or **READY** threads

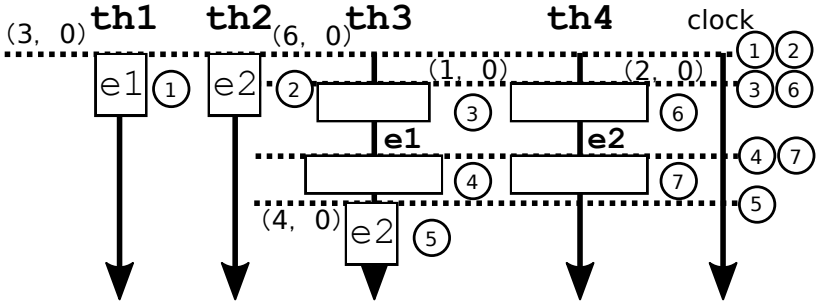
2. Earliest among all the event notifications stored in Σ that can wake up threads

If the two requirements are fulfilled, the scheduler delivers the event notification and wakes up all the threads that are waiting on this event.
The two requirements are demanded to avoid potential *late wake-up* of threads due to unknown future behaviors of other threads. Two possible scenarios are shown in Figure 5.4. The rectangles represent the segments executed by threads. Note that the clock axis represents the wall clock time. The scenario in Figure 5.4a explains requirement 1. Segments 1 and 2 of threads `th1` and `th2` notify event `e`. Segment 4 of thread `th3` waits for `e`. We assume that under Out-of-Order execution, `th1` and `th3` are scheduled to run first. Segment 1 notifies `e` at timestamp (3,0). Next, `th3` starts waiting for `e` at (1,0). Although `e` has already been notified at (3,0), the scheduler decides not to deliver it to wake up `th3`. This is because `th2` is still in **READY** at (2,0) and the scheduler cannot predict if `th2` notifies `e` before (3,0). In this example, `th2` does notify `e` at (2,0) and therefore `th3` should wake up at (2,1). Requirement 1 successfully prevents a late wake-up of `th3` at (3,1).

The scenario in Figure 5.4b explains requirement 2. Two events `e1` and `e2` are notified by threads `th1` and `th2` at timestamps (3,0) and (6,0). By requirement 1, the two event notifications are not delivered because threads `th3` and `th4` are still in **READY** at earlier timestamps. Next, `th3` and `th4` are scheduled to run and then wait for `e1` at (1,0) and `e2` at (2,0) respectively. Now, both event notifications: e1 at (3,0) and e2 at (6,0) meet requirement 1 as there are no threads in **RUN** or **READY**. However, as specified by requirement 2, only the earliest notification: e1 at (3,0) can be delivered. Notification e2 at (6,0) is not delivered. This is because the simulator is not able to



(a) Scenario 1



(b) Scenario 2

Figure 5.4: Scenarios explaining requirement 1 and 2

predict whether the thread waked up by `e1` will notify `e2` at an earlier timestamp before (6,0). In this example, `th3` notifies `e2` at (4,0) in segment 5 after being waked up by `e1`, and therefore `th4` should wake up at (4,1). Requirement 2 successfully prevents a late wake-up of `th4` at (6, 1).

The original event delivery strategy is very conservative and sometimes makes false decisions to not deliver an event notification. In scenario 1, if segment 2 is modified to not notify event `e`, then notification of `e` at (3,0) is actually safe to deliver immediately after `th3` starts waiting because no earlier notification of `e` will happen in the future. However, forced by requirement 1, the scheduler needs to wait until segment 2 finished execution to make this delivery. Similar for requirement 2. If the scheduler knows what will happen in the future, an event notification may be delivered earlier instead of being held until fulfilling requirements 1 and 2. Consequently, the corresponding waiting threads resume execution earlier and result in faster simulation. This motivates our idea of optimizing the event delivery strategy in OoO PDES with prediction information.

5.4 Optimized Event Delivery Strategy With Prediction

In this section, we first propose the optimized event delivery algorithm and then discuss about the optimization of Σ to reduce the space and time complexity.

5.4.1 Optimized Event Delivery Algorithm

Due to the Out-of-Order execution, a waiting thread th may be waked up by 1) event notifications that are already notified and stored in Σ or 2) event notifications that will be notified in the future by current running, ready or waiting threads. Based on this observation, we first use ETP to predict the earliest timestamp a waiting thread can wake up, and then use this information to determine if an event notification can be delivered to wake up a waiting thread. The details are shown in Algorithm 3. Note that the earliest timestamp a waiting thread th can wake up is denoted as $th.\tau$ in Algorithm 3.

The algorithm contains four steps. In the first three steps, it calculates $th.\tau$ of each waiting thread th:

1. In lines 4-8, we initialize $th.\tau$ with the earliest timestamp at which th is waked up by event notifications in Σ.

2. In lines 11-21, we update $th.\tau$ by *predicting* the earliest timestamp at which a running or ready thread th_r directly/indirectly wakes up th. Specifically, EPT is looked up according to the segment IDs of th_r and th. As discussed in Section 5.3.1, the look-up result is the minimum timestamp advance for th_r to wake up th. By adding this result to the timestamp of th_r, we get the earliest predicted wake up timestamp of th by th_r.

3. In lines 24-38, we update $th.\tau$ by *predicting* the earliest timestamp at which another waiting

Algorithm 3 Optimized Event Delivery Strategy using Prediction

```
1: function AnalyzeAndDeliverEvents
2:     do
3:         ▷ Step 1: event notifications affect τ of waiting threads
4:         for all notification ∈ Σ do
5:             for all th ∈ getWaitingThreads(notification) do
6:                 th.τ ← min(th.τ, getTimestamp(notification))
7:             end for
8:         end for
9:
10:        ▷ Step 2: running and ready threads affect τ of waiting threads
11:        for all th ∈ WAIT do
12:            segID ← getSegmentID(th)
13:            for all th_r ∈ RUN ∪ READY do
14:                segID_r ← getSegmentID(th_r)
15:                t_r ← getTimestamp(th_r)
16:                t_pred ← ETP[segID_r, segID]
17:                if isValid(t_pred) then
18:                    th.τ ← min(th.τ, t_r + t_pred)
19:                end if
20:            end for
21:        end for
22:
23:        ▷ Step 3: waiting threads affect τ of other waiting threads
24:        L ← number of waiting threads
25:        𝔻 ← ∅                                   ▷ Waiting threads with determined τ
26:        for k ← 1 to L do
27:            th_k ← Waiting thread that has the k^th smallest τ
28:            insert th_k to 𝔻
29:            segID_k ← getSegmentID(th_k)
30:            t_k ← th_k.τ
31:            for all th ∈ WAIT do
32:                segID ← getSegmentID(th)
33:                t_pred ← ETP[segID_k, segID]
34:                if isValid(t_pred) then
35:                    th.τ ← min(th.τ, t_k + t_pred)
36:                end if
37:            end for
38:        end for
39:
40:        ▷ Step 4: Deliver event notifications by checking τs
41:        for all notification ∈ Σ do
42:            for all th ∈ getWaitingThreads(notification) do
43:                if getTimestamp(notification) = th.τ then
44:                    deliverNotificationToThread(notification, th)
45:                end if
46:            end for
47:        end for
48:    while no waked up thread and event notification delivered
49: end function
```

thread th_k directly/indirectly wakes up th. The prediction is done the similar way as in step 2: we first look up ETP and then add the result to the timestamp of th_k. However, th_k is still waiting and does not have a valid timestamp assigned until it has been waked up. Therefore, we instead use $th_k.\tau$ as the timestamp of th_k because $th_k.\tau$ is the earliest timestamp at which th_k can wake up. At first this seems like an endless loop: τs are used to update τs. In fact, there exists a topological order among all the τs: τ of a waiting thread may only be updated by τs of other waiting threads with smaller values. According to this topological order, we design a loop to update τs, as shown in lines 26-38. On the k^{th} iteration, we use $th_k.\tau$ of the waiting thread th_k that has the k^{th} smallest τ among all waiting threads to update other τs and stores th_k into the set $\mathbb{D}$.

Theorem 1. At the end of the k^{th} iteration, $\forall th_D \in \mathbb{D}, th_D.\tau \leq th_k.\tau$.

Proof. Theorem 1 is proved by induction.

a) *Base case*: At the end of the first iteration, $\mathbb{D}$ contains only th_1. It is correct that $\forall th_D \in \mathbb{D}$, $th_D.\tau \leq th_1.\tau$.

b) *Inductive hypothesis*: Let th_{last} be the last waiting thread added to $\mathbb{D}$. Let $\mathbb{D}' = \mathbb{D} \cup th_{last}$. Our I.H. is: $\forall th_{D'} \in \mathbb{D}', th_{D'}.\tau \leq th_{last}.\tau$

c) *Using the I.H.*: Assume I.H. is correct on the $(k-1)^{th}$ iteration. Let th_{k-1} be the waiting thread added to $\mathbb{D}$ on the $(k-1)^{th}$ iteration, th_k be the waiting thread added to $\mathbb{D}$ on the k^{th} iteration, $\mathbb{D}_{k-1}$ be the $\mathbb{D}$ at the end of the $(k-1)^{th}$ iteration, $\mathbb{D}_k$ be the $\mathbb{D}$ at the end of the k^{th} iteration. By using the I.H., $\forall th_{D_{k-1}} \in \mathbb{D}_{k-1}, th_{D_{k-1}}.\tau \leq th_{k-1}.\tau$. Also, since $\mathbb{D}$ only grows in size, $\mathbb{D}_k = \mathbb{D}_{k-1} \cup th_k$. Because on the $(k-1)^{th}$ iteration, we select th_{k-1} rather than th_k, this indicates that at the beginning of the $(k-1)^{th}$ iteration, $th_{k-1}.\tau \leq th_k.\tau$. Also, on the $(k-1)^{th}$ iteration we can only update other τs to be values larger than $th_{k-1}.\tau$ because ETP contains only positive timestamps. Therefore, $th_{k-1}.\tau \leq th_k.\tau$ still holds on the k^{th} iteration. Since 1) $th_{k-1}.\tau \leq th_k.\tau$, 2) using the I.H., $\forall th_{D_{k-1}} \in \mathbb{D}_{k-1}, th_{D_{k-1}}.\tau \leq th_{k-1}.\tau$, 3) $\mathbb{D}_k = \mathbb{D}_{k-1} \cup th_k$, by combining these inequalities we prove that $\forall th_{D_k} \in \mathbb{D}_k, th_{D_k}.\tau \leq th_k.\tau$. Therefore, the I.H. is still correct on the k^{th} iteration. □

According to Theorem 1, once a waiting thread th_k is inserted to $\mathbb{D}$, $th_k.\tau$ will not change in following iterations. Therefore, it is safe to use $th_k.\tau$ to update other τs.
In lines 41-47, Algorithm 3 checks for each waiting thread th if it can be waked up by an event notification $notification$ in Σ at $th.\tau$. If true, $notification$ is then safe to be delivered to wake up th because th is impossible to wake up any earlier.

The `do-while` loop in line 48 handles the situation where the only thread that can wake up is waiting for a `sc_event_and_list`. If this is not correctly handled, the simulation may stop early and violate the SystemC semantics. Details are not described in this chapter for brevity.
Now we demonstrate Algorithm 3 using the scenario in Figure 5.4a. When thread `th3` starts waiting, it cannot wake up immediately though event `e` is already notified at (3,0) in segment 1 by `th1`. This is because the scheduler predicts that segment 2 will in the future notify `e` and therefore `th3`.τ is (2,1). However, if the scenario is changed such that segment 2 does not notify `e` or notifies `e` after (3,0), notification of `e` at (3,0) would be delivered immediately when `th3` enters segment 3 because `th3`.τ is now (3,1). Similar for the scenario in Figure 5.4b. In conclusion, Algorithm 3 helps the scheduler to deliver event notifications earlier. As a result, waiting threads can resume execution earlier.

5.4.2 Complexity Analysis and Optimization

In this section we first analyze the complexity of Algorithm 3. Let:

1. N be the number of event notifications in Σ
2. W be the number of threads in **WAIT**
3. R be the number of threads in **RUN** $\cup$ **READY**

The time complexity of step 1 is $N \times W$ because the two functions MIN() and GETWAITINGTHREADS()

are both of constant time complexity. The time complexity of step 2 is $W \times R$ because isValid() and table look-up of ETP are both of constant time complexity. The time complexity of step 3 is $W \times W$ based on the implementation. getSegmentID() is implemented as a table look-up function and has constant time complexity. The time complexity of step 4 is $N \times W$. Therefore, the overall time complexity of Algorithm 3 is $W \times (W + R + 2 \times N)$.

Note that W and R are fixed and specified by the SystemC model, so we can only optimize N. Because Σ stores all the event notifications since the start of simulation, its size N keeps growing dramatically and decreases the simulation speed over time. To solve the problem, we remove event notifications from Σ which are earlier than all running or ready threads after AnalyzeAndDeliverEvents() is called [2]. The optimization is safe and valid because the removed event notifications will not in the future wake up any threads. The proof is omitted here for brevity. With this optimization, N is no longer the number of event notifications since the start of simulation but the number of active event notifications. This practically speeds up Algorithm 3 and reduces space cost.

5.5 Experiments and Results

We have implemented Algorithm 3 as an extension of the RISC OoO PDES simulator from [35]. The RISC infrastructure also provides a SystemC compiler that statically analyzes an input SystemC model. ETP is automatically generated by the compiler and provided to the simulator in the instrumented SystemC model. We have evaluated our approach with synthetic examples generated by the TGFF tool and also real-world DVD decoder and GoogLeNet [85] examples. For evaluation, we have measured the execution times under the sequential Accellera simulator (Seq), the original RISC OoO PDES simulator (Org) and the RISC OoO PDES simulator with optimized

[2] `sc_event_and_list` causes an exception here. If a thread th is waiting for a `sc_event_and_list`, the timestamp that th started waiting is also considered in the comparison

event delivery strategy using prediction proposed in this work (Prd). Experiments were performed on an Intel Xeon E3-1240 multi-core processor with 4 cores, 2-way hyperthreaded. The CPU frequency-scaling was turned off so as to provide accurate and repeatable results.

5.5.1 TGFF Examples

In this experiment, we evaluate the performance of the proposed approach with synthetic examples which are automatically generated by the TGFF tool with SystemC extension [58]. Figure 5.5 shows the generic structure of the generated SystemC models. The model contains multiple lanes of nodes between `Stimulus` and `Monitor`. All nodes are connected by user-defined fifo channels. Each channel contains two events. Each node is a SystemC module with a single thread that first reads the data from the input port, performs some intensive computation and finally sends the result to the output port. The model is parameterized and we are able to control:

1. the number of lanes m.
2. the number of nodes per lane n.
3. the computation workload of each node w.

In this experiment, each node has random workload and each lane has the same amount of nodes. The workload of each node is determined by the iterations of a `for` loop, which varies from then thousand to one million. We generated 20 benchmarks with m varying from 1 to 16 and n varying from 2 to 16. Table 5.1 shows the run-times of the examples with Seq, Org and Prd. It also shows the speedups of Org vs. Seq and Prd vs. Seq. The speedups are shown in bold font. Table 5.1 allows the following observations:

1. PRD is faster than SEQ. A maximum speed-up of 6.3x is achieved by PRD over SEQ with

Table 5.1: Results of Synthetic Examples: run-time (secs) and speedup (%)

m \ n	2					4					8					16				
	Seq	Org		Prd		Seq	Org		Prd		Seq	Org		Prd		Seq	Org		Prd	
1	76.71	76.77	**100**	52.34	**147**	119.77	119.86	**100**	53.08	**225**	176.71	176.81	**100**	54.94	**322**	355.52	355.81	**100**	72.36	**491**
2	119.80	100.16	**120**	53.11	**225**	176.69	128.62	**137**	54.77	**323**	355.50	241.22	**147**	71.51	**497**	672.03	436.20	**154**	112.16	**599**
4	176.83	101.22	**175**	55.61	**318**	355.55	161.40	**220**	72.56	**490**	672.33	276.70	**243**	113.86	**590**	1365.01	530.30	**257**	215.25	**634**
8	355.88	115.23	**309**	84.73	**420**	672.14	190.62	**353**	117.95	**570**	1364.99	353.10	**387**	216.28	**631**	2712.08	664.17	**408**	430.36	**630**
16	672.12	161.94	**415**	129.86	**518**	1364.71	290.58	**470**	219.88	**621**	2712.02	549.42	**494**	431.16	**629**	5491.15	1140.51	**481**	895.85	**613**

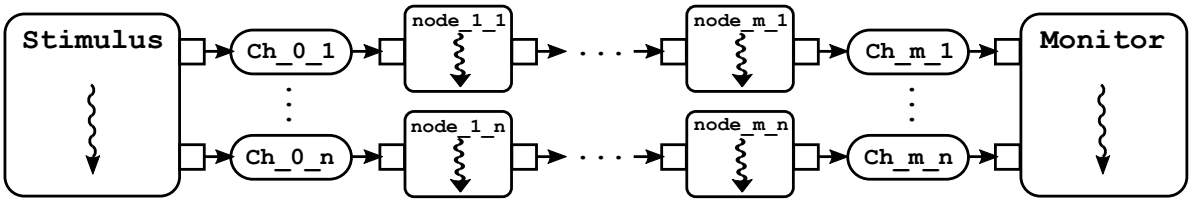


Figure 5.5: SystemC model of synthetic examples

m=4 and n=16, which is impressive on a 4-core machine with 8 hyperthreads. Note that a naive theoretical speed-up of 8x cannot be achieved. This is because there are only four FPUs on the host processor. Due to the intensive computations in each node, the eight hyperthreads are not allowed to run fully in parallel. We also notice that the speedup slightly drops when $m = 16$ and $n = 16$. This is due to the largely increased context switching and contentions of FPUs between threads.

2. PRD is faster than ORG. The speedup is most obvious when there is only one lane in the model. ORG has no speedup against SEQ while PRD has an increasing speedup with the number of nodes. Under ORG, requirement 1 and 2 in Section 5.3-5.3.2 become a global barrier and forbid threads that have different timestamps to execute Out-of-Order. On the other hand, PRD makes correct predictions and identifies that only neighboring nodes have dependency while others are able to run Out-of-Order. Therefore, more events are delivered and more threads wake up in the same scheduling step, which as a result increases the execution speed of the model. Due to the hardware limitations (number of hyperthreads and FPUs), the speedup decreases with the increasing of m. However, PRD is still significantly faster than ORG.

The observations confirm the effectiveness of the proposed event delivery strategy using predictions. We achieve a maximum speedup of 6.3x over Accellera sequential simulation and 4.9x over the original OoO PDES, which are impressive on a 4-core machine.

5.5.2 DVD Player

In this experiment we evaluate our approach using the DVD player example that is similar to the one in Figure 5.1a. After decoding, the results are sent to a `Monitor` module. The communication in this model is via user-defined double-handshake channels. The results of run-times and speedups compared to the Accellera sequential simulation are shown in Table 5.2.

In this example, the three decoding lanes are independent and are able to execute in parallel.

Table 5.2: Results of DVD Player: run-time (secs) and speedup (%)

Seq	Org		Prd	
209.51	88.49	**236**	73.35	**284**

However the original OoO PDES (Org) imposes a false global barrier such that one lane can only continue execution until the other two lanes have finished execution. The barrier reduces the parallelism level of the model and decreases the execution speed, resulting in a speedup of 2.3x over Seq. On the other hand, our proposed approach successfully predicts that the three lanes are independent and therefore decoder threads wake up Out-of-Order. As a result, the execution speed increases and achieves a speedup of 2.8x over Seq and 1.2x over Org. Note that a naive maximum speedup of 3x is not achieved because the workload of `VideoDecoder` and `AudioDecoder`s are different. Specifically, `VideoDecoder` takes longer to process its frames and becomes a sequential bottleneck. According to Amdahl's law, a speedup of 2.8x is reasonable. This experiment confirms the correctness and effectiveness of our proposed event delivery strategy.

5.5.3 GoogLeNet

Image classification is a hot topic nowadays [82, 83, 84]. One convincing approach for image classification and detection is GoogLeNet [85], which is a deep convolutional neural network model. In this experiment, we implement a SystemC model of GoogLeNet [86]. Each layer is implemented in a separate module. The communications are via channels. The architecture is shown in Figure 5.6. Including `Stimulus` and `Monitor`, there are a total of 146 module instances in this SystemC model and each module instance has a single thread. In this experiment, 500 images are fed into the GoogLeNet SystemC model for classification. The results are verified and are correct. The experimental results of run-times and speedups compared to the Seq are shown in Table 5.3

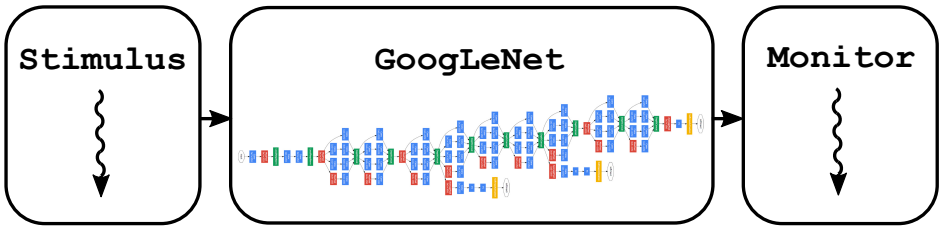


Figure 5.6: SystemC model of GoogLeNet [85]

Table 5.3: Results of GoogLeNet: run-time (secs) and speedup (%)

Seq	Org		Prd	
947.30	361.31	**262**	210.97	**450**

In this experiment, Prd achieves a speedup of 4.5x against Seq. A naive maximum speedup of 8x cannot be achieved. This is because the workloads of module instances (layers in GoogLeNet) are not perfectly balanced and heavy ones become sequential bottlenecks in the data flow of the model. Therefore, eight hyperthreads cannot execute totally in parallel. Also, there are only four FPUs which may introduce contentions between threads. Nevertheless, the result is still impressive on

a 4-core machine. Compared to Org, Prd is 1.7x faster. This experiment confirms again that the proposed event delivery strategy using prediction is effective and correct.

5.6 Conclusion

In this chapter, we propose an optimized event delivery strategy using prediction information for OoO PDES. Our work allows the scheduler to deliver more events each scheduling step, resulting in more threads running in parallel and increased simulation speed. We have demonstrated the effectiveness of the proposed approach with synthetic and demonstration examples. Significant and impressive speedups are achieved against Accellera sequential simulation and the original OoO PDES.

Chapter 6

Conclusion

In this chapter, we summarize the contributions of this dissertation and then explore potential researches to be done in the future.

6.1 Contributions

The goals of this dissertation are listed in Section 1.4. We fulfill the goals with the following four contributions:

- A scalable RISC tool flow for statically analyzing and protecting third party IPs during OoO PDES [62] fulfills the first goal.
- An advanced static analysis approach for modern SystemC TLM-2.0 models [63] fulfills the second goal.
- A set of coding guidelines for RISC users to achieve higher simulation speed [64] fulfills the third goal.

- A more efficient event delivery algorithm in OoO PDES scheduler using prediction information [65] fulfills the third goal.

6.1.1 A Scalable Solution for Statically Analyzing 3rd Party IPs

In Chapter 2, we proposed the PSG technique which enables the RISC compiler to statically analyze SystemC models composed of multiple translation units and IPs. The technique also aims to protect IPs from security leakage. PSGs are first constructed on the IP provider's side. Then, they are shipped together with the library files to the user. During static analysis of the user's SystemC model, PSGs are integrated at top level to reconstruct the complete SG. Experimental results confirm the effectiveness of this approach.

6.1.2 A Static Analysis Approach for SystemC TLM-2.0 Models

In Chapter 3, we proposed the static analysis approach for SystemC TLM-2.0 models in the RISC compiler. A new SCP technique is introduced to facilitate the static analysis. In SystemC TLM-2.0 models, pointers are frequently used, which makes it difficult to statically analyze data conflicts because pointers may dynamically point to any memory address. However, we notice that in a well-written SystemC TLM-2.0 model, pointers are entangled via TLM-2.0 callback methods. Based on this observation, we propose the approach to analyze entangled variables using SCP. Four types of TLM-2.0 communications are supported with this approach: BTI, NBTI, DMI and DTI. Hierarchical and interconnected communication schemes are also correctly analyzed.

6.1.3 Coding Guidelines for RISC Users

In Chapter 4, we proposed coding guidelines for RISC users. Since OoO PDES schedules SystemC processes at segment level, a proper granularity of SG allows the simulator to run more threads in parallel, which in turn speeds up the simulation. The granularity of SG can be controlled by placing scheduling step primitives, i.e., wait statements. Two major suggestions are described in Section 4.2.5, namely: use the wait-for-delta-cycle primitive as the extra segment boundary, and partition the heavy segments.

6.1.4 A More Efficient Event Delivery Algorithm in OoO PDES Scheduler Using Prediction Information

In Chapter 5, we proposed a new event delivery algorithm in OoO PDES. Prediction information about the behaviors of threads in the future is provided to the OoO PDES simulator. The algorithm first calculates the earliest possible wake-up timestamp of each waiting thread. If at that timestamp the thread can be waked up by an event, the scheduler then moves it to the *READY* queue. The algorithm allows more threads to wake up at each scheduling step, which in turn increases the simulation speed.

6.2 Future Work

In addition to the work presented in this dissertation, we propose some potential topics to research in the future.

6.2.1 More Accurate Static Analysis for SystemC Models

Current RISC compiler cannot statically analyze arbitrary pointers and arrays, which increases the false data conflicts in the data conflict table and reduces the simulation speed. Also, it is not able to analyze PCP and SCP given port and socket arrays. In the future, we would like to improve the static analysis algorithm to support the above scenarios for higher accuracy.

6.2.2 More Efficient OoO PDES

In this dissertation, we optimize the event delivery algorithm in OoO PDES. However, there is a time overhead in the event delivery algorithm itself. In the future work, we plan to optimize the algorithm to reduce the overhead. Furthermore, the experiments are mostly performed on an 8-core machine. In the future, we plan to perform more experiments on a many-core processor to further analyze the scalability of the OoO PDES algorithm.

Bibliography

[1] Senouci, B. and Rabiaa Zitouni. FPGA based Networked Embedded Systems Design and Prototyping : Automobile oriented Applications. 2016.

[2] Khan, Jehangir, Smaïl Niar, Mazen A. R. Saghir, Yassin Elhillali and Atika Rivenq. Driver assistance system design and its optimization for FPGA based MPSoC. In IEEE 7th Symposium on Application Specific Processors (2009): 62-65.

[3] Claus, Christopher, Walter Stechele and Andreas Herkersdorf. Autovision: A Run-time Reconfigurable MPSoC Architecture for Future Driver Assistance Systems. In Information Technology 49 (2007): 181 - 187.

[4] Stein, Gideon P., Elchanan Rushinek, Gaby Hayun and Amnon Shashua. A Computer Vision System on a Chip: a case study from the automotive domain. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (2005): 130-130.

[5] Soojin Park. Software Requirement Specification Based on a Gray Box for Embedded Systems: A Case Study of a Mobile Phone Camera Sensor Controller. In Computers 2019, 8, 20.

[6] Dražen Pašalić, Branimir Cvijić, Dušanka Bundalo and Zlatko Bundalo, Goran Kuzmić. Embedded systems for user identification in access to objects and services using mobile phone. In 6th Mediterranean Conference on Embedded Computing, Bar, 2017, pp. 1-4.

[7] Bruce Mehler, David Kidd, Bryan Reimer, Ian Reagan, Jonathan Dobres and Anne McCartt.

Multi-modal assessment of on-road demand of voice and manual phone calling and voice navigation entry across two embedded vehicle systems. In Ergonomics 59:3, pages 344-367.

[8] Andreas Kamilaris and Andreas Pitsillides. Mobile Phone Computing and the Internet of Things: A Survey. In IEEE Internet of Things Journal, vol. 3, no. 6, pp. 885-898, Dec. 2016.

[9] José Manuel Bote, Joaquín Recas, Francisco Rincón, David Atienza and Román Hermida. A Modular Low-Complexity ECG Delineation Algorithm for Real-Time Embedded Systems. In IEEE Journal of Biomedical and Health Informatics, vol. 22, no. 2, pp. 429-441, March 2018.

[10] Dimitra Azariadi, Vasileios Tsoutsouras, Sotirios Xydis and Dimitrios Soudris. ECG signal analysis and arrhythmia detection on IoT wearable medical devices. In 5th International Conference on Modern Circuits and Systems Technologies, Thessaloniki, 2016, pp. 1-4.

[11] Joao T. Fernandes, B. M. Racha, R. P. Paiva and Tiago J. Cruz. Using Low Cost Embedded Systems for Respiratory Sounds Auscultation. In 15th International Symposium on Wireless Communication Systems, Lisbon, 2018, pp. 1-5.

[12] Tarcísio Pereira, Deivson Albuquerque, Aêda Sousa, Fernanda Alencar and Jaelson Castro. Towards a Metamodel for a Requirements Engineering Process of Embedded Systems. In VI Brazilian Symposium on Computing Systems Engineering, Joao Pessoa, 2016, pp. 93-100.

[13] Ouda, A.N. A robust adaptive control approach to missile autopilot design. In International Journal of Dynamics and Control 6, 1239–1271 (2018).

[14] Bahaaeldin Gamal Abdelaty, Ahmed Nasr Ouda; Yehia Zakaria Elhalwagy and Gamal Ahmed Elnashar. Embedded Tracking System for Ground Moving Vehicle. In International Conference on Aerospace Sciences and Aviation Technology, 17, 1-6.

[15] Jang, Joon-Ha and Jin-Young Choi. Formal Specification and Verification of System of Systems Using UPPAAL: A Case Study of a Defensive Missile Systems. In JCM 12 (2017): 482-488.

[16] Ignacio Sañudo, Paolo Cortimiglia, Luca Miccio, Marco Solieri, Paolo Burgio, Christian Di Biagio, Franco Felici, Giovanni Nuzzo and Marko Bertogna. The Key Role of Memory in Next-Generation Embedded Systems for Military Applications. In Proceedings of 6th International Conference in Software Engineering for Defence Applications, 2018.

[17] Meikang Qiu and Edwin H. -M. Sha. Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems. In ACM Transactions on Design Automation of Electronic Systems, 14, 2, Article 25, April, 2009.

[18] S. A. Edwards and O. Tardieu. SHIM: a deterministic model for heterogeneous embedded systems. In IEEE Transactions on Very Large Scale Integration Systems, vol. 14, no. 8, pp. 854-867, Aug. 2006.

[19] S. Mohanty, V. K. Prasanna, S. Neema and J. Davis. Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In ACM SIGPLAN Notices 37, 7, June 2002.

[20] Amal Ben Ameur, Didier Martinot, Patricia Guitton-Ouhamou, Valerio Frascolla, Francois Verdier and Michel Auguin. Power and performance aware electronic system level design. In 12th IEEE International Symposium on Industrial Embedded Systems, Toulouse, 2017, pp. 1-4.

[21] Ashok B. Mehta. ESL (Electronic System Level) Verification Methodology. In ASIC/SoC Functional Design Verification. Springer, Cham.

[22] Andreas Gerstlauer, Christian Haubelt, Andy D. Pimentel, Todor P. Stefanov, Daniel D. Gajski, and Jügen Teich. Electronic System-Level Synthesis Methodologies. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 28, no. 10, pp. 1517-1530, Oct. 2009.

[23] K. Keutzer, A.R. Newton, J.M. Rabaey and A. Sangiovanni-Vincentelli. System-level design:

orthogonalization of concerns and platform-based design. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, pp. 1523-1543, Dec. 2000.

[24] David C Black, Jack Donovan, Bill Bunton and Anna Keist. SystemC: From the ground up. In Springer Science and Business Mediam 2009 Dec 18.

[25] Frank Ghenassia. Transaction-level modeling with SystemC. Dordrecht, The Netherlands, Springer, 2005.

[26] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer and Shuqing Zhao. SpecC: Specification language and methodology. In Springer Science and Business Media, 2012 Dec 6.

[27] IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual, IEEE Computer Society, January 2012.

[28] Andreas Gerstlauer, Rainer Dömer, Junyu Peng and Daniel D. Gajski. System Design: A Practical Guide with SpecC, Kluwer Academic Publishers, Boston, 2001.

[29] OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL, the Open SystemC Initiative (OSCI), July 2009.

[30] George S. Fishman. Principles of discrete event simulation. John Wiley and Sons, New York, NY, Jan, 1978.

[31] SystemC Language Working Group, SystemC 2.3.1, Core SystemC Language and Examples, Accellera Systems Initiative, 2014.

[32] Weiwei Chen, Xu Han and Rainer Dömer. Multicore Simulation of Transaction-Level Models Using the SoC Environment. In IEEE Design and Test of Computers, vol. 28, no. 3, pp. 20-31, May-June, 2011.

[33] Richard M. Fujimoto. Parallel discrete event simulation. In Communications of the ACM, Oct, 1990.

[34] Weiwei Chen, Xu Han, Che-Wei Chang, Guantao Liu and Rainer Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 33(12):1859-1872, 2014.

[35] Lab for Embedded Computer Systems (LECS), Recoding Infrastructure for SystemC [Online], Available: http://www.cecs.uci.edu/~doemer/risc.html#RISC060

[36] Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza and Rainer Dömer. RISC Compiler and Simulator, Release V0.6.0: Out-of-Order Parallel Simulatable SystemC Subset. Center for Embedded and Cyber-Physical Systems, Technical Report 19-04, September 2019.

[37] Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. ROSE User Manual: A Tool for Building Source-to-Source Translators Draft User Manual (version 0.9.10.231). Lawrence Livermore National Laboratory, April 19, 2019.

[38] Rainer Dömer, Guantao Liu and Tim Schmidt. Parallel Simulation, chapter 17 in Handbook of Hardware/Software Codesign by Soonhoi Ha and Jürgen Teich. Springer Netherlands, Dordrecht, June 2017.

[39] Anirudh Kaushik and Hiren D. Patel. SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models. In Proceedings of Forum on Specification and Design Languages, Paris 2013.

[40] Tim Schmidt, Zhongqi Cheng and Rainer Dömer. Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation. In Proceedings of Design, Automation and Test in Europe, Dresden, Germany, March 2018.

[41] Weiwei Chen and Rainer Dömer. Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions. In Proceedings of Design, Automation and Test in Europe, Grenoble, France, March 2013.

[42] Janne Virtanen, Panu Sjövall, Marko Viitanen, Timo D. Hämäläinen and Jarno Vanne. Distributed systemc simulation on manycore servers. In IEEE Nordic Circuits and Systems Conference, Copenhagen, 2016, pp. 1-6.

[43] Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Dietmar Petras and Andreas Hoffmann. SystemC-link: Parallel SystemC simulation using time-decoupled segments. In Design, Automation and Test in Europe Conference and Exhibition, Dresden, 2016, pp. 493-498.

[44] Denis Becker, Matthieu Moy and Jérôme Cornet. Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study. In Electronics 2016, 5(2), 22.

[45] Matthieu Moy. Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach. In Proceedings of Design, Automation and Test in Europe, 2013.

[46] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard and Julien Zory. Relaxing Synchronization in a Parallel SystemC Kernel. In IEEE International Symposium on Parallel and Distributed Processing with Applications, 2008.

[47] Tianlin Li, Yiping Yao, Wenjie Tang, Feng Zhu and Zhongwei Lin. An efficient multi-threaded memory allocator for PDES applications. In Simulation Modelling Practice and Theory, Volume 100, April 2020, 102067.

[48] Brandon Waddell and James F. Leathrum: A Multithreaded Simulation Executive in Support of Discrete Event Simulations. In Winter Simulation Conference, National Harbor, MD, USA, 2019, pp. 2677-2688.

[49] Stefano Carnà, Serena Ferracci, Emanuele De Santis, Alessandro Pellegrini and Francesco Quaglia. Hardware-Assisted Incremental Checkpointing in Speculative Parallel Discrete Event Simulation. In Winter Simulation Conference, National Harbor, MD, USA, 2019, pp. 2759-2770.

[50] Masashi Shiraishi, Atsuo Ozaki and Shusuke Watanabe. Optimistic parallel simulation engine for predicting future situations incorporating observation data. In 18th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Kanazawa, 2017, pp. 351-356.

[51] Alessandro Pellegrini. Optimizing memory management for optimistic simulation with reinforcement learning. In International Conference on High Performance Computing and Simulation, Innsbruck, 2016, pp. 26-33.

[52] Emanuele Santini, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. Hardware-Transactional-Memory Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models. In IEEE 22nd International Conference on High Performance Computing, Bangalore, 2015, pp. 145-154.

[53] Deepak Jagtap, Nael Abu-Ghazaleh and Dmitry Ponomarev. Optimization of Parallel Discrete Event Simulator for Multi-core Systems. In IEEE 26th International Parallel and Distributed Processing Symposium, Shanghai, 2012, pp. 520-531.

[54] Georg Kunz, Daniel Schemmel, James Gross and Klaus Wehrle. Multi-level Parallelism for Time- and Cost-Efficient Parallel Discrete Event Simulation on GPUs. In ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, Zhangjiajie, 2012, pp. 23-32.

[55] Bacon David and Sweeney Peter. Fast Static Analysis of C++ Virtual Function Calls. In ACM Special Interest Group on Programming Languages, 1999.

[56] John Viega, J.T. Bloch, Yoshi Kohno and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In ACM Transactions on Information and System Security, 2011, 257-267,

[57] Rio Astorga, David del, Manuel F Dolz, Luis Miguel Sánchez, J Daniel García, Marco Danelutto and Massimo Torquati. Finding Parallel Patterns through Static Analysis in C++

Applications. In The International Journal of High Performance Computing Applications 32, no. 6 (November 2018): 779–88.

[58] Guantao Liu, Tim Schmidt and Rainer Dömer. A Segment-Aware Multi-Core Scheduler for SystemC PDES. In Proceedings of the International High Level Design Validation and Test Workshop, Santa Cruz, California, October 2016.

[59] Guantao Liu, Tim Schmidt, Rainer Dömer, Ajit Dingankar and Desmond Kirkpatrick. Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories. In Proceedings of the Asia and South Pacific Design Automation Conference 2015, Tokyo, Japan, January 2015.

[60] Tim Schmidt, Guantao Liu and Rainer Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In Proceedings of the Design Automation Conference 2017, Austin, TX, June 2017.

[61] Daniel Mendoza, Agit Dingankar, Zhongqi Cheng and Rainer Dömer. Integrating Parallel SystemC Simulation into Simics(R) Virtual Platform. In Proceedings of the Design and Verification Conference in Europe, Munich, Germany, October 2019.

[62] Zhongqi Cheng, Tim Schmidt, Rainer Dömer. Enabling IP Reuse and Protection in Out-of-Order Parallel SystemC Simulation. In Proceedings of the International Embedded Systems Symposium, Springer, Friedrichshafen, Germany, September 2019.

[63] Zhongqi Cheng, Rainer Dömer. Analyzing Variable Entanglement for Parallel Simulation of SystemC TLM-2.0 Models. In ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5s, article 79, 20 pages, October 2019.

[64] Zhongqi Cheng, Tim Schmidt, Rainer Dömer. SystemC Coding Guideline for Faster Out-of-Order Parallel Discrete Event Simulation. In Proceedings of Forum on Specification and Design Languages, Munich, Germany, September 2018.

[65] Zhongqi Cheng, Emad Arasteh, Rainer Dömer. Event Delivery using Prediction for Faster Parallel SystemC Simulation. In Proceedings of Asia and South Pacific Design Automation Conference, Beijing, China, January 2020.

[66] S. Sarkar, S. Chanclar G and S. Shinde. Effective IP reuse for high quality SOC design. In Proceedings 2005 IEEE International SOC Conference, Herndon, VA, 2005, pp. 217-224.

[67] Andrew B. Kahng, John Lach, William H. Mangione-Smith, Stefanus Mantik, Igor L. Markov, Miodrag Potkonjak, Paul Tucker, Huijuan Wang and Gregory Wolfe. Constraint-based watermarking techniques for design IP protection. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 10, pp. 1236-1252, Oct. 2001.

[68] Marcello Dalpasso, Alessandro Bogliolo and Luca Benini. Hardware/software IP protection. In Proceedings of the 37th Annual Design Automation Conference, Association for Computing Machinery, New York, NY, USA, 593–596.

[69] Tim Schmidt, Guantao Liu and Rainer Dömer. Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation. In Asia and South Pacific Design Automation Conference, Tokyo, Japan, January 2017.

[70] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[71] John F. Canny. A Computational Approach To Edge Detection. In IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.

[72] Zhongqi Cheng, Tim Schmidt, Guantao Liu and Rainer Dömer. Thread- and Data-Level Parallel Simulation in SystemC, a Bitcoin Miner Case Study. In Proceedings of the International High Level Design Validation and Test Workshop 2017, Santa Cruz, California, October 2017.

[73] Markus Becker, Giuseppe Di Guglielmo, Franco Fummi, Wolfgang Mueller, Graziano Pravadelli and Tao Xie. RTOS-aware refinement for TLM2.0-based HW/SW designs. In Design, Automation and Test in Europe Conference and Exhibition, Dresden, 2010, pp. 1053-1058.

[74] Adán Kohler and Martin Radetzki. A SystemC TLM2 model of communication in wormhole switched Networks-On-Chip. In Forum on Specification and Design Languages, Sophia Antipolis, 2009, pp. 1-4.

[75] Jones Y. Mori and Michael Huebner. A high-level analysis of a multi-core vision processor using SystemC and TLM2.0. In International Conference on ReConFigurable Computing and FPGAs, Cancun, 2014, pp. 1-6.

[76] Rainer Dömer. Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation. In IEEE Embedded Systems Letters, vol. 8, no. 4, pp. 81-84, December 2016.

[77] DOULOS tutorial: Getting Started with TLM-2.0 [Online]. Available: https://www.doulos.com/knowhow/systemc/tlm2/tutorial__1/

[78] Accellera SystemC supplemental material [Online]. Available: https://www.accellera.org/downloads/standards/systemc.

[79] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland and David Svoboda. Java coding guidelines: 75 recommendations for reliable and secure programs. Addison-Wesley, 2013 Aug 23.

[80] Robert Green and Henry Ledgard. Coding guidelines: Finding the art in the science. In Communications of the ACM, 2011 Dec 1;54(12):57-63.

[81] Moin Syed and Sarah C Nelson. Guidelines for establishing reliability when coding narrative data. In Emerging Adulthood, 2015 Dec;3(6):375-87.

[82] Robert M. Haralick, K. Shanmugam and Its'Hak Dinstein. Textural Features for Image Classification. In IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-3, no. 6, pp. 610-621, Nov. 1973.

[83] D. Lu and Q. Weng. A survey of image classification methods and techniques for improving classification performance. In International Journal of Remote Sensing, 28:5, 823-870,

[84] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie and Mu Li. Bag of Tricks for Image Classification with Convolutional Neural Networks. In The IEEE Conference on Computer Vision and Pattern Recognition, 2019, pp. 558-567

[85] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. Going deeper with convolutions. In The IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, 2015.

[86] Emad Malekzadeh Arasteh and Rainer Dömer. An Untimed SystemC Model of GoogLeNet. In Proceedings of the International Embedded Systems Symposium, 2019.

[87] Wolfgang Mueller, Rainer Dömer and Andreas Gerstlauer. The Formal Execution Semantics of SpecC. In Proceedings of the International Symposium on System Synthesis, Kyoto, Japan, October 2002.

[88] Sandro Rigo, Rodolfo Azevedo and Luiz Santos. Electronic system level design: an open-source approach. In Springer Science and Business Media; 2011 Apr 28.

[89] Grant Martin, Brian Bailey, Andrew Piziali. ESL design and verification: a prescription for electronic system level methodology, Elsevier, 2010 Jul 27.

[90] L. Benini, A. Bogliolo and G. De Micheli. A survey of design techniques for system-level dynamic power management. In IEEE transactions on very large scale integration systems. 2000 Jun;8(3):299-316.

[91] A. Sangiovanni Vincentelli. Electronic system design in the automobile industry. In IEEE Micro, 2003 Jul 9;23(3):8-18.

[92] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming 2003 Jun 11 (pp. 167-178).

[93] Konstantin Serebryany and Timur Iskhodzhanov. Data race detection in practice. In Proceedings of the workshop on binary instrumentation and applications 2009 Dec 12 (pp. 62-71).

[94] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation 2002 May 17 (pp. 258-269).

[95] Robert H.B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming 1991 Apr 1 (pp. 133-144).

[96] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In ACM Sigplan Notices. 1995 Jun 1;30(6):1-2.

[97] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In ACM SIGPLAN Notices. 2004 Jun 9;39(6):145-57.

[98] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In ACM SIGPLAN Notices. 1999 May 1;34(5):77-90.

[99] Christoph Schumacher, Rainer Leupers, Dietmar Petras and Andreas Hoffmann. parSC: Synchronous parallel SystemC simulation on multi-core host architectures. In IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Scottsdale, AZ, 2010, pp. 241-246.