UNIVERSITY OF CALIFORNIA,
IRVINE

A Compiler Infrastructure for Static and Hybrid Analysis of Discrete Event System Models

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Tim Schmidt

Dissertation Committee:
Professor Rainer Dömer, Chair
Professor Kwei-Jay Lin
Professor Fadi Kurdahi

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ACRONYMS

API Application Programming Interface

AST Abstract Syntax Tree

CPU Central Processing Unit

DES Discrete Event Simulation

DUT Device Under Test

GPU Graphics Processing Unit

IP Intellectual Property

LECS Lab for Embedded Computer Systems

NoC Network-on-Chip

PC Personal Computer

PCP Port Call Path

PDES Parallel Discrete Event Simulation

RISC Recoding Infrastructure for SystemC

RTL Register-Transfer Level

SG Segment Graph

SIMD Single Instruction Multiple Data

SLD System Level Design

SLDL System Level Description Language

TCG Thread Communication Graph

TLM Transaction-Level Modeling

ACKNOWLEDGMENTS

Here we are, the last chapter is written and the section acknowledgments is the last blank spot. So far, I read many cheesy acknowledgments and I am still not sure about the proper wording. Let's see where this one goes.

The most important person for this dissertation was clearly professor Rainer Dömer. He welcomed me with the words *"Strange is good, ordinary would be boring! ;-)"* and I agree with him. Definitely, I was the loudest, stubbornest, and probably most enthusiastic student over the last years. He gave me the chance to express my opinion, make mistakes, and correct me when needed. The chance of being deeply involved in a great research project with Intel cooperation from the beginning on is quite unique. I would like to thank him for all of his support.

Also, I would like to thank professor Kwei-Jay Lin and Fadi Kurdahi for serving as my committee members.

I would like to thank Intel Cooperation for the support by funding the RISC project. Specifically, Desmond, Abijit, and Adjit for their discussions and valued input for this project.

Furthermore, I would like to thank Dan Quinlan and the ROSE team for their compiler framework. Dan fixed issues for us and made it possible to achieve great progress on our side of the project.

Also, I would like to thank my lab friends Guantao and Zhongqi for their discussions and support. Their solid work of contributing examples, test cases, implementing the simulator and extending the infrastructure made this dissertation possible.

Additionally, I would like to thank my former undergraduate students Alex, Farah, Nikka, Stanley, and Steven for working with me. They have written many test cases, transformed models, and optimized the SystemC library.

On the personal level I would like to thank various people.

First, I would like to thank my friend Carsten who listened to all of my ups and downs over last years. I am not sure where this journey would be without his support. Also, I would like to thank Matthias and Weiwei for their support, help, and friendship. Moreover, I would like to thank Maryam and Davit for being one of my best friends on and off campus. They made my life humorous and entertaining in and around Irvine. A big thank you to all of them!

Next, everybody who was in our lab room has noticed the 'wall' of board games and my enthusiasm for gaming. As the host of the weekly board game night, we had 25 different nationalities and the diversity undergrad students, graduate students, visiting students, post docs, visiting professors, professors, and many others. The weekly gathering gave us the opportunity to connect to others and gain new perspectives in a free and open space. I

would like to thank everybody who showed up and played with us.

During my time in Irvine, I stayed with Joyce and Ron where I became a fully integrated family member. I would like to thank both of them for their hospitality and the support they gave me. Also, I would like to thank our dog Christopher for his never ending happiness.

Furthermore, I would like to thank Daniel Gajski for teaching me chess, explaining the world to me, and giving me a broader perspective of life. Particularly, I enjoyed our weekly lunch and our critical discussion about anything and everything while having a burger and a soda.

Last but not least, I would like to thank Melanie, Grace, and Amy. Whenever there was an issue of paper work, they helped me. Whenever there was a need to fix something, they did it.

CURRICULUM VITAE

Tim Schmidt

EDUCATION

Doctor of Philosophy in Computer Engineering **2018**
University of California, Irvine *Irvine, California, USA*

Master of Science in
Embedded Systems and Microrobotics **2013**
University of Oldenburg *Oldenburg, Germany*

Bachelor of Science in Computer Sciences **2010**
University of Paderborn *Paderborn, Germany*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2018**
University of California, Irvine *Irvine, California, USA*

TEACHING EXPERIENCE

Teaching Assistant **2015–2017**
University of California, Irvine *Irvine, California, USA*

INDUSTRIAL EXPERIENCE

Research Intern **2014**
Intel Cooperation *Hillsboro, Oregon, USA*

REFEREED BOOK CHAPTERS

R. Dömer, G. Liu, T. Schmidt. **Parallel Simulation** in the *Handbook of Hardware/Software Codesign* by S. Ha and J. Teich. Springer Netherlands, Dordrecht, Netherlands

REFEREED JOURNAL PUBLICATIONS

T. Schmidt, Z. Cheng, and R. Dömer. **RISC: A Static Analysis Framework for Parallel Simulation of SystemC.** Under review, March 2018.

G. Liu, T. Schmidt, and R. Dömer. **A Communication-Aware Thread Mapping Framework for SystemC PDES.** Under review, February 2017.

REFEREED CONFERENCE PUBLICATIONS

T. Schmidt, Z. Cheng, R. Dömer. **Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation**. In Proceedings of Design, Automation and Test in Europe (DATE) Conference, 2018, Dresden, Germany.

T. Schmidt, G. Liu, R. Dömer. **Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation**. In Proceedings of *the Design Automation Conference (DAC)*, 2017, Austin Texas, USA.

T. Schmidt, G. Liu, R. Dömer. **Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation**. In Proceedings of *the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2017, Tokyo, Japan.

G. Liu, T. Schmidt, R. Dömer, A. Dingankar, D. Kirkpatrick. **Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories**. In Proceedings of *the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2015, Tokyo, Japan.

REFEREED WORKSHOP PUBLICATIONS

Z. Cheng, T. Schmidt, R. 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 (HLDVT)*, 2017, Santa Cruz, USA.

G. Liu, T. Schmidt, R. Dömer. **A Segment-Aware Multi-Core Scheduler for SystemC PDES**. In Proceedings of *the International High Level Design Validation and Test Workshop (HLDVT)*, 2016, Santa Cruz, USA.

T. Schmidt, G. Liu, R. Dömer. **Automatic Generation of Thread Communication Graphs from SystemC Source Code**. In Proceedings of *the International Workshop on*

Software and Compilers for Embedded Systems (SCOPES), 2016, St. Goar, Germany.

T. Schmidt, K. Grüttner, R. Dömer, A. Rettberg. **A Program State Machine Based Virtual Processing Model in SystemC**. In Proceedings of *ACM 4th Embedded Operating Systems Workshop (EWiLi)*, 2014, Lisbon, Portugal.

TECHNICAL REPORTS

G. Liu, T. Schmidt, Z. Cheng, R. Dömer. **RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset**. *Report 17-05, Center for Embedded and Cyber-Physical Systems*, University of California, Irvine, USA, July 2017.

F. Arabi, T. Schmidt, R. Dömer. **A Light Weight SystemC Library for Faster Compilation**. *Report 16-07, Center for Embedded and Cyber-Physical Systems*, University of California, Irvine, USA, October 2016.

G. Liu, T. Schmidt, R. Dömer. **RISC Compiler and Simulator, Beta Release V0.3.0: Out-of-Order Parallel Simulatable SystemC Subset**. *Report 16-06, Center for Embedded and Cyber-Physical Systems*, University of California, Irvine, USA, September 2016

G. Liu, T. Schmidt, R. Dömer. **RISC Compiler and Simulator, Alpha Release V0.2.1: Out-of-Order Parallel Simulatable SystemC Subset**. *Report 15-02, Center for Embedded and Cyber-Physical Systems*, Irvine, USA, October 2015.

T. Schmidt et. al. Project group document. **FPGA-basiertes Echtzeit-Kamerasystem fur Fahrerassistenz (FPGA based real time camera system for driving assistance**. University of Oldenburg, Germany, September 2011, Germany.

THESES

T. Schmidt, Master thesis, **A Program State Machine Based Virtual Processing Model in SystemC**, *University of Paderborn*, March 2013, Germany

T. Schmidt, Bachelor thesis, **Development of a Synthesizable Fixed Point / Integer Library for SystemC**, *University of Oldenburg*, March 2010, Germany

OPEN SOURCE SOFTWARE RELEASES

G. Liu, T. Schmidt, Z. Cheng, R. Dömer. **RISC Compiler and Simulator, Release V0.4.0**. (available at http://www.cecs.uci.edu/ doemer/risc.html#RISC040), July 2017.

G. Liu, T. Schmidt, R. Dömer. **RISC Compiler and Simulator, Beta Release V0.3.0**. (available at http://www.cecs.uci.edu/ doemer/risc.html#RISC030), September 2016.

G. Liu, T. Schmidt, R. Dömer. **RISC Compiler and Simulator, Alpha Release V0.2.1**. (available at http://www.cecs.uci.edu/ doemer/risc.html#RISC021), October 2015.

G. Liu, T. Schmidt, R. Dömer. **RISC Compiler and Simulator, Alpha Release V0.2.0**. (available at http://www.cecs.uci.edu/ doemer/risc.html#RISC020), September 2015.

G. Liu, T. Schmidt, R. Dömer. **RISC API, Alpha Release V0.1.0**. (available at http://www.cecs.uci.edu/ doemer/risc.html#RISC010), June 2014.

ABSTRACT OF THE DISSERTATION

A Compiler Infrastructure for Static and Hybrid Analysis of Discrete Event System Models

By

Tim Schmidt

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2018

Professor Rainer Dömer, Chair

The design of embedded systems is a well-established research domain for many decades. However, the constantly increasing complexity and requirements of state-of-the-art embedded systems pushes designers to new challenges while maintaining established design methodologies. Embedded system design uses the concept of Discrete Event Simulation (DES) to prototype and test the interaction of individual components.

In this dissertation, we provide the Recoding Infrastructure for SystemC (RISC) compiler framework to perform static and hybrid analysis of IEEE SystemC models. On one hand, RISC generates thread communication charts to visualize the communication between individual design components. The visualization respects the underlying discrete event simulation semantics and illustrates the individual synchronization steps. On the other hand, RISC translates a sequential model into a parallel model which effectively utilizes multi- and many-core host simulation platforms for faster simulation. This work extends the conflict analysis capabilities for libraries, dynamic memory allocation, channel instance awareness, and references. Additionally, the traditional thread level parallelism is extended with data level parallelism for even faster simulation.

Chapter 1

Introduction

In this chapter, we introduce the foundation for this dissertation. Frist, we put this work in the context of System Level Design (SLD) for embedded systems in Section 1.1 and introduce the System Level Description Language (SLDL) SystemC in Section 1.2. Following, we propose our RISC in Section 1.3 and the underlying Segment Graph (SG) data structure in Section 1.4. Next, we define the goals for this dissertation in Section 1.5 and conclude this chapter with the related work in Section 1.6.

1.1 System Level Design

Embedded systems are part of our daily life, visible as a cell phone or invisible in a combustion engine in a car. The high demand of constantly increasing functionality of these products is associated with more complex design processes. Modern system-on-chip architectures have up to billions of transistors [4] which increase the design complexity dramatically. Designers follow multiple strategies to cope with this complex process to save time and money.

One option is to describe models of prototypes on different levels of abstraction and increase

their granularity iteratively during the design process. After a prototype fulfills the requirements at a certain abstraction level, the model is refined to the next lower abstraction level. Figure 1.1 shows established abstraction levels with the number of components per model at each level. In this dissertation, we focus on designs at the system level.

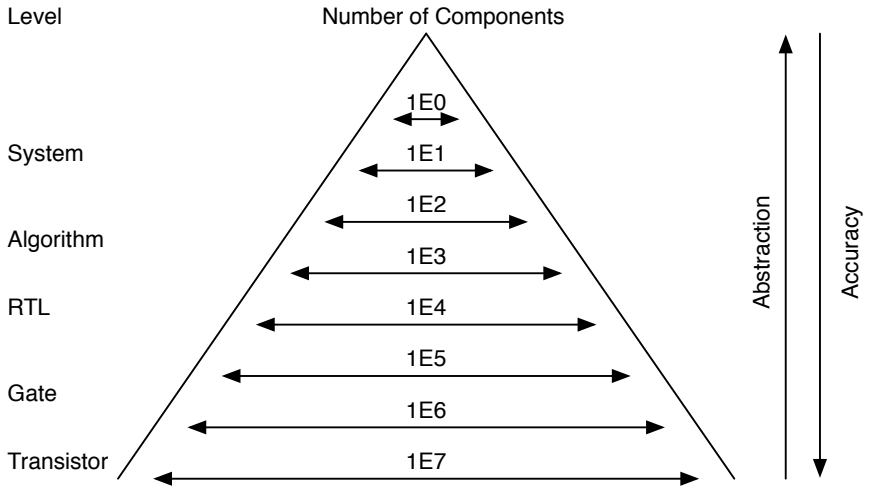


Figure 1.1: Level of abstractions for embedded system design [24]

Designers use simulation as a tool to make better design decisions for their prototypes. However, simulations of complex systems are time consuming and become a bottleneck in the tool flow. Traditionally, simulation uses the concept of DES to simulate the individual components in a sequential fashion.

In this dissertation, we provide a compiler infrastructure to analyze system level models. The proposed RISC compiler infrastructure is implemented for the IEEE SystemC SLDL [1]. The infrastructure is extendable and we can use it in many ways. On one hand, we obtain model information, e.g. communication aspects through visual thread communication graphs. On the other hand, we have advanced static and novel dynamic analysis to execute simulation in parallel.

[1]Note that here the term RISC does not represent the term reduced instruction set computer.

1.2 SystemC

In this section [46], we are introducing the SystemC class library for C++ that is developed for modeling and simulating hardware and software components of embedded systems. UC Irvine presented the very first SystemC version 0.9 in September 1999 and the current version is 2.3.2. The Accellera Systems Initiative [5] maintains the recent IEEE Std 1666™-2011 [3] standard. In this introduction, we focus on core aspects that are the basic components and the simulation kernel.

SystemC Components

In SystemC the structural hierarchy of a model is defined by modules. A SystemC module is the smallest container of functionality with state, behavior, and structure for hierarchical connectivity [9, definition on p. 49]. A module is a class that is derived from the `sc_module` class and can contain sub-modules. This mechanism allows creating structural hierarchy.

The SystemC simulation process is the basic unit of execution [9, p. 51]. A simulation process is associated with a member function of the module and can model for instance combinatorial or sequential circuits. Each process has a sensitivity list that can contain zero, one, or many events. The SystemC scheduler activates a process, if an associated event is triggered.

SystemC distinguishes between three different types of processes, namely `SC_THREAD`, `SC_CTHREAD`, and `SC_METHOD`. A running `SC_METHOD` process is executed from the beginning to the end without any interrupt. After the process has completed, the control returns to the SystemC scheduler. `SC_THREAD` and `SC_CTHREAD` processes are initially started in the elaboration phase (see the following section) and executed until the associated function has *ended* or through a call of the `wait()` function. A `SC_CTHREAD` function is

sensitive to a clock and a `wait` simulates a clock cycle. A `SC_THREAD` can wait for an individual event or a timed event.

The communication between modules is performed by the so-called channel-interface-port concept. A module has ports that allows communication with the environment. The individual ports are typed through an interface. Channels are bound to ports, however, the bound channel and port must use the same interface otherwise a binding is not possible. If a process performs an operation via the port, the interaction will be forwarded to the channel. This concept separates computation and communication. Another type of ports is the so-called export. An export, as the name mentions, exports the interface of a channel or another export to the parent module. A port can be bound to a port or an export.

Figure 1.2 shows a graphical annotation of SystemC elements. The modules $m1$ and $m2$ are communicating via the channel c. The channel provides two different interfaces that are indicated by the different colors. Module $m1$ requires a channel with a blue, and module $m2$ and sub module s require a green interface. The channel 2 is contained in the model $m2$. Via the export, the blue interface is provided to the parent module *testbench*. The process $p1$ drives the port and process $p2$ can only communicate to process $p1$ via events.

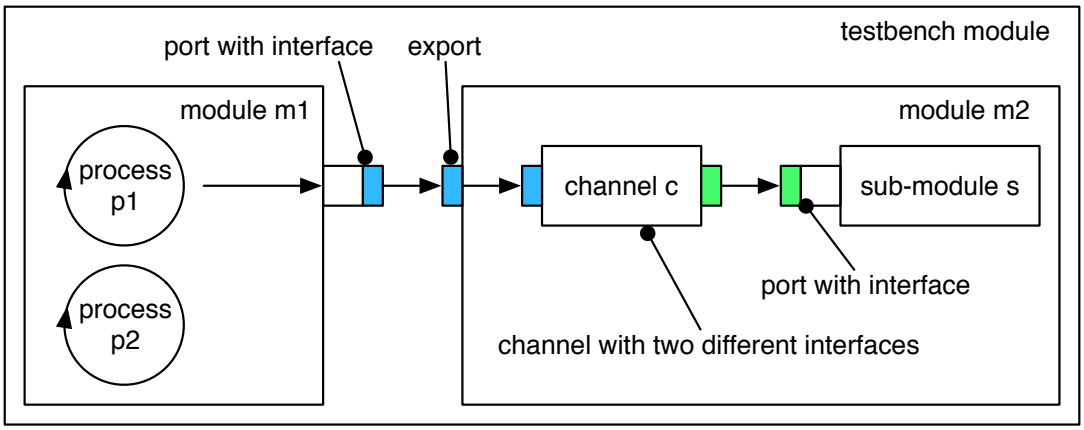


Figure 1.2: Graphical structure of a SystemC model.

SystemC Simulation Kernel

The SystemC simulation kernel has three major phases, namely elaboration, simulation, and clean up. The elaboration is the first phase, when all modules are instantiated and channels are bound to the ports. When all ports and modules are correctly bound, the second phase will start. The function call `sc_start()` starts the simulation of the SystemC scheduler. The semantic of the SystemC simulation kernel is close to the VHDL or Verilog simulation kernel [36] and has eight steps [26, chapter 2.10]. Figure 1.3 shows the individual simulation steps of the kernel. After simulation, the third phase clean up starts and all allocated resources are deallocated.

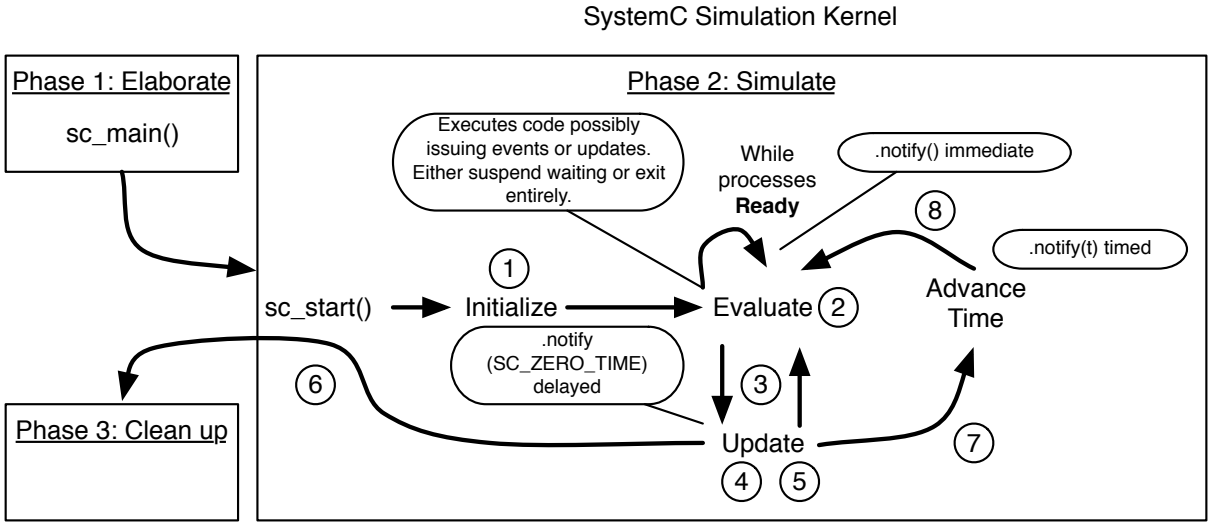


Figure 1.3: The SystemC simulation kernel (source [9, p. 109]). The focus is on the different notification types. Immediate notifications will be considered at the same evaluation phase. Delayed notifications will be considered at the next evaluation phase. A timed notification will appear after t time units.

In the following we discuss the eight steps in detail:

1. Initialize: At the initialization, each process is executed once. A `SC_METHOD` is executed completely and both, the `SC_THREAD` and the `SC_CTHREAD`, are either executed completely or until a `wait` statement occurs. A special function call in the constructor

of the module avoids the initialization of a process. The execution order is undefined and can vary in different simulation kernel implementations.

2. *Evaluate*: The simulator picks a process, which is ready. The process will be executed and may notify events. Every process that is sensitive to this event will be put into the ready set. The process keeps executing until the function ended or a synchronization point occurs.

3. If the ready set is not empty go to step 2.

4. Update: A process can request an update in step 2. The update mechanism is used for delta cycles. A delta cycle contains the evaluation and the update (see Figure 1.3).

5. The update step can trigger new events. If any events have been notified, the processes that are sensitive to these events will be moved to the ready set. If the ready set is not empty go back to step 2. This will increase the delta count, however, not the simulated time.

6. If there is no more timed notification, the simulation will finish.

7. If there is a timed notification, the simulation will continue to the next advanced time cycle and set the delta count to 0.

8. Determine which processes are sensitive to the timed event, and put the processes into the ready set. Then go to step 2.

The timeline of a simulation can be described by two dimensions. On the X axis is the simulated time and on the Y axis is the number of delta cycles. Figure 1.4 shows such a diagram.

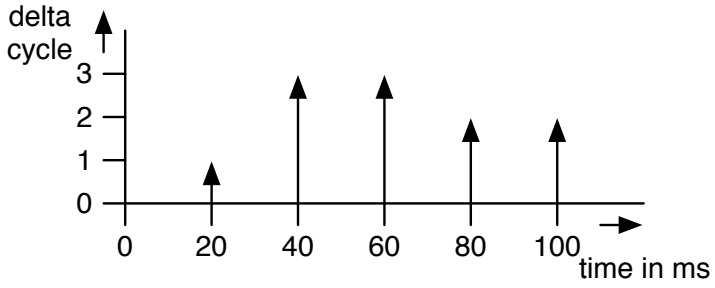


Figure 1.4: Time diagram with delta cycles

1.3 Recoding Infrastructure for SystemC (RISC)

The RISC is a compiler infrastructure for static and hybrid analysis of system level designs written in SystemC. Note again that the term RISC does not represent the reduced instruction set computer architecture.

In this section, we discuss the RISC software stack and the tool flow of the infrastructure.

1.3.1 Software Stack

Figure 1.5 shows the software stack of RISC. On top of a C/C++ software foundation, we selected the ROSE compiler [42] and its internal representation (IR) to generate and maintain an abstract syntax tree (AST) of the design model and the SystemC library. Our SystemC IR layer represents SystemC constructs, such as modules, channels, instances, threads and ports. On top of this, we have placed our Segment Graph Generator (see Section 1.4) which creates graph for the individual threads. The individual back ends (see Section 1.3.2) use the layers below to perform their individual analysis.

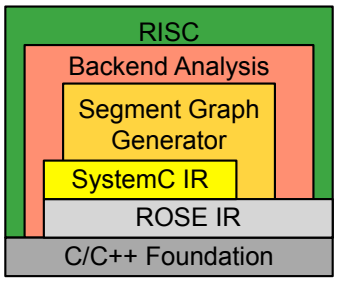


Figure 1.5: Software stack of the RISC compiler

1.3.2 Tool Flow

Now, we discuss the tool flow of the RISC compiler infrastructure which is the central contribution of this dissertation. Figure 1.6 shows the flow of the components in the infrastructure and the dependencies between the individual components.

The input for the RISC compiler is the IEEE SystemC library and a SystemC model. Additionally, the designer can provide results of a previously executed dynamic analysis to support the static analysis.

The compiler has individual steps while processing the input. The foundation is the ROSE compiler infrastructure [42] which translates the SystemC library and the input model into an Abstract Syntax Tree (AST). The ROSE Application Programming Interface (API) makes it possible to perform analysis of variables and data types, modify the AST, and to perform source-to-source transformations. The RISC front end uses these tools to have SystemC sensitive analysis.

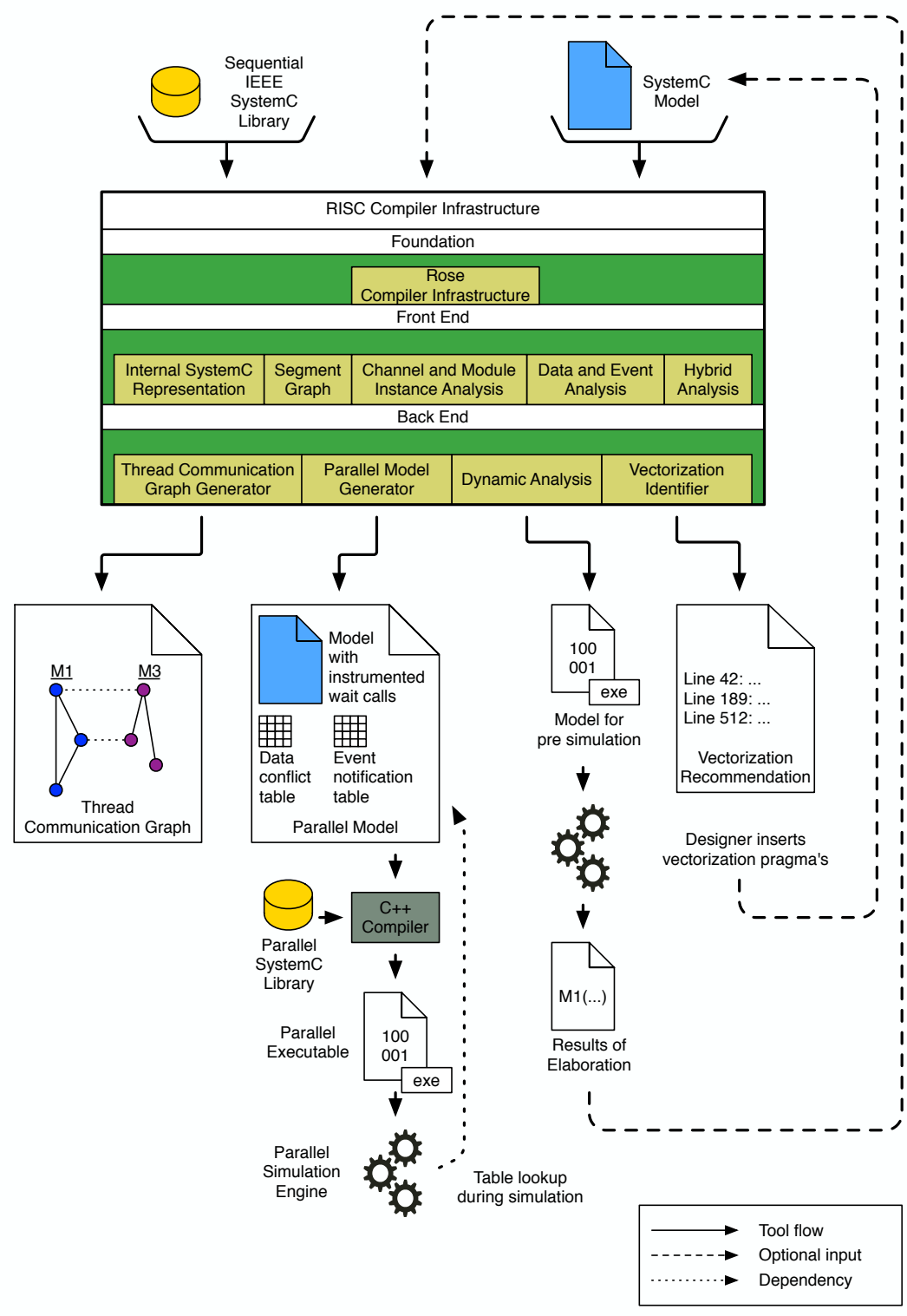


Figure 1.6: Tool flow of the RISC compiler

Specifically, the internal SystemC representation analyzes the input model and identifies all

user defined definitions. In other words, the AST is traversed for the module and channel definitions and their properties as simulation threads, variables, and ports. Based on the knowledge of the internal representation, for each simulation thread a related SG is created. Next, the data conflict and the event notification dependencies among the individual segments are computed (see Section 1.4). To avoid false positive conflicts, we perform the channel and module instance analysis (see Section 3.1.1) to distinguish different instances of variables. Alternatively, we prepare the model thorough transformations for the hybrid analysis (see Chapter 4).

The output of the compiler depends on the individual backend, namely the thread communication graph generator, the parallel model generator, the dynamic analysis, and the vectorization identifier.

The *Thread Communication Graph Generator* produces communication charts to gain a better understanding of the design. Such charts show communication between the individual segments and the related modules which respect to the DES semantics. Through this information, legacy source code is faster analyzable and visually documented. The graph generation is described in detail in Chapter 2.

The output from the *Parallel Model Generator* is a parallel SystemC model (cpp file) including the original model with instrumented `wait` statements, a data conflict table, and an event notification table (see Section 1.4). In other words, this model contains information for the parallel simulation library how to run the individual simulation threads in parallel on different Central Processing Unit (CPU)s. Next, this file is compiled with a general purpose compiler (e.g. GNU's `g++` or Intel's `icpc` for extra vectorization support) and generates an executable. The executable links against a parallel version the SystemC library [29] which supports advanced simulation technologies like out-of-order scheduling [16] and conflict prediction [15]. Given a parallel SystemC model, the simulator synchronizes the individual threads through the instrumented `wait` statements and prevents data and timing hazards

through the data conflict and event notification table. Advanced techniques to reduce the amount of false positive hazards are introduced in Chapter 3.

The *Dynamic Analysis* back end generates an executable which simulates until the end of elaboration (see Section 1.2) to gain structural information about the design. After completing the elaboration, the module hierarchy with the corresponding members are written into a file. This information it taken into account for a more precise analysis. The complete context of this concept is explained in Chapter 4.

The traditional Parallel Discrete Event Simulation (PDES) focuses on thread level parallelism. However, it does not utilize the support of vectorization units for data level parallelism. The *Vectorization Identifier* addresses this issue and analyzes the individual threads for data level parallelism opportunities. The output of the back end is a list of loops which are recommended for vectorization. However, this list cannot be automatically instrumented through the limitations of static analysis (e.g. overlapping arrays). The designer must add the vectorization proposals manually to the file through the limitations of static analysis. This technique is discussed in Chapter 5.

1.4 Segment Graph

The *Static Conflict Analysis* [22] identifies potential race conditions between the individual threads. In detail, we partition each thread into so-called segments. A segment considers all potentially executed statements between two scheduling steps. A new scheduling step is triggered with a `wait()` function call which gives control back to the simulation kernel. Figure 1.8 shows a SG of the source code in Figure 1.7.

```
0 void foo() {
1   r++;
2   wait();
3   a=b+c;
4   if(condition){
5     i++;
6     wait();
7     j++;
8   } else {
9     b=x+y;
10  }
11  z=z*z;
12  wait()
13  y=z+4; }
```

Figure 1.7: Example Source Code [48]

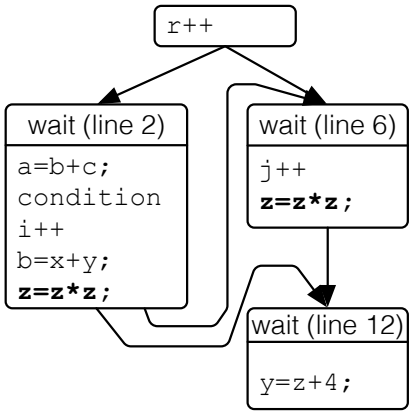


Figure 1.8: Segment Graph [48]

Algorithm 1 formally defines the central function BuildSG [48] which build the SG. Function BuildSG is recursive and builds the graph of segments by traversing the AST of the design model. Here, the first parameter *CurrStmt* is the current statement which is processed next. The set *CurrSegs* contains the current set of segments that leads to *CurrStmt* and thus will incorporate the current statement. For instance, while processing the assignment `z=z*z` in Figure 1.8, *CurrSegs* is the set {wait(line2), wait(line6)}, so the expression will be added to both segments.

Algorithm 1 Segment Graph Generation

```
1: function BuildSG(CurrStmt, CurrSegs, BreakSegs, ContSegs)
2:     if isBoundary(CurrStmt) then
3:         NewSeg ← new segment
4:         for Seg ∈ CurrSegs do
5:             AddEdge(Seg, NewSeg)
6:         end for
7:         return { NewSeg }
8:     else if isCompoundStmt(CurrStmt) then
9:         for Stmt ∈ CurrStmt do
10:            CurrSegs ← BuildSG(Stmt, CurrSegs, BreakSegs, ContSegs)
11:        end for
12:        return CurrSegs
13:    else if isIfStmt(CurrStmt) then
14:        AddExpression(IfCondition, CurrSegs);
15:        NewSegSet1 ← BuildSG(IfBody, CurrSegs, BreakSegs, ContSegs)
16:        NewSegSet2 ← BuildSG(ElseBody, CurrSegs, BreakSegs, ContSegs)
17:        return NewSegSet1 ∪ NewSegSet2
18:    else if isBreakStmt(CurrStmt) then
19:        BreakSegs ← BreakSegs ∪ CurrSegs
20:        CurrSegs ← ∅
21:        return CurrSegs
22:    else if isContinueStmt(CurrStmt) then
23:        ContSegs ← ContSegs ∪ CurrSegs
24:        CurrSegs ← ∅
25:        return CurrSegs
26:    else if isExpression(CurrStmt) then
27:        if isFunctionCall(CurrStmt) then
28:            return AddFunctionCall(CurrStmt, CurrSegs)        ▷ See Figure 1.9
29:        else
30:            AddExpression(CurrStmt, CurrSegs)
31:            return CurrSegs
32:        end if
33:    else if isLoop(CurrStmt) then
34:        return AddLoop(CurrStmt, CurrSegs)                    ▷ See Figure 1.10
35:    end if
36: end function
```

If *CurrStmt* is a boundary statement (e.g. `wait`), a new segment is added to *CurrSegs* with the corresponding transition edges (lines 2 to 7 in Algorithm 1). Compound statements are processed by recursively iterating over the enclosed statements (lines 8 to 12) while

conditional statements are processed recursively for each possible flow of control (from line 13). For example, the `break` and `continue` statements represent an unconditional jump in the program. For handling these keywords, the segments in *CurrSegs* move into the associated set *BreakSegs* or *ContSegs*, respectively. After completing the corresponding loop or `switch` statement, the segments in *BreakSegs* or *ContSegs* move back to the *CurrSegs* set.

For brevity, we illustrate the processing of function calls and loops in Figure 1.9 and Figure 1.10. The analysis of function calls is shown in Figure 1.9. In step 1, the dashed circle represents the segment set *CurrSegs*. The RISC algorithm detects the function call expression and checks if the function is already analyzed. If the function is encountered for the first time, the function is analyzed separately, as shown in step 2. Otherwise, the algorithm reuses the cached SG for the particular function. Then, in step 3, each expression in segment 1 of the function is joined with each individual segment in *CurrSegs* (set 0). Finally, segments 4 and 5 represent the new set *CurrSegs*.

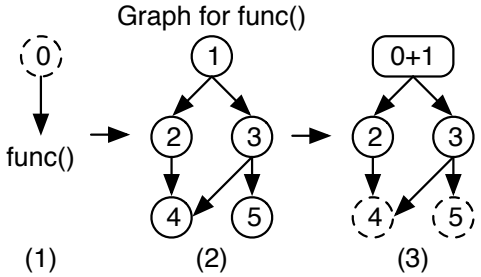


Figure 1.9: Function call processing [48]

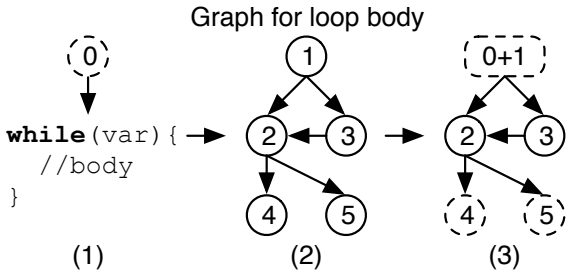


Figure 1.10: Loop processing [48]

Correspondingly, Figure 1.10 illustrates the SG analysis for a while loop. Again, the dashed circle in step 1 represents the incoming set *CurrSegs*. The algorithm detects the `while` statement and analyzes the loop body separately. The graph for the body of the loop is shown in step 2. Then each expression in segment 1 is joined into the segment set 0 and the new set *CurrSegs* becomes the joined set of 0+1, 4, and 5. Note that we have to consider

set 0+1 for the case that the loop is not taken.

Listing 1.1 shows an example SystemC model where two threads th_1 and th_2 run in parallel in modules *M1* and *M2*, respectively. Both threads write to the global variable x, th_1 in lines 14 and 17, and th_2 in line 35 since reference p is mapped to x. Before we can mark these write conflicts in the data conflict table, we need to generate the SG for this example. The SG with corresponding source code lines is shown in Figure 1.11, whereas Figure 1.12 shows the variable accesses by the segments. Note that segments 3 and 4 of thread th_1 write to x, as well as segment 8 of th_2 which writes to x via the reference p. Thus, segments 3, 4, and 8 have a write conflict. This is marked properly in the corresponding data conflict table shown in Figure 1.13.

```
1  #include "systemc.h"
2  int x = 0;
3  int y;
4  SC_MODULE(M1) {     // Module M1
5    SC_HAS_PROCESS(M1);
6    sc_event &event;
7    M1(sc_module_name name, sc_event &e): event(e)
8    { SC_THREAD(main); }
9    void main() {
10     int temp = 0;
11     while(temp++<2) {
12       wait(1, SC_MS);
13       wait(event);
14       x = temp;
15     }
16     wait(3, SC_MS);
17     x = 27;
18   }
19 };
20 SC_MODULE(M2) {     // Module M2
21   SC_HAS_PROCESS(M2);
22   int i;
23   int &p;
24   sc_event &event;
25   M2(sc_module_name name, int &pp, sc_event &e):
26     sc_module(name), p(pp), i(0), event(e)
27   { SC_THREAD(main); }
```

```
28    void main() {
29      do {
30        wait(2, SC_MS);
31        y = i;
32        event.notify(SC_ZERO_TIME);
33      } while(i++<2);
34      wait(4, SC_MS);
35      p = 42;
36    }
37  };
38  SC_MODULE(Main) {    // Module Main
39    sc_event event;
40    M1 m1;
41    M2 m2;
42    Main(sc_module_name name):
43      sc_module(name), m1("m1", event), m2("m2", x, event)
44    { }
45  };
46  int sc_main(int argc, char **argv) {
47    Main m("main");
48    sc_start();
49    return 0;
50  }
```

Listing 1.1: SystemC example with two parallel threads in modules *M1* and *M2*.

In general, not all variables are straightforward to analyze statically. SystemC models can contain variables at different scopes, as well as ports which are connected by port maps. The RISC compiler distinguishes and supports the following cases for the static variable access analysis.

1. Global variables, e.g. `x`, `y` in lines 2 and 3 of Listing 1.1: This case is discussed above and is handled directly as tuple (*Symbol*, *AccessType*).

2. Local variables, e.g. `temp` in line 10 for Module `M1`: Local variables are stored on the stack and cannot be shared between different threads. Thus, they can be ignored in the variable access analysis.

3. Instance member variables, e.g. `i` in line 2 for Module `M2`: Since classes can be

instantiated multiple times and then their variables are different, we need to distinguish them by their complete instance path added to the variable name. For example, the actual symbol used for the instance variable `i` in module `M2` is `m.m2.i`.

4. References, e.g. `p` in line 23 in module `M2`: RISC follows references through the module hierarchy and determines the actual mapped target variable. For instance, `p` is mapped to the global variable `x` via the mapping in line 43.

5. Pointers: RISC currently does not perform pointer analysis. This is planned as future work. For now, RISC conservatively marks all segments with pointer accesses as potential conflict with all other segments.

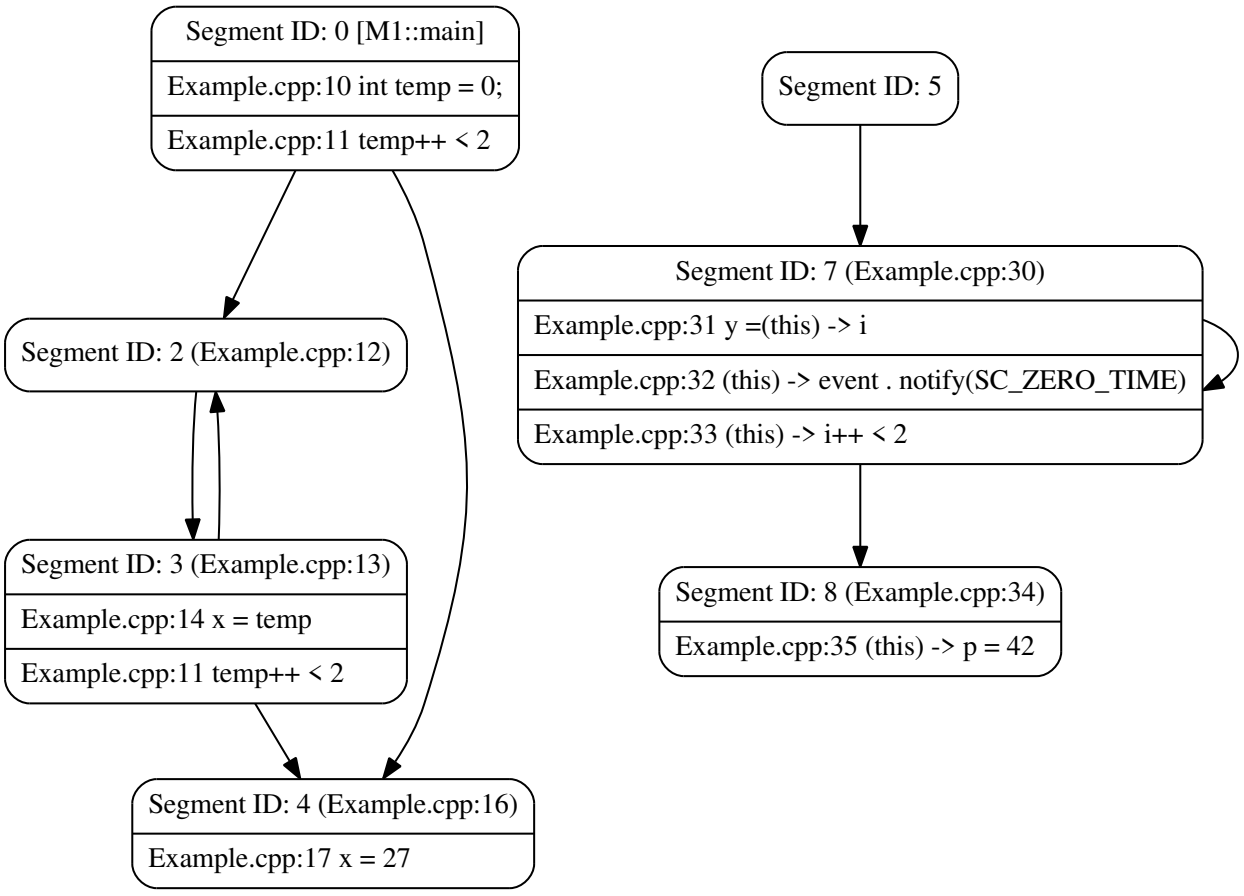


Figure 1.11: Segment graph generated by RISC for the example in Listing 1.1, source [22]

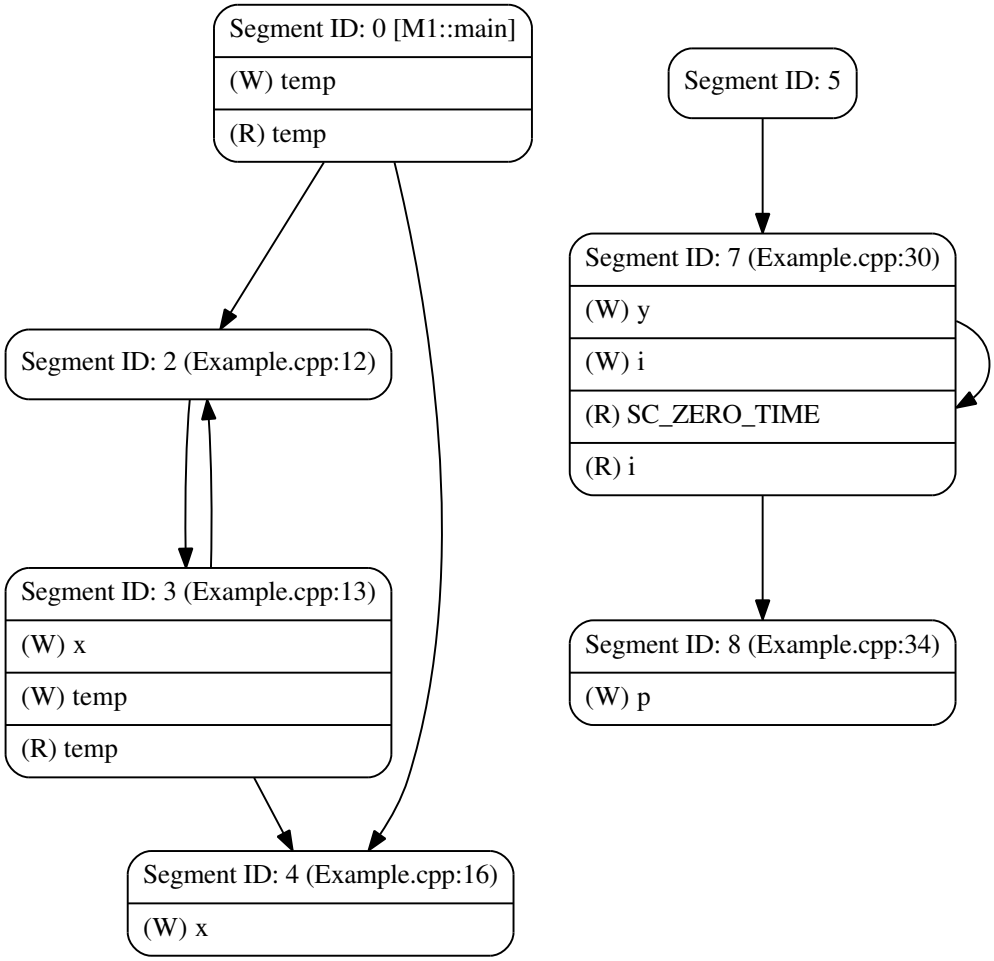


Figure 1.12: Segment graph with variable access list generated by RISC for the example in Listing 1.1, source [22]

	0	2	3	4	5	7	8
0							
2							
3			T	T			T
4			T	T			T
5							
7						T	
8			T	T			T

Figure 1.13: Data conflict table for the example in Listing 1.1, source [22]

	0	2	3	4	5	7	8
0							
2							
3							
4							
5							
7			T				
8							

Figure 1.14: Event notification tables for the example in Listing 1.1, source [22]

1.5 Goals

In this dissertation, we aim to provide a compiler infrastructure to analyze system level designs for the SystemC library. We define the following requirements for this dissertation.

- **Graphical representation**: Design exploration considers refinement, extending, and optimizing of given models. Refinement is a complex task especially the process of analyzing the communication and synchronization steps between the individual components. The RISC compiler should provide visual support to guide the designer through the refinement process.

- **Advanced conflict analysis**: The parallel simulation is based on the SG data structure which partitions threads into smaller source code segments. In the next step, the individual segments are analyzed for data and timing hazards. The necessarily pessimistic conflict analysis causes false positive conflicts to avoid simulation corruption through false negative conflicts. However, a too pessimistic analysis prevents parallel simulation and the simulator executes threads sequentially. The RISC compiler needs a more sophisticated analysis to prevent false positive channel conflicts and to handle references properly.

- **Hybrid analysis**: Design exploration requires extensive prototyping of models with different configurations to sort out misleading paths. Designers provide such configurations via command line to configure properties, e.g., the grid size of a NoC model. The static analysis takes place at compile time and consequently it cannot consider parameters which are provided at run time. The RISC compiler needs a hybrid analysis technique to process models which take command line parameters into account.

- **Vectorization**: Traditional PDES focuses on thread parallelism. Modules are executed by individual threads which execute on a set of available CPUs. We take data

level parallelism into account to simulate the individual threads at a higher speed in parallel. The RISC compiler needs to identify loops for vectorization to exploit the host simulation platform.

- **Open Source**: SystemC is an open source project which benefits from contributions of many individuals and companies. This work matches this philosophy as an open source project and as such is our contribution to the SystemC and EDA community.

The remainder of this dissertation is organized as following. In Chapter 2 [48], we introduce the concept of a Thread Communication Graph (TCG). In an ideal top-down system design flow, graphical diagrams are designed before an executable specification in a SLDL is derived. Such initial charts typically also serve as visual documentation of the textual specification and aid in maintaining the model. In the absence of graphical charts, e.g., in case of legacy or 3rd party code, a textual SLDL model is hard to comprehend for any unfamiliar designer. Here, we propose to automatically extract graphical charts from given SystemC code to ease the understanding of the source code with a visual representation. Specifically, we extract the communication flow between the threads from the design model by use of an automatic SystemC compiler infrastructure that statically analyzes the code and generates custom TCG similar to message sequence charts. Our experimental results on embedded applications demonstrate that our novel static analysis can quickly extract accurate TCG that are highly useful for designers in becoming familiar with new source code.

In Chapter 3 [47], we introduce advanced techniques to improve the precision of the static conflict analysis. Many parallel SystemC approaches expect a thread safe and conflict free model from the designer. Alternatively, an advanced compiler can identify and avoid possible parallel access conflicts. While manual conflict resolution can theoretically be more precise, it is impractical for real world applications because of the inherent complexities. Here automatic compiler-based analysis is preferred which provides conservative conflict avoidance with minimal false positives. Section 3.1 [47] introduces a novel compiler technique called

Port Call Path (PCP) analysis that greatly reduces the amount of false positive conflicts resulting in significantly increased simulation speed. Experimental results show that the new analysis reduces the amount of false conflicts by up to 98% and, on a 4-core processor, speeds up the simulation up to 3x for a NoC particle simulator and 3.5x for a Bitcoin miner SystemC model. In Section 3.2, we introduce a new technique to resolve C++ references in SystemC models. Experimental results show that these improvements reduce the amount of false conflicts by up to 98% and, on a 4-core processor, speed up the simulation by up to 3.5x for a Bitcoin miner SystemC model. Additionally, we measure a speedup of 1.51x from the parallel particle simulator with reference version to the parallel particle simulator without references.

In Chapter 4 [50], we introduce the concept of a hybrid analysis. Parallel SystemC approaches expect a thread-safe and race-condition-free model from the designer or use a compiler which identifies the race conditions. However, they have strong limitations for real world examples. Two major obstacles remain: a) all the source code must be available and b) the entire design must be statically analyzable. In this chapter, we propose a solution for a fast and fully accurate parallel SystemC simulation which overcomes these two obstacles a) and b). We propose a hybrid approach which includes both static and dynamic analysis of the design model. We also handle library calls in the compiler analysis where the source code of the library functions is not available. Our experiments demonstrate a 100% accurate execution and a speedup of 6.39x for a Network-on-Chip particle simulator.

In Chapter 5 [49], we use the RISC compiler to detect vectorization opportunities system level designs. Most parallel SystemC approaches have two limitations: (a) the user must manually separate all parallel threads to avoid data corruption due to race conditions, and (b) available hardware vector units are not utilized. In this chapter, our infrastructure exploits opportunities for data-level parallelization. Our experimental results show a nearly linear speedup of $N \times M$, where N and M denote the thread and data-level factors, respectively.

In turn, a 4-core multi-processor achieves a speedup of up to 8.8x, and a 60-core Xeon Phi processor reaches up to 212x.

1.6 Related Work

In this section, we discuss other works that are related to this dissertation. SLDLs in context of the SG haven been intensively discussed in the Lab for Embedded Computer Systems (LECS). In the first part, we review the RISC compiler framework of this dissertation which shares similarities with the SpecC environment [21]. In the second part, we discuss other works in the domain of parallel simulation and DES.

1.6.1 LECS Group

We discuss related work of the SG of the LECS group in this section. Figure 1.15 visualizes the following discussion and dependencies between the individual contributions.

The SpecC language and compiler was introduced by Dömer [21] for modeling and simulating embedded system models. The SpecC compiler uses static analysis to obtain information of a model and generates an executable for simulation. Chen extended the SpecC compiler infrastructure in many ways, however, we exclusively focus on extensions related to this dissertation. First, the SpecC compiler and simulator is extended to run simulations in parallel on the thread level [18]. This extension includes the introduction of the SG and the related data, event, and timing tables. Different *types of behaviors* (modules) and channels can simulate in parallel if the related segments are conflict free.

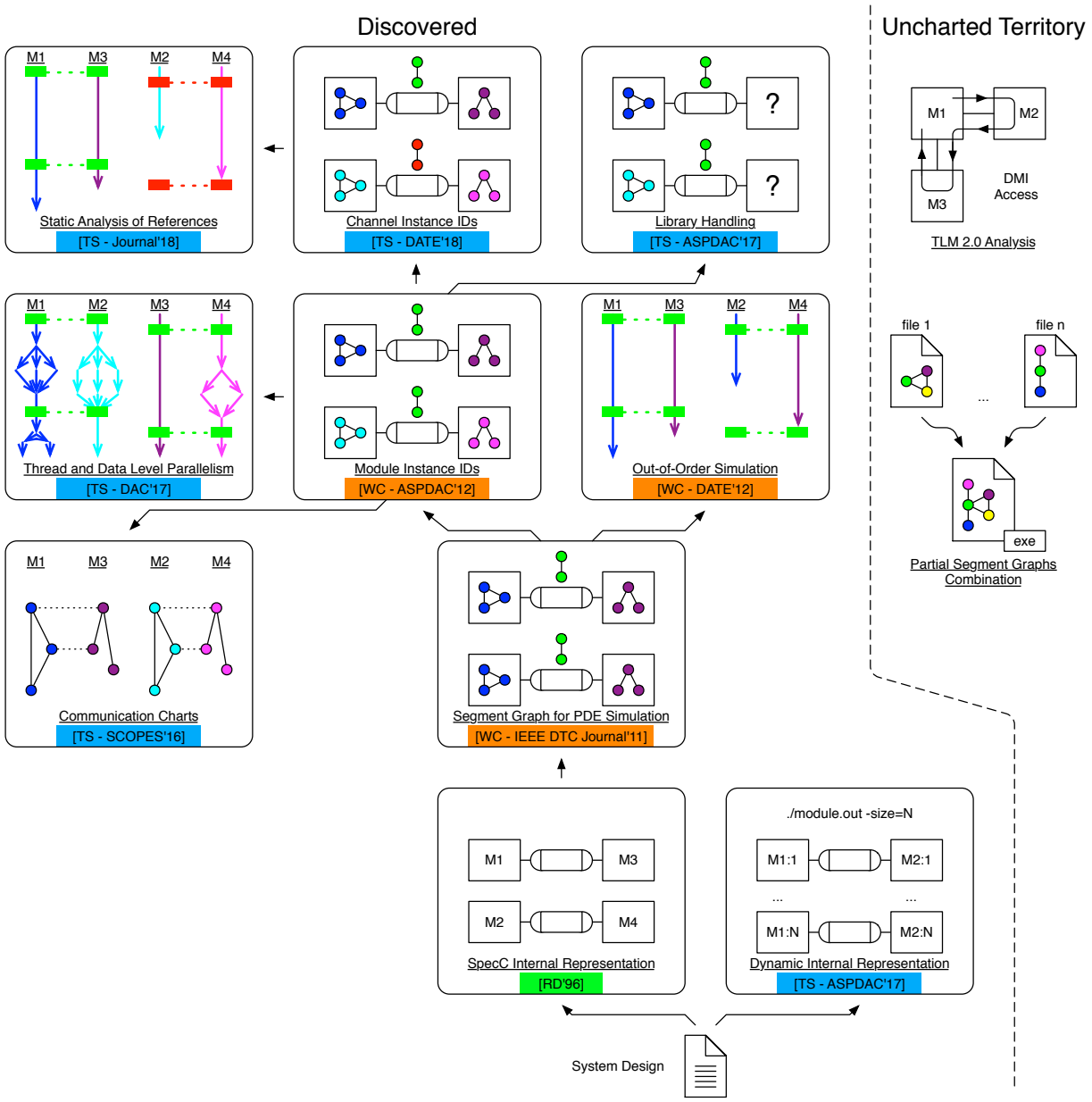


Figure 1.15: Context and overview of this dissertation

This work of PDES is extended in [14] where the concept of instance isolation is introduced. The SpecC compiler distinguishes between multiple instances of the *same module type*. Each module is associated with an individual instance analysis which is respected during the conflict analysis. Instead of comparing symbols only, we now compare symbols with the associated module instance ID. Channel variables, however, are *not* distinguished. The

reason and the needs for advanced channel instance ID analysis are discussed in Section 3.1.1.

[16] introduces the concept of out-of-order simulation where simulation threads break the time boundary. In other words, simulation threads run ahead of the global time and synchronize later when there is no upcoming conflict.

The concept of a *hybrid analysis* was introduced for the RISC compiler. Specifically, the model is pre-simulated until the end of the elaboration and writes structural information in a file. This file takes the RISC compiler into account and can perform a more precise analysis. The advanced hybrid analysis is for instance needed when modules are dynamically created in a loop. This works is published in [50] and is discussed in Chapter 4.

The work of module instance IDs is extended for *channel instance IDs* to reduce the number of false positive marked race conditions. Therefore, a new technique, the PCP, is introduced to store control flow transitions from a module to a channel. This makes it possible to identify channel instance IDs and to distinguish member variables. This work [47] is discussed in Section 3.1.1.

References can appear as member variables or as function parameter in models. Unresolved references must be marked as conflicts in the conflict analysis to avoid race conditions and generate false positives conflicts. The combination of advanced static analysis and the PCP technique makes it possible to resolve references and to determine mapped variables. This still unpublished work to reduce the number for false positives conflicts in context of references is discussed in Section 3.2.

Another work in context of static analysis and *library handling* is published in [50] and is discussed in Chapter 4. An annotation scheme for libraries is introduced which is considered by the static analysis. This scheme makes it possible to include 3rd party libraries in a thread-safe parallel simulation.

Previous works focus on simulation speed through thread level parallelism. A new approach for *thread and data level parallelism* for system level simulation is introduced in Chapter 5 and published in [49]. Each thread is analyzed for vectorizable loops in the model to benefit from vectorization units. After the analysis, the designer gets a list of recommendations for vectorization and has to decide which loops should be vectorized.

The SG is interpreted in a new way for documentation aspects in terms of *Communication Charts*. A graphical visualization makes is possible to visualize communication steps between individual elements and their dependencies. This work is discussed in Chapter 2 and published in [48].

The section *uncharted territory* in Figure 1.15 describes future opportunities to utilize the RISC compiler infrastructure for new projects. Transaction-Level Modeling (TLM) 2.0 eliminates the concept of channel and introduces the concept of direct memory access to save context switches and gain additional speedup. It should be possible to identify the individual communication patterns and prepare these models for parallel simulation as well. Another open task is the treatment of partial SGs. Currently, it is a requirement of the RISC compiler to have the entire design in one file. However, real world applications are distributed over multiple files. There is a need to generate SGs for individual files and combine them later.

1.6.2 Other Related Work

In this section we discuss a general overview of contributions in the domain of PDES. In Chapter 2, Chapter 3, Chapter 4, and Chapter 5 we compare these works more detailed and analyze them in context of the individual contribution of the dissertation.

Chen worked in the LECS group, too, and focused her research on PDES and static analysis for SpecC models. She provides a nice and comprehensive related work section in [13]. The

sections "General Programming Approaches" and "Parallel Discrete Event Simulation" are taken from Chen.

General Programming Approaches

The concept of multithreaded programming is implemented or provided through libraries in many programming languages. Posix Threads [2] [41] and OpenMP [39] are well established libraries for parallel programming in C and C++. Java has the language the dedicated keyword `synchronized` and provides the interface `Runnable` and the class `Thread`. Also there are dedicated languages for parallel programming like Erlang [8] from Erricson, Cilk [20] [52] from Intel/MIT, CUDA [38] from Nvidia, among many others.

Parallel Discrete Event Simulation

The domain of PDES is a well studied topic in the literature. A first study is provided by Chandy and Misra in 1979 [12]. Additionally, Fujimoto made a contribution for PDES in 1990 [23] and Nicol and Heidelberger for parallel execution for serial simulators in 1996 [37]. Generally speaking, PDES techniques can be categorized in two groups, namely, conservative and optimistic. Conservative techniques prevent data and timing hazards and they need advanced analysis techniques to eliminate false positive race conditions. Optimistic techniques assume no hazards and perform rollback strategies when a hazard is detected. However, rollback strategies are expensive to implement and they are time costly during simulation. The work of this dissertation belongs to the category of conservative strategies.

PDES can be performed on a local host or even distributed over a network of host machines. This approach is discussed in [12] and [27]. However, partitioning the model in the complex task and the simulation speed is limited through the network connection, too.

SystemC Related

CARH [40] is an architecture for validating system-level designs. The software documentation generation tool Doxygen[7] and other open source tools generate a XML representation of source code.

A very similar tool is the systemc-clang framework [28] for static analysis of SystemC models which generates an intermediate representation of Register-Transfer Level (RTL) and TLM designs. systemc-clang can identify communication properties (callback function name, socket name, payload information, and others) through static analysis in TLM 2.0 style. Their compiler recognizes communication function calls. PinaVM [33] is a tool which bases on LLVM to extract structural information and is inspired by [35]. Scoot [10] is a tool for type checking to gain faster simulation via code re-synthesis.

Time decoupling is a widely-used method that speeds up the simulation of SystemC models. Parts of the model execute in an unsynchronized manner for a user defined time quantum. However, this strategy is associated with inaccurate simulation results [25]. [54] and [55] propose a technique to parallelize time-decoupled designs. This technique requires the designer to manually partition and instrument the model in parts of time-decoupled zones.

A tool flow for parallel simulation of SystemC RTL models was proposed in [44]. The model was partitioned according to a modified version of the Chandy–Misra–Bryant algorithm [12]. In contrast, our conflict analysis considers the individual statements of threads.

The authors in [53] describe an API to manually transform a sequential SystemC design into a parallel design. An approach of static analysis for simulation of SystemC models on GPUs was provided in [51]. Bombieri [11] proposes an automated transformation of RTL applications to GPUs.

Chapter 2

Static Communication Graph Generation

In this chapter, we use the introduced RISC compiler infrastructure (see Section 1.3) and the SG data structure (see Section 1.4) to generate visual representation of SystemC models. The new introduced TCG is helpful to become familiar with 3rd party source code and to represent graphically the individual communication and synchronization steps between models.

2.1 Introduction

A picture is worth a thousand words. In the absence of a picture or graphical charts, the model of a system described in a SLDL is hard to comprehend. For the system designer who faces legacy code that needs to be reused and revised, identifying essential design elements, such as the main design modules, threads and their communication patterns, becomes a tedious and lengthy task. Before adjustments, improvements or extensions of the model can

be applied, the existing source code needs to be read, analyzed and fully understood. Without documented schematic charts, such reverse engineering can require months of unproductive time.

In this chapter, we propose an automated technique to quickly generate graphical charts from SystemC source code that can help the system designer in understanding third party or legacy models. Our automatic chart generator, which is based on sophisticated static code analysis by a novel SystemC compiler, quickly produces module hierarchy trees and multi-thread communication charts that assist the designer to puzzle out and grasp the intricacies of the complex source code. As such, this work provides a powerful resource in getting familiar with legacy or third-party SystemC code.

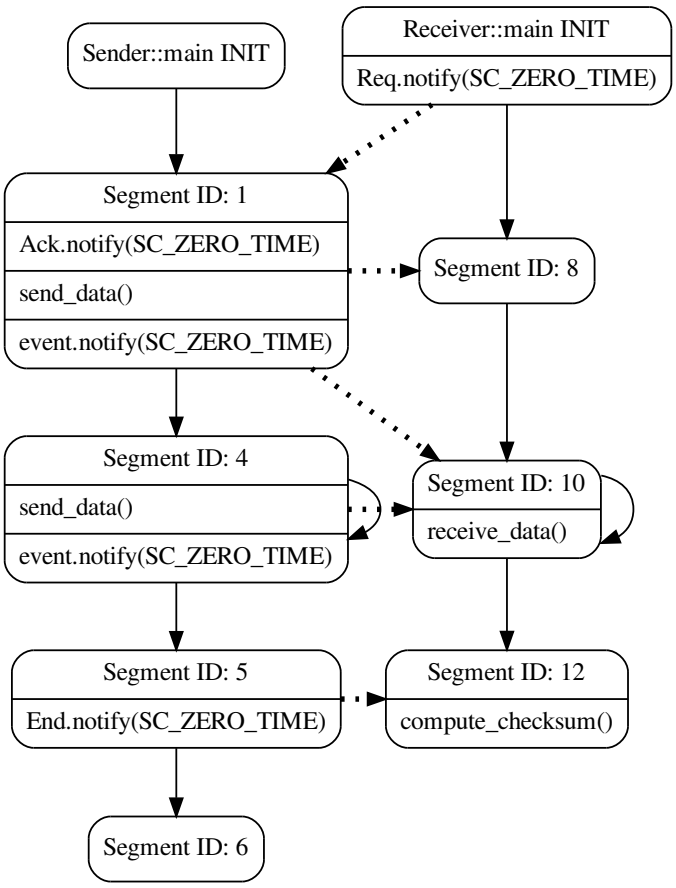


Figure 2.1: Generated TCG from SystemC code

Figure 2.1 shows an example TCG that we generated automatically from original SystemC source code for a model where a pair of modules obviously communicates in producer-consumer fashion. Instead of reading the 586 lines of source code distributed over 3 files that make up the design model, a simple glance over the quickly generated figure reveals the essential communication protocol used between the modules Sender and Receiver. A closer look at the chart then shows that events Req, Ack, and End are used to synchronize the communicating parties, then data is exchanged in a loop, and finally a checksum is computed. The automatically generated TCG clearly saves the designer precious time in understanding the model.

The key contributions of this chapter are as follow:

- We developed a dedicated compiler to analyze the structural and behavioral aspects of legacy SystemC code.
- We integrated the SystemC DES semantics in the analysis.
- We designed a novel algorithm to extract communication patterns and illustrate them in form of a chart.
- We demonstrated the capabilities of our project on a 3rd party library and an abstract AMBA bus model.
- We contribute our SystemC compiler to the open source community.

2.1.1 Related work

Static analysis of source code has been discussed in various works. CARH[40] is an architecture for validating system-level designs. The software documentation generation tool Doxygen[7] and other open source tools generate a XML representation of source code. Our work differs in that we analyze and identify the structural and behavioral aspects of the model. Specifically, we focus on the communication pattern in a given design. We utilize the knowledge of DES semantics to achieve a deeper recognition of the design. In comparison, general purpose tools like Doxygen cannot handle this task. The missing sensitivity to the SystemC semantics does not allow deeper analysis. Doxygen is familiar with C++ constructs like classes, templates, functions, and other concepts. However, the tool is not trained to analyze modules, channels, and event notifications and cannot extract communication.

A very similar tool is the systemc-clang framework [28] for static analysis of SystemC models which generates an intermediate representation of RTL and TLM designs. systemc-clang can

identify communication properties (callback function name, socket name, payload information, ...) through static analysis in TLM 2.0 style. Their compiler recognizes communication function calls. The attributes of the communication type are analyzed through the function parameters. However, we are analyzing the port binding, linked channels, and communication peers as well. Thus, our approach is designed for static analysis of structural and behavioral communication charts among the threads in the model.

PinaVM [33] is a tool which bases on LLVM to extract structural information. This work is inspired from [35]. Scoot [10] is a tool for type checking to gain faster simulation via code re-synthesis.

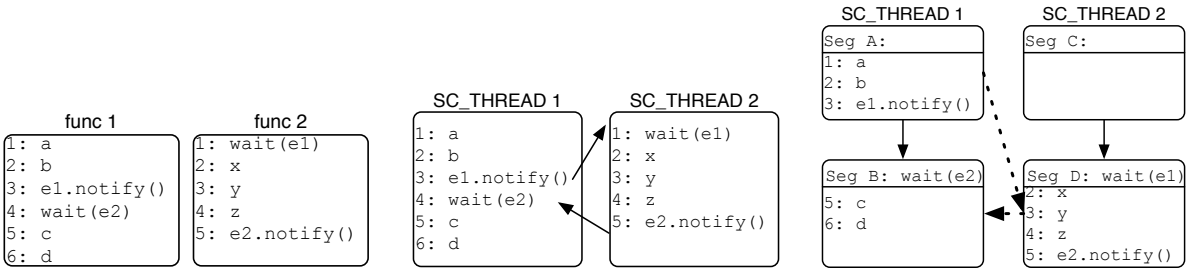
A SG was first proposed in [16] for out-of-order scheduling to speedup simulation in context of the SpecC language, and later [17] to detect race conditions and parallel execution conflicts. In contrast, we are generalizing the concept of the SG for SystemC and aim at extracting communication graphs from source code.

The rest of this chapter organized as following. We introduce the TCG in Section 2.2. Following, in Section 2.3 we explain how to generate a TCG from a SG. Our experiments are presented in Section 2.4, followed by a summary and future work in Section 2.5.

2.2 Thread Communication Graph

The design process of embedded systems typically requires combining various in-house modules and third party components, each of which the system designer needs to become familiar with. Minimizing the time to study new parts is critical, so the designer needs to focus on the essential aspects, including the communication and causal chain of composed components. Tools like Doxygen can generate call graphs to illustrate the software function hierarchy. However, these tools are agnostic to the system design semantics. To be effective, SystemC

constructs for structure and communication need to be recognized and properly represented.



(a) Two functions (b) Threads and events (c) DES dependency

Figure 2.2: Analysis of communicating threads

The problem of missing semantic analysis is illustrated in Figure 2.2. Two functions are shown in Figure 2.2a, independent entities, as software tools would see them. When SystemC semantics are applied, threads and events can be identified and the chart in Figure 2.2b can visualize the synchronization between the two threads. Even more so, we can utilize the knowledge of DES semantics and explicitly show the communication and timing dependencies between the two threads.

Figure 2.2c shows the resulting graph when `wait()` is correctly represented as a construct that incurs a delay due to the underlying multi-thread scheduling. For the remainder of this chapter, we will refer to this as a *segment boundary* that separates the *segments* of code that SystemC semantics imply as being executed without interruption. We will formally define the corresponding SG below in Section 2.3.

In Figure 2.2c, the solid edges show the flow of thread execution over individual segments, whereas the event notification dependencies are indicated by dashed lines. Here, segment A notifies event `e1` for which segment D is waiting. Thus, the designer can see immediately that segment D must be executed before segment B.

Please note that for this analysis references and port mappings of events must be resolved as well. For this, our approach generates an instance tree over the entire design hierarchy so

that shared variables can be correctly disambiguated.

2.3 Static Compiler Analysis

At the core of our SystemC visualization is the RISC, an advanced compiler framework for analyzing, executing and instrumenting SystemC models. The fundamentals of the RISC infrastructure and the SG are introduced in Section 1.3 and Section 1.4.

2.3.1 Thread Communication Graph

Based on the generated SG, we then extract the TCG to aid designers who face legacy code that needs to be reused and revised. For this, we identify and pair the synchronization and communication points in the individual scheduling steps in the design.

The SG already determines which code elements are potentially executed in any given scheduling step. However, we have to add the edges for the identified synchronization and communication points. Specifically for event notifications, we have to analyze the `notify()` and `wait()` function calls in each segment. Additionally, we need to identify any channel communication calls, e.g `read()` and `write()`. Finally, the mapped channels and events are followed and matched through the design hierarchy.

Port Mapping

SystemC ports provide a flexible interface to send and receive data via mapped channels (or other mapped ports). While the indirect function calls via ports to channel methods are a powerful modeling feature, it is difficult to follow the actual flow of control in unfamiliar code. Here, our RISC compiler can help and determine which port is mapped to which

channel, including for cases where this mapping goes through multiple levels of the design hierarchy.

To resolve port mappings, two steps are required. First, a port needs to be unambiguously identified in the hierarchy of the design. Second, we have to follow the module hierarchy to find the mapped channel.

For step 1, we identify a port in the design through a so-called *instance path*. An instance path is a list of tuples where each tuple contains a scope and an instance. For example, the path to a port DataIn could be [GlobalScope::top] → [Top::platform] → [Platform::datain] → [DataIn::port1]. Note that the instance path uniquely identifies a port, even if there are multiple instances of this port in the module hierarchy.

For step 2, we use the instance path to identify the mapped channel. Specifically, we analyze the mapping between a tuple and its successor. Here, we check the module constructor and identify the mapping to a channel or another port, and repeat the process as needed. In each iteration we go up in the instance tree until the port is mapped to a channel.

Event and Reference Mapping

SystemC threads use events for synchronization with each other. If the synchronizing threads are in the same module, a shared event variable in the module can be used. However, if threads have to synchronize across module boundaries, an event at a higher level in the hierarchy is needed, which is typically mapped via references.

Here, we can determine the reference mapping in the same fashion as the port mapping. We describe the event by an instance path and go up the path until the reference is mapped to an actual variable.

Handling of Loops

Loops are naturally present in virtually all algorithms and clearly need to be supported by our source code analysis. However, general loops can also lead to complex control flows that are difficult to represent cleanly in visual graphs. For the purpose of our TCG, we aim at a loop abstraction that simplifies the understanding of the protocol between the communicating parties. Specifically, our goal is to visualize the overall sequential flow of exchanged messages, in similar manner as exhibited by message sequence charts[34].

For our TCG, we support loops in one of two ways. On the one hand, we can assume that each loop will be taken at least once. On the other hand, loops can be unrolled. By default, our TCG assumes the first option because not every loop can be unrolled. Additionally, loop unrolling can lead to state explosion and often decreases the readability of the TCG.

In context of communication protocols, it is reasonable to assume that the sender and the receiver are exchanging messages equally. Listing 2.1 shows an example where `thread1` notifies `thread2` ten times. Here, the sender and the receiver are exchanging messages and thus the associated `notify()` and `wait()` functions must be called equally often. Figure 2.3 shows the TCG for the example in Listing 2.1. The graphs shows that the loop is taken at least once.

```
1 void thread1() {
2   for(int i = 0; i < 10; i++) {
3     event.notify(SC_ZERO_TIME);
4     /* Do some stuff */
5     wait(SC_ZERO_TIME); } }
6 void thread2() {
7   for(int i = 0; i < 10; i++) {
8     wait(event);
9     /* Do some stuff */ } }
```

Listing 2.1: Producer and consumer example

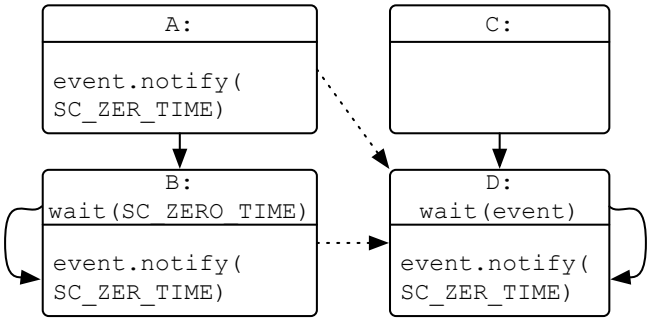


Figure 2.3: Example of a loop with synchronization

2.3.2 Module Hierarchy

We can use the SystemC IR to determine the module hierarchy of the design model in two steps. First, we identify the top module of the design. Unless explicitly specified by the designer, we assume that the top module is declared in the function `sc_main`. From the declaration, we can then derive the module definition. Finally, we can traverse the hierarchy of the design as described in Listing 2.2.

```
1 traverse_hierarchy(ModuleDefintion md) {
2   foreach(ModuleInstance mi
3       in get_all_sub_mdules(md)) {
4     traverse_hierarchy(get_module_definition(mi))
5   }
6 }
```

Listing 2.2: Traversing the module hierarchy

The function `traverse_hierarchy()` takes the module definition *md* and iterates over all its child modules. For each child module instance *mi*, the corresponding module definition is determined and the function `traverse_hierarchy()` is called recursively.

2.3.3 Optimization and Designer Interaction

The system designer has a set of configuration options and parameters available to get a better picture of the design. For example, the designer can select what types of edges should be displayed for the communication. Our TCG distinguishes native event notifications, primitive, and hierarchical channel communication.

The system designer can also select only a subtree of the module hierarchy or a subset of the SystemC threads for which the TCG will be generated. Thus, the designer can easily choose and focus on the points of interest and see the communication and dependencies between them.

Finally, pseudo comments may be used to indicate loop unrolling and `wait` statements can be annotated and labeled. This information is then displayed in the generated graph for enhanced readability.

Overall, the system designer can quickly and iteratively generate custom charts, getting more familiar, and obtaining a better picture of the model and its components.

2.3.4 Visualization

Our RISC compiler performs the analysis and graph generation based on the internal representation of the model and generates DOT files [6] for the SG and TCG. These files can then be visualized by the DOT tools, e.g. as interactive chart on screen or as PDF.

2.3.5 Accuracy and Limitations

The current implementation of our fully automated compiler has some limitations. We cannot handle pointers. Also, currently we cannot match array indices in port mappings.

Our compiler produces charts which are giving an impression of the communication behavior. However, static analysis can misinterpret situations and illustrate too many or too few communication edges. As mentioned, a picture is worth a thousand words (and even if a few words are inaccurate, the picture helps a lot in quick comprehension). For instance, in Figure 2.7 we can see immediately that the master $m1$ requests the bus through the arbiter. After the arbiter acknowledges the request, master $m1$ starts communicating to the slave. We should emphasize that our TCG generator is fully automated and quickly visualizes identified communication patterns without designer interaction. The generated illustrations may be inaccurate in minor aspects (limitations listed above), but they nevertheless convey an overall image that is helpful for getting to understand the source code quickly.

2.4 Experiments

We have evaluated our TCG generation from SystemC source code on a Mandelbrot graphics application, an AMBA bus model, and the S2CBench benchmark set. For all examples, we tested both our hierarchical and communication analysis.

2.4.1 Mandelbrot Renderer

The Mandelbrot example computes a stream of Mandelbrot [32] images and as such is a representative for highly parallel graphics applications. The source code is complex, heavily instrumented with macros for customization. Here, we choose 2 parallel renderer modules.

The Mandelbrot module hierarchy is shown in Figure 2.4.

```
Top top
    +- DataChannel c1
    +- DataChannel c2
    +- Stimulus stimulus
    +- Platform platform
       +- DataIn din
       +- DUT dut
          +- main (thread)
          +- mb1 (thread)
          +- mb2 (thread)
          +- mb3 (thread)
          +- mb4 (thread)
       +- DataOut dout
    +- Monitor monitor
```

Figure 2.4: Module hierarchy of the Mandelbrot

We can deduce that the modules *Stimulus* and *Monitor* feed data in and out of the *Platform* and in turn the Device Under Test (DUT) (via *DataIn* and *DataOut*). The DUT hosts one main and four worker threads *mb1*, *mb2*, *mb3*, and *mb4*.

Next, we performed behavioral analysis and generated TCG, such as Figure 2.5. The *INIT* segments start threads and the solid arrows illustrate the transitions between the segments. The dashed arrows show the event synchronization. The generated Figure 2.5 focuses on the synchronization between the main and worker threads in the module DUT. We can see that the thread *main* notifies the worker threads *mb1*, *mb2*, *mb3*, and *mb4* when data is available and both respond back to *main*.

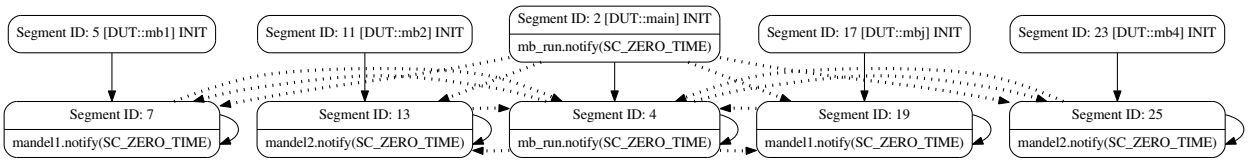


Figure 2.5: Mandelbrot TCG for DUT

Figure 2.6 shows a higher level of communication analysis, where our RISC compiler first analyzed the module hierarchy and channel binding to identify the port mapping. Then it followed the port mapping through the different module levels and associated them with the corresponding `read()` and `write()` function calls. The resulting communication and data flow is generated in Figure 2.6. We can easily see that the data flows from *Stimulus* via *DataIn* into the DUT where the coordinates are processed. An image is then sent via *DataOut* to the *Monitor* module.

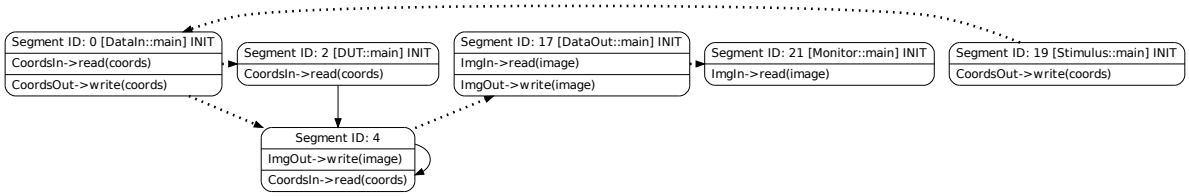


Figure 2.6: Generated Mandelbrot TCG with data flow from Stimulus via DUT to Monitor

For both diagrams the assumption that loops are taken once fits very well. The combination of the structural and behavioral analysis provides quick insight into the behavior of the design.

2.4.2 AMBA Bus Model

As an example with complex multi-component communication, we reimplemented an AMBA bus model from [43] at TLM abstraction. The generated module hierarchy is shown in Figure 2.8.

The corresponding TCG for a BWRITE operation is shown in Figure 2.7. Here, the red dashed lines represent event notifications and blue dashed lines communication via channels. We can clearly see that master `M1` requests the bus through the arbiter via the event `areq1`. In turn, the arbiter grants the bus to `M1` via the `agnt0` event. Next, `M1` uses the bus to send data via `BD` to the `Slave`, which receives the `address` via the `Decoder`. Without this chart, the designer would need to read and study the 498 lines of code and manually figure out the port mappings and execution dependencies.

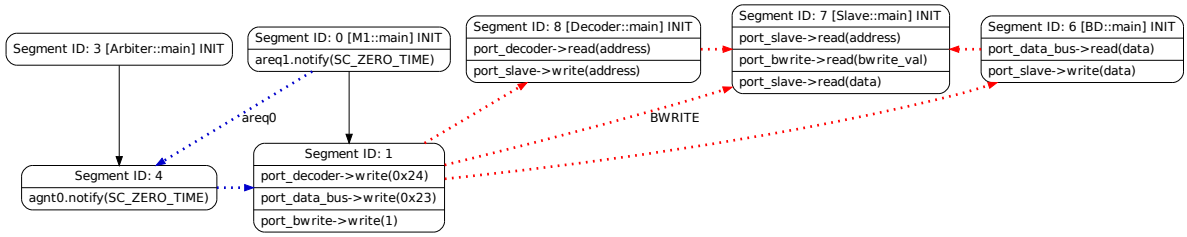


Figure 2.7: Communication Graph generated from an AMBA bus model for a BWRITE operation

2.4.3 S2C Testbench

Finally we have evaluated our TCG generation on the S2CBench [45], a benchmark suite of 16 synthesizable SystemC models that includes industrial, automotive, security, telecommunication, and consumer applications. Our RISC compiler accurately generates the module hierarchy and TCG for these examples[1]. Due to space limitations, we unfortunately cannot present the resulting graphs here.

Each TCG was generated in less than 34 seconds, where on average the compiler spent 15 seconds on AST creation (ROSE parser), 10 seconds for SystemC IR generation, and 9 seconds on the TCG.

[1] For six of the 16 benchmarks, we manually replaced indexed array variables with regular ones, since our RISC compiler currently cannot match array indices in port mappings.

```
Top top
    +- Channel master1_to_decoder
    +- Channel master2_to_decoder
    +- Channel address_bus_to_decoder
    +- Channel master1_to_data _bus
    +- Channel master2_to_data _bus
    +- Channel data_to_bus _to_slave
    +- Channel decoder_to_slave
    +- Channel master1_to_slave _bwrite
    +- Master1 m1
    +- Master2 m2
    +- Arbiter arbiter
    +- DataBus databus
    +- Slave slave
    +- Decoder decoder
```

Figure 2.8: Module hierarchy of the AMBA bus

2.5 Summary

Becoming familiar with an unknown SystemC design is an often necessary and complex process. Designers have to identify communication patterns between threads and the behavior of individual components which can consume weeks of unproductive work.

In this chapter, we propose the RISC compiler framework to analyze and identify structural, behavioral and communication aspects of SystemC models. Specifically, we automatically extract execution, synchronization and communication dependencies and visualize them quickly as TCG.

The experimental evaluation of our framework using more than a dozen SystemC examples

from different application domains shows that the automatically generated module hierarchy and visual communication charts are very helpful for system designers in becoming familiar with legacy or third party source code.

Chapter 3

Static Analysis for Reduced Conflicts

The RISC compiler infrastructure transforms sequential IEEE SystemC models to parallel executable models. The data and event analysis module in RISC compiler front end (see Figure 1.6 and Section 1.4) determines which segments (see Section 1.4) can be executed in parallel and stores this information in tables. False positive table entries prohibit an effective parallel simulation and increase the simulation time. In this section, we address how to reduce false positive table entries through channel variables in Section 3.1 and false positive table entries through unresolved references in Section 3.2.

3.1 Channel Analysis

In this section, we introduce the PCP analysis technique to reduce the number of false positive table entries. We evaluate the importance of this technique for a audio and video streaming application, a Mandelbrot renderer, a Bitcoin miner application, and a NoC particle simulator.

3.1.1 Introduction

Embedded systems are part of our daily life, visible as a cell phone or invisible in a combustion engine in a car. The high demand of constantly increasing functionality of these products is associated with more complex design processes. Designers use simulations as a tool to make better design decisions for their prototypes. However, simulations of complex systems are time-consuming and become a bottleneck in the tool flow.

SystemC [3] has been established as the de-facto and official Accellera standard for modeling and simulating of embedded systems. The official IEEE proof-of-concept simulator runs simulations in a sequential fashion. This means only one simulation thread is active at any time during the simulation, also if many design parts could be simulated in parallel. Consequently, the simulator can use at most one core on multi and many-core host simulation platforms.

SystemC TLM 2.0 is a library-based approach to speed up the simulation where simulation threads are temporally decoupled. Specifically, the designer defines manually a time quantum in which a thread performs without performing any synchronization points. Unfortunately, the gained speedup comes with the disadvantage of simulation inaccuracy [25]. Other works such as [53] and [54] provide a parallel simulation kernel where the user has to manually translate the sequential design into a parallel design. In detail, the individual simulations threads must be analyzed for conflicting variable access. An overlooked conflict can compromise the simulation and let the simulation fail.

Problem Definition

A parallel SystemC simulator needs the information which segments of a simulation thread can be executed in parallel. One option is to provide this information in form of a conflict

table. A table entry must be *true* if two segments have a potentially conflicting variable access, otherwise the entry should be *false*, allowing parallel execution.

More specifically, a table entry is classified into four categories: true positive (TP), true negative (TN), false positive (FP), and false negative (FN). The first two categories TP and TN describe a correctly marked positive variable access conflict, respectively, no access conflict. A FP is a table entry where a conflict is marked, however, no actual conflict exists. A FN is a table entry where no conflict is marked, however, a conflict exists. This kind of a table entry is not allowed because it compromises the simulation.

The conflict table generation can be done in two ways, manually or automatically. On one hand, the manual analysis can eliminate many FP and FN entries because the designer can utilize application knowledge. However, this analysis is very time consuming and is not applicable for real world examples. On the other hand, the compiler driven automatic analysis can be done in a few minutes. However, a compiler cannot be as precise as the manual analysis, e.g. due to static pointer analysis. Consequently, a compiler must be *conservative* to safeguard the simulation correctness. In other words, table entries are marked as a potential conflict where actually no conflict exists. This behavior causes FP entries and limits the parallel simulation performance.

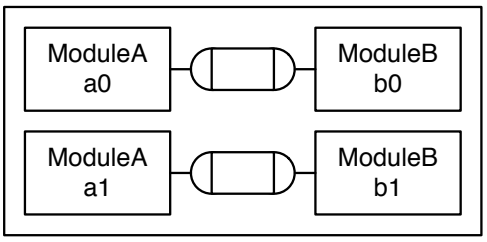


Figure 3.1:
Two pairs of communicating modules

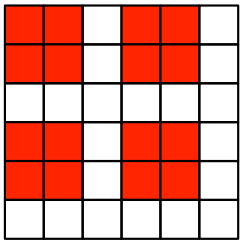


Figure 3.2:
Data conflict table

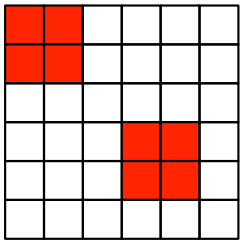


Figure 3.3:
PCP-sensitive data conflict table

Figure 3.1 shows an example where two pairs of a sender and a receiver exchange data. The

corresponding tables of conflicting variables accesses (red filled) are shown in Figure 3.2 and Figure 3.3. The left table is created with a conservative conflict analysis. Often, the parallel simulator can run only one simulation thread because of FP conflicts. The right table is built with a more precise conflict analysis. In this simple example 50% of the positive marked conflicts are FP conflicts. Consequently, the simulator can run more threads in parallel. Our RISC compiler [30] instruments the conflict table in design file. Later, the parallel SystemC simulator uses the conflict table for scheduling decisions.

In this chapter, we extend the static analysis capability of the SystemC compiler [30] to largely reduce the FP entries in the conflict table. Specifically, we introduce the PCP technique to have more precise context information of accessed variables. During the needed SG analysis, we take the related port call history into account. This information allows us to distinguish between individual variables in channels instead of clustering them.

Related Work

Parallel discrete-event simulation (PDES) [23] is a well-studied subject. The SG data structure for PDES was first introduced for synchronous and out-of-order fashion in [16]. This concept is extended with instance IDs for modules to have higher precision in the conflict analysis in [14]. An extension for thread and data level parallelism is in [49]. In contrast to these works, our new PCP technique for SGs allows a more precise analysis for variable accesses in channels to reduce FP conflicts. Also, in contrast to [50], our analysis is purely static.

The concept of time decoupling is implemented in the SystemC TLM 2.0 library where threads execute for a user defined quantum in an unsynchronized manner. The missing synchronization between threads limits the number of context switches to gain higher simulation speed. However, the simulation boost is associated with the price of lower accuracy [25]. The

works in [54] and [55] propose techniques to parallelize time decoupled designs for multi-core systems. The designer must manually partition the design into a thread safe model which is conflict free. So an overlooked conflict can compromise the simulation and leads to simulation failures. The authors in [53] describe an API to transform manually a sequential SystemC design into a parallel design. Compared to these works, our compiler driven approach automatically identifies race conditions and instruments the design. In other words, the transformation does not require application-specific knowledge or manual modeling.

In [51] static analysis of SystemC models is done to use GPUs as a simulation platform. In contrast, our static analysis uses module and channel instance IDs to distinguish channel variables. Other works in context of parallel SystemC are in [44]. They are using a modified version of the of the Chandy-Misra-Bryant algorithm for their analysis. In comparison, we provide a new analysis to have precise analysis for object instances.

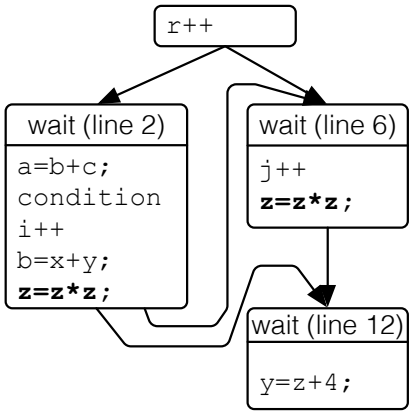
3.1.2 Conflict Analysis

Segment Graph

As we have already discussed in Section 1.4, the SG is a data structure to partition the individual simulation threads of a design in smaller pieces. Later, we use this graph to identify potential race conditions and notification dependencies in the design. In detail, for each simulation thread we partition the source code into segments and link them respectively to the control flow. A segment includes all statements between two scheduling steps. The transition from the application domain back to the scheduler domain happens through a `wait()` function call. Figure 3.4a shows some example source code and the related SG in Figure 3.4b.

```
0 void foo() {
1   r++;
2   wait();
3   a=b+c;
4   if(condition){
5     i++;
6     wait();
7     j++;
8   } else {
9     b=x+y;
10  }
11  z=z*z;
12  wait()
13  y=z+4; }
```

(a) Source Code



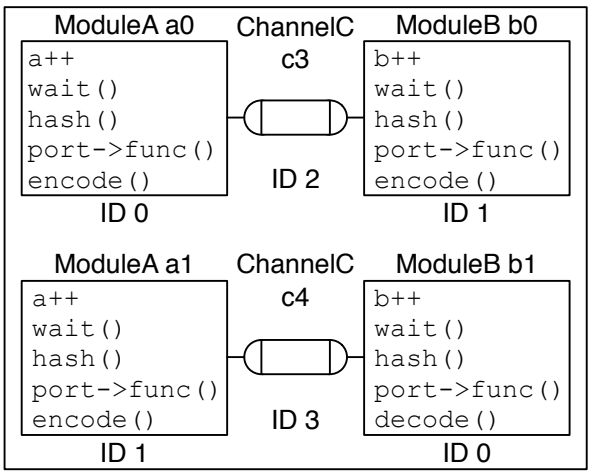
(b) Segment Graph

Figure 3.4: Example of a Segment Graph [49]

Figure 3.4b shows the difference to the traditional control flow graph and a SG. The SG has four segments, one initial and three related to `wait()` function calls. The statement in Line 11 `sqr=sqr*sqr` appears in the two segments from Line 2 and Line 6. This is possible because both can reach Line 11. Note that a SG is generated per module definition and not per module instance.

Variable Access Analysis

The data conflict analysis compares all pairs of segments for potential race conditions. Basically, two steps are needed to identify a conflict between two segments. First, we create two sets of read and written variables for each segment. In particular, we traverse each expression of a segment and identify the accessed variable symbols. Second, we have to check if there is a read-write or write-write dependency between the two segments. For instance, in Figure 3.5a, the first segment of ModuleA and ModuleB contains only the symbols of `a` and `b`, respectively. Consequently, the segments do not cause a race condition and can be simulated in parallel.



(a) Module details

```
SC_MODULE(Top) {
  ChannelC c1; // ID 0
  ChannelC c2; // ID 1
  ChannelC c3; // ID 2
  ChannelC c4; // ID 3
  ModuleA a0; // ID 0
  ModuleA a1; // ID 1
  ModuleB b1; // ID 0
  ModuleB b0; // ID 1
  ...
};
```

(b) Source code

Figure 3.5: Additional module details of Figure 3.1

The situation is different for the module instances `a0` and `a1`. The SG is generated per thread and not per instance. Consequently, both instances are represented by the same SG. So, the symbols `ModuleA::a` and `ModuleA::a` are in conflict because the SG stores only the accessed symbol. This problem can be solved through *module instance IDs* [14]. Each module has a unique ID which is defined due to the declaration order and its type in the source file. Figure 3.5b shows the declaration and related IDs for the design in Figure 3.5a. Now, the conflict analysis considers segments in context of a module instance ID. So we can distinguish between the module instances and their members.

Limitations of Module Instance IDs

Modules communicate to other modules via channels to exchange data. Technically, ports are the gateways to channels and they are the linking component between module and channel. So, through a *port call*, the control flow leaves the scope of a module and enters the scope of a channel. Exemplary, Figure 3.6 shows the transitioning control flow of a simulation thread from a module into a channel and back.

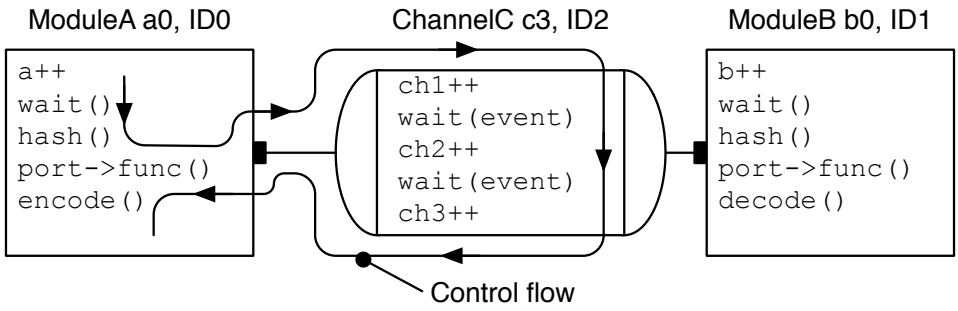
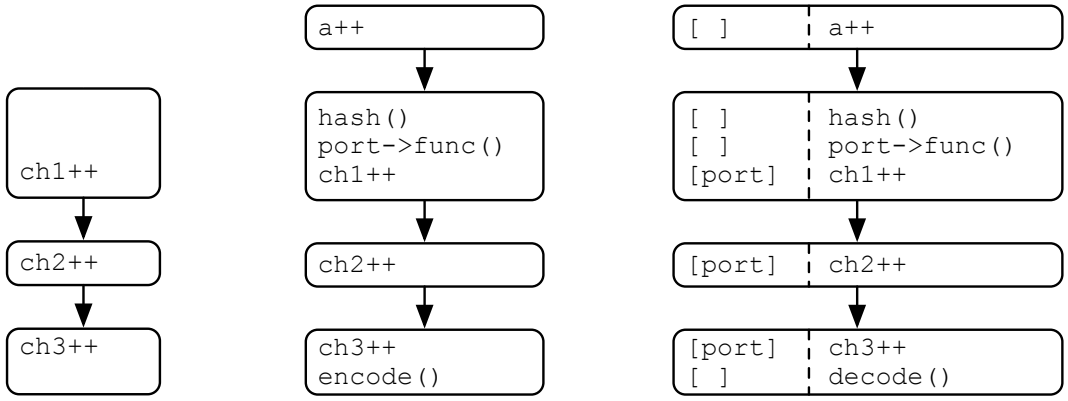


Figure 3.6: Additional channel details of Figure 3.5

A segment contains symbols of module and channel members at the same time. The union of these symbols occurs only because segment boundaries are defined as entry points to the scheduler. So, a port call does not start a new segment and is interpreted as a regular function call. Consequently, a segment contains the union of expressions which come from modules and channels. This is demonstrated in Figure 3.6 where a module communicates through a channel. The associated SG in Figure 3.7b shows expressions from the module (e.g. `hash()`) and channel (e.g. `ch1++`) in the second segment.



(a) Channel only

(b) Without Port Call Path

(c) With Port Call Path

Figure 3.7: Segment Graph of Figure 3.6 for ModuleA

The conflict analysis in context of channels with only module instance IDs is more challeng-

ing. In detail, the function HasAccessConflict($\text{Seg}_1, \text{Id}_{S1}, \text{Seg}_2, \text{Id}_{S2}$) determines if two segments have a conflict. The instance ID parameters are the module instance IDs where the simulation thread of the segment is spawned. So, all segment members are interpreted with the given module instance ID. However, this interpretation is not valid for channel variables. In Figure 3.6, module `a0` and module `b0` are bound to channel `c3`. Segment 2 starts after the `wait()` call in ModuleA and includes the channel expression `ch1++`. Similarly, Segment 4 starts after the `wait()` call in ModuleB and also includes the channel expression `ch1++`. Segment 2 interprets the variable `ch1++` with instance ID 0 and segment 4 interprets the variable `ch1++` with instance ID 1. This gives the impression that both modules are not connected to the same variables. So, the result would be a false negative conflict which is not allowed. Correctly, the variable `ch1++` must be interpreted with instance ID 2 following the declaration order in Figure 3.5b.

The SG stores only the read and write access of the potentially used symbols of a segment. A naive approach is a scope analysis of each symbol. If a symbol is declared inside a channel, the instance ID will be mapped to a dedicated instance ID for all channels. However, this strategy has two strong limitations for the speed of parallel simulation.

First, all channel variables get the same instance ID. So, all channel variables are in conflict with each other. As a result, the parallel communication between modules happens in a sequential fashion for the entire simulation.

Second, the sequential communication affects the parallel communication of modules. The segment where the port call takes place and the segment where the port call returns inherit the conflicts from the channel. This is illustrated in Figure 3.6 and Figure 3.7b where ModuleA communicates via ChannelC. The second segment includes the function `hash()` as well as the increment expression of the channel member `ch1++`. The last segment includes the function `encode()` and the increment expression of the channel member `ch3++`. Both segments can only be executed in parallel to other segments when there is no other

communication during that simulation cycle.

3.1.3 Port Call Path Sensitive Segment Graphs

Our proposed PCP analysis for channels allows to distinguish individual channels through *channel instance IDs* without modifying the design. In other words, the two communicating parties in Figure 3.5a can then execute in parallel without interfering with each other.

Port Call Analysis

For this advanced approach, the conflict analysis needs more context information. So, we store for each expression additionally the PCP history. The SG with and without PCP for the example Figure 3.6 is shown in Figure 3.7b and Figure 3.7c. Particularly, the PCP can be a single port call or a list of port calls, e.g. through hierarchical channels communication.

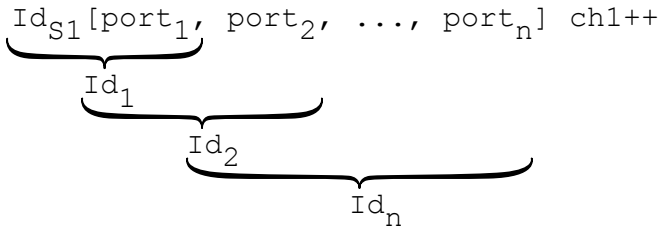


Figure 3.8: Translation of a module instance ID to channel instance ID

The function HasAccessConflict($\mathrm{Seg}_1, \mathrm{Id}_{S1}, \mathrm{Seg}_2, \mathrm{Id}_{S2}$) determines if two segments have a variable access conflict. Initially, every expression of a segment is interpreted with the related instance ID function parameter. However, the instance ID for channel expressions must be updated for the later conflict analysis. Exemplary, Figure 3.8 shows the expression `ch1++` with the associated PCP (port_1, ..., port_n) of segment Seg_1. A central tool to determine the channel ID is the API of the RISC compiler [30]. Specifically, we

have created the function GetChannelId(port, id) to determine for a given port and instance ID the instance ID of the mapped channel. The parameter `id` describes the instance ID of the enclosing object for the parameter `port`. For hierarchical mappings, first, Id_1 is computed via GetChannelId($\text{port}_1, \text{Id}_{S1}$). Afterwards, Id_i is computed via GetChannelId($\text{port}_i, \text{Id}_{i-1}$). The function TranslateChannelId(Id_{S1}, GetPcp(s)) includes all steps to compute the final channel ID.

Advanced Access Analysis

The algorithm to detect access conflicts between two segments is shown in Algorithm 2. Essentially, we compute for each variable symbol two attributes to store the read and write instance ID context. Let's say a segment has read access to the instances 1, 2, and 3 of variable `x`. Also, there is no write access to any instances of variable `x`. The resulting sets are $x_{\text{Read}} = \{1, 2, 3\}$ and $x_{\text{Write}} = \{\}$. Before an instance ID is added to a set, channel context analysis is necessary. If the symbol is in a module, its instance ID applies. If a symbol is declared in a channel, we determine the related channel ID. For instance this is in Line 5 where we use the function TranslateChannelId().

Algorithm 2 Conflict analysis between two segments

```
1: function HasAccessConflict(seg_1, id_1, seg_2, id_2)
2:     for all s ∈ GetReadSymbols(seg_1) do
3:         id ← id_1
4:         if IsChannelSymbol(s) then
5:             id ← TranslateChannelId(id_1, GetPcP(s))
6:         end if
7:         S_Read ← S_Read ∪ {id}
8:     end for
9:     for all s ∈ GetWriteSymbols(seg_1) do
10:        id ← id_1
11:        if IsChannelSymbol(s) then
12:            id ← TranslateChannelId(id_1, GetPcP(s))
13:        end if
14:        S_Write ← S_Write ∪ {id}
15:    end for
16:    for all s ∈ GetReadSymbols(seg_2) do
17:        id ← id_2
18:        if IsChannelSymbol(s) then
19:            id ← TranslateChannelId(id_2, GetPcP(s))
20:        end if
21:        if id ∈ S_Write then
22:            return true
23:        end if
24:    end for
25:    for all s ∈ GetWriteSymbols(seg_2) do
26:        id ← id_2
27:        if IsChannelSymbol(s) then
28:            id ← TranslateChannelId(id_2, GetPcP(s))
29:        end if
30:        if id ∈ S_Read ∨ id ∈ S_Write then
31:            return true
32:        end if
33:    end for
34:    return false
35: end function
```

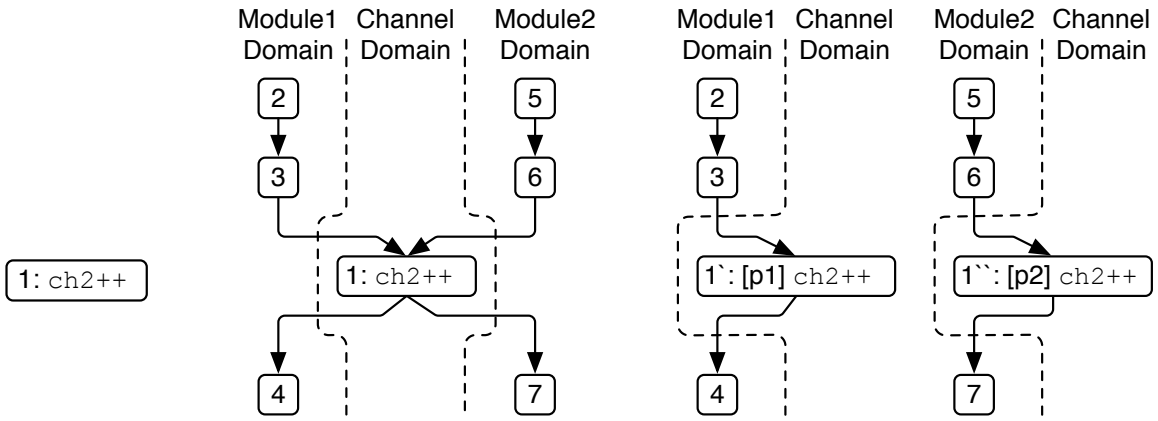
Overall, the algorithm is divided into two steps. In the first step, the read and write analysis of segment seg_1 takes place. All symbols are marked with their access type.

In the second step, a similar analysis happens. However, instead of adding a new read or

write instance ID to the symbol, an access check is performed. In Line 21 we check if a read-write conflict exists, i.e. if the read symbol of seg_2 is written by seg_1. In detail, we check if the symbol is already marked with the same instance ID. If this is the case, we return `true` because a conflict exists. Similarly, in Line 30 we check for a write-write conflict. We iterate over the written symbols of segment seg_2. If the symbol is already marked as read or written, a conflict exists.

Segment Graph Generation with Port Call Context

The classic SG is generated in two steps. In the first step, all channel functions are analyzed. For each function a partial SG is built as in Figure 3.9a. In the second step, the individual simulation threads are analyzed. When a port call takes place, the pre-analyzed segments of the channel are reused and *linked* as in Figure 3.9b. Consequently, channel segments are shared between multiple simulation threads, no individual PCP can be associated, and channel variables cannot be distinguished by module IDs.



(a) Channel Segments (b) Linking (Old) (c) Cloning (New)

Figure 3.9: Handling of channel segments from Figure 3.6

Our new SG generation with the PCP extension uses a more sophisticated strategy. Specif-

ically, each expression in a channel takes into account the individual port call context for the analysis, like `ch2++` in Figure 3.9a. So, sharing of pre-analyzed channel segments between port calls is not possible. Similar to the regular SG, we analyze all channel functions first. Next, we continue with the analysis of the simulation threads, but when we hit a port call `p`, the related SG for the channel function is *cloned* and inserted instead of linked. Therefore, channel related segments appear for each port call individually as in Figure 3.9c. Concluding, the PCP of the cloned segments is extended. First, the PCP of the port call `p` is prepended to the PCP of the cloned segments. Second, the port of `p` itself is added to the cloned segments. This strategy is needed for the analysis of nested port calls in hierarchical channels.

Event Notification Table

The event notification table contains the notification dependencies between the individual segments. This information is needed for the simulator to schedule threads with respect to possible wake-up times. Without this information, threads could run ahead, events get lost, and the simulation fails. Since event notifications are analyzed with the same approach as regular variables, it is important to take the instance ID of events into account. So that false positives do not create extraneous conflicts, we apply the new technique of the PCP analysis for the event notification analysis as well.

3.1.4 Experiments

We have implemented the PCP analysis to demonstrate the importance for parallel simulation. The following experiments show the reduction of false positive data conflicts and event notifications. Additionally, we show the speedup between the sequential and the parallel execution with (pcp) and without PCP (old). Our experiments consist of four different

application examples, namely a video Mandelbrot renderer, a high-level video decoder, a Bitcoin miner, and a NoC particle simulator. The execution times are measured on an Intel Xeon E3-1240 processor with 4 cores at 3.4 GHz. To obtain unambiguous measurements, we have turned CPU frequency scaling and hyper-threading off.

Note that in contrast to [49], we had to create for each individual channel instance an individual channel class resulting in large code duplication. Otherwise, the conflicting channel variables caused too many FP conflicts. In this work, all models contain only a single channel class with multiple instances.

Video Player

In the video player example, the stimulus sends data to an audio and a video decoder. After decoding, the processed stream goes to two speakers and one display unit. The design uses a generic channel type to transfer the data between the units. Without the PCP analysis, the individual channels are blocking each other. Additionally, they prevent the parallel decoding. In turn, the execution is effectively sequential and has the same execution time as the sequential reference simulation. In contrast, the new PCP analysis largely reduces the channel related false positive conflicts. Consequently, communication and computation can run in parallel. This enables a speedup of 1.48x (from 20.36 sec down to 13.76 sec) while reducing the data conflicts from 484 to 196 and the event notifications from 168 to 36.

Mandelbrot Renderer

The Mandelbrot renderer is a parallel video application to compute the Mandelbrot set. Figure 3.10 shows the architecture of the model. Basically, the controller orchestrates a number of renderer units. Each unit computes a different slice of the image. During the simulation, the controller triggers the individual slices. Then, a slice computes the Mandelbrot set for

given coordinates in a shared memory. The controller receives a completion signal from the units and then saves the frame. Finally, new coordinates are provided to all slices for the next frame.

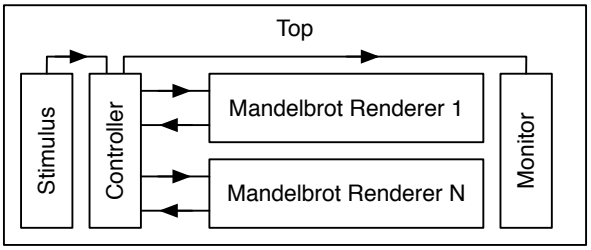


Figure 3.10: Structure of a Mandelbrot video renderer

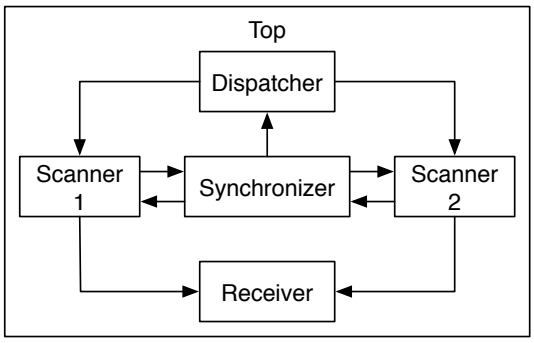


Figure 3.11: Structure of a Bitcoin miner model [19]

Table 3.1 shows the experiments of the Mandelbrot renderer application with up to 8 computation units. For all configurations, the analysis is performed with (pcp) and without (old) the PCP technique. The new analysis eliminated 2,024 false positive conflicts for the set with 8 units. The false positive conflicts effectively sequentialize the parallel execution. The PCP technique enabled a speedup of up to 3.36x on the 4-core host.

Table 3.1: Speedup of the Mandelbrot video renderer

Renderer Units	1.old	1.pcp	2.old	2.pcp	4.old	4.pcp	8.old	8.pcp
Entries in table	441	441	784	784	1,764	1,764	4,900	4,900
Data conflicts	162	154	270	190	702	262	2,430	406
Event notifications	73	40	146	58	424	94	1,508	166
Speedup	1.00	0.99	1.00	1.68	1.00	2.90	1.00	3.36

Bitcoin Miner

Bitcoin is a digital currency and is established as a decentralized payment system [19]. Bitcoin mining describes the process of generating new Bitcoins through solving math problems.

When a new Bitcoin enters the system it must prove the correctness of creation. Particularly, this proof happens through a cryptographic hash algorithm. The Bitcoin miner application is doing this with three stages which include work dispatching, scanning, and result receiving. Figure 3.11 shows the basic structure of the SystemC model. The scanners compute the hash algorithms for a given coin in parallel. If a scanner completes the computation of a coin, it synchronizes with the dispatcher and a new coin is sent to the scanners.

Table 3.2: Speedup of the Bitcoin miner example

Worker	1.old	1.pcp	2.old	2.pcp	4.old	4.pcp	8.old	8.pcp
Entries in table	100	100	256	256	784	784	2704	2704
Data conflicts	84	84	209	129	641	237	2069	501
Event notifications	4	4	8	4	16	4	32	4
Speedup	1	1	1	1.97	0.99	3.56	0.99	3.5

Table 3.2 shows the results for the Bitcoin miner experiment. The design was executed with 1, 2, 4, and 8 scanners. The amount of false positive entries effectively sequentializes the execution of the old parallel version. In comparison, the new PCP analysis resolved the conflicts. As a result, the individual workers execute in parallel and gain a speedup of up to 3.5x.

Network-on-Chip Particle Simulator

Our last experiment simulates a NoC particle simulator to demonstrate the importance of reduced false positive table entries. A tile communicates with its neighbors through bidirectional channel to the north, south, east, and west. In this model, particles move in a 2-dimensional space and affect each other. Each tile covers the computation of the moving particles for are defined area in the space. If a particle leaves the space of a tile, it transitions to the neighbor tile through channel communication.

The simulation of the particle simulator has three major stages. In the first stage, the platform communicates a set of particles to the individual tiles as an initial configuration. In the following second stage, the simulation of the particles starts and tiles compute the position for their associated particles. Particles are exchanged between neighboring tiles if they leave the scope of their hosting module. In the final step, the tiles communicate the position of the hosting particles back to the platform.

Table 3.3 shows the results of the particle simulator for grid sizes from 2x2 up to 6x6. For each size, the model is analyzed with (pcp) and without (old) the PCP analysis.

Table 3.3: Reduced false positive conflicts and speedup through the PCP analysis of the NoC particle simulator

	2x2.old	2x2.pcp	3x3.old	3x3.pcp
Entries in Table	14,641	14,641	73,441	73,441
Data conflicts (dc)	14,641	1,999	73,441	4,621
Event notifications (en)	8,988	843	28,153	1,893
False positive dc in number / percent	12,642	86.35%	68,820	93.71%
False positive en in number / percent	8,145	90.62%	26,260	93.28%
Execution time seq in sec	2,354.24		1,048.78	
Execution time par in sec	2355.7	1191.52	1046.1	577.66
Speedup	1.00	1.98	1.00	1.82

	4x4.old	4x4.pcp	5x5.old	5x5.pcp
Entries in Table	231,361	231,361	564,001	564,001
Data conflicts (dc)	231,361	8,303	564,001	13,181
Event notifications (en)	88,465	3,363	348,600	5,253
False positive dc in number / percent	223,058	96.41%	550,820	97.66%
False positive en in number / percent	85,102	96.20%	343,347	98.49%
Execution time seq in sec	590.37		378.79	
Execution time par in sec	590.57	195.82	379.98	134.87
Speedup	1.00	3.01	1.00	2.81

	6x6.old	6x6.pcp
Entries in Table	1,168,561	1,168,561
Data conflicts (dc)	1,168,561	19,363
Event notifications (en)	722,556	7,563
False positive dc in number / percent	1,149,198	98.34%
False positive en in number / percent	714,993	98.95%
Execution time seq in sec	262.58	
Execution time par in sec	270.31	86.96
Speedup	0.97	3.02

The first important observation is that both conflict tables have the same amount of entries for a given grid size.

Second, the amount of conflicts varies between the two versions. On one hand, in the old version all entries in the data conflict (DC) table are marked as conflict. In other words, all segments are conflicting with each other. Consequently, a simulation is effectively sequential.

On the other hand, in the pcp version not all table entries are marked as conflict. For the 2x2 example the data conflict table has 14,641 (old) and 1,999 (pcp) positive conflict entries. This means that the old analysis caused 12,642 false positive conflicts which are 86.53% of the table entries. With increasing grid size, the amount of false positive conflicts increases regarding to the number of channels. The channel related false positive errors is increasing up to 98% for the 6x6 grid size. A similar observation can be made for the event notification (EN) table.

Third, the speedup for the old version is 1 at most. This means that parallel simulation had no impact. In contrast, the PCP sensitive analysis has a speedup of up to 3x.

3.1.5 Summary

In this chapter, we extended a fully automatic compiler infrastructure to parallelize IEEE SystemC simulation. Our new PCP technique eliminates false positive conflicts in the analysis which impacts especially channel communication. Before, false conflicts severely compromised the parallel simulation of channels as well the computation in modules. We demonstrated the importance of the PCP technique for diverse examples and reduced the amount of false conflicts by up to 98%. As a result, we gained a speedup of up to 3.5x on a 4-core host machine.

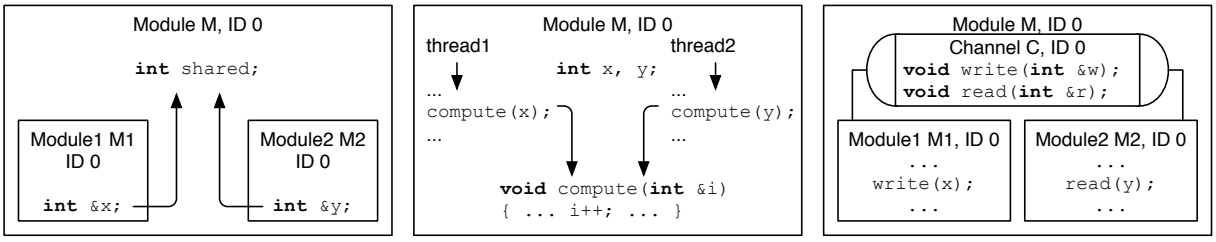
3.2 Reference Analysis

In this section, we address the topic of C++ references in SystemC models. Specifically, we introduce compiler techniques to resolve function parameter references, module member references, and references in context of channel communication. We discuss the impact of false positive conflicts and analysis techniques to avoid them.

For the evaluation, we use the same applications as for the channel conflict analysis in Section 3.1, namely, an audio and video streaming application, a Mandelbrot renderer, a Bitcoin miner application, and a NoC particle simulator.

3.2.1 Analysis of Reference Variables

SystemC designers use C++ references to model shared memory and events, as well as to execute more efficiently to save simulation time. For the data conflict analysis, it is important to identify the read and written variables. If a target of a reference cannot be identified, the segment must be marked as a conflict with all other segments to prevent race conditions. This pessimistic strategy can cause a lot of False Positive conflicts and slows down parallel simulation.



(a) Shared variable as reference (b) Function argument reference (c) Channel parameter as reference

Figure 3.12: Different usage of C++ references in SystemC models

Figure 3.12 shows three different uses of references in SystemC. First, in Figure 3.12a, the modules M1 and M2 have the member references `x` and `y` respectively which are mapped to the variable `shared` in the common top module M. Both modules exchange data through the shared variable. Second, in Figure 3.12b, module M has two threads which both call the function `compute()`. The parameter `i` is passed by reference and represents the variables `x` and `y` which are declared in the same module. Third, in Figure 3.12c, module M1 and M2 are connected through the channel C to exchange data in a synchronized fashion. Like in

the second example, the parameters are passed by reference. However, the mapped variables of the references are in different modules. Specifically, the function parameters `w` and `r` represent the variable `x` from module M1 and the variable `y` from module M2.

References to Shared Variables

For the first situation in Figure 3.12a, we introduce the *instance path* which describes a module or channel instance with the associated instance ID in the model hierarchy. For example, in Figure 3.12a the module instances M1 and M2 are sub modules of the module instance M. So, we describe them through the paths *[M,0 → M1,0]* and *[M,0 → M2,0]*. While analyzing the segments of M1, we hit access to the variable of `x` and identify it as a reference and member variable. Through the instance path, we analyze the constructor of the parent class, namely, the constructor call of M2 in the initializer list of M. Here, we have two cases. On one hand, the provided argument is a member variable of M and we identify the associated symbol for the future analysis. This situation is illustrated in Figure 3.12a and the symbol of `x` is replaced with the symbol of `shared` with instance ID 0. On the other hand, the provided argument in the constructor is a reference as well. In this case, we continue traversing the instance path and analyze the constructor of the next parent module.

Function Argument References

In the second case in Figure 3.12b, the reference is a local function parameter and represents the symbol of the function call argument. Here, we perform a function call analysis and annotate reference parameters with the called argument. Instead of using the reference symbol during the data conflict analysis, we then use the annotated symbols. We discuss this strategy for the example of Figure 3.13 which shows a more detailed version of Figure 3.12b.

The function call graph generation starts with thread1 which calls the function `compute()`

at the beginning. The function parameter `i` is analyzed as a reference, so the symbol of `i` is annotated with the symbol of the member variable `x`. Next, the function call analysis follows the control flow and detects function call `encode()` which takes a reference as parameter as well. Consequently, the parameter `e` is annotated with the symbol of the calling function. However, at this point we annotate the symbol of `x` instead of `i` because `i` is a reference as well. After traversing the call graph for `encode()`, the analyze the function call of function `decode()`. Again, parameter `d` is a reference. However, this time we annotate the symbol of `d` with the member variable symbol of `z`. After finishing the function call analysis for thread1, the analysis for thread2 takes place. The procedure is the same, however, instead of annotating symbol `x`, symbol `y` is annotated.

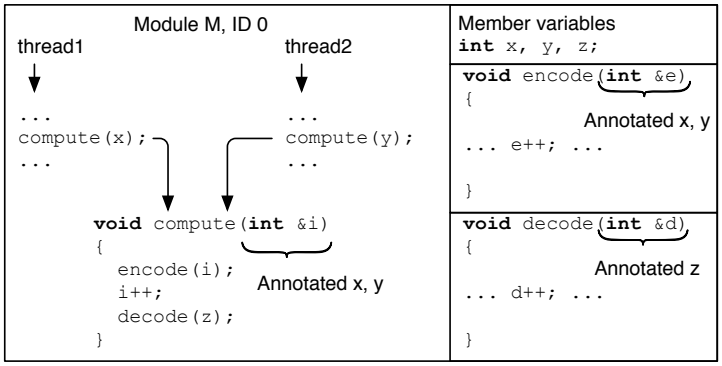


Figure 3.13: Detailed annotation of references from Figure 3.12b

The function call and reference analysis takes places before the data conflict analysis starts. During the data conflict analysis, member references and function references are treated differently. Member references are lookup up through the constructor calls. For function parameter references, we use the annotated variable symbols.

Channel References

In the third case in Figure 3.12c, the two modules M1 and M2 communicate through the channel C. To resolve the variables `w` and `r`, we combine the techniques of instance paths from Figure 3.12a, variable symbol forwarding from Figure 3.12b, and the PCP from Section 3.1.1. First, we notice that the variables `w` and `r` are references and they belong to a channel function. So, we use the PCP technique to identify the channel instance ID of C. Next, we use the RISC analysis to determine all instance paths of connected modules to channel C, which are *[M,0 → M1,0]* and *[M,0 → M2,0]*. The reference arguments `w` and `r` are annotated with the symbols of `x` and `y` through the function call analysis. Afterwards, we iterate over all instance paths and check if one of the annotated variables is declared in any module of the instance path. If this is the case, we determine the resolved variable and the associated instance ID through the instance path.

For more complicated situations, we can combine the individual techniques. For instance in Figure 3.12a, instead of having the variables `x` and `y` as module members of M they could be also member references. In this case, we combine the techniques of Figure 3.12a and Figure 3.12b.

3.2.2 Experiments

In this section, we discuss the evaluation of the proposed PCP and reference resolving techniques in SystemC models. The following experiments show the reduction of false positive (FP) data conflicts (DC) and event notifications (EN). Additionally, we show the speedup between the sequential and the parallel execution without PCP (old), with PCP (pcp), and with references resolving (ref). Our experiments consist of four different application examples, namely a video Mandelbrot renderer, a high-level video decoder, a Bitcoin miner, and a NoC particle simulator. For the experiments with references in the model, we use references

in the channel functions.

The execution times are measured on an Intel Xeon E3-1240 processor with 4 cores at 3.4 GHz. To obtain stable measurements, we have turned CPU frequency scaling and hyper-threading off.

We evaluate the PCP extension with support for references and measure the reduced false positive conflicts and gained simulation speedup. Specifically, first we run the experiments with the PCP extension (pcp) and afterwards with PCP in combination with the implemented referencing resolving techniques (ref). To emphasize the importance of reference support, we use the same SystemC models for the evaluation as for the evaluation of the PCP technique in Section 3.1.1. We changed the function parameters of the channel function from regular variables to references and discuss this effect in terms of speedup as well. In the following, we only discuss the data conflict, since there are no event references in the models.

Video Player

The player example has a sequential execution time of 20.34 sec, parallel execution time without reference support of 20.37 sec, and with reference support of 13.87 sec. This is a speedup of 1.46x which is similar to the speedup of the experiments in the previous section. The identical speedup is related to the nature of the communication in this example. The channel interfaces for the read and write function take integer values as parameters. The channels are communicating integer values the through channels. Copying an integer value or the value of a reference doesn't make any difference since addresses in C++ also are integers. Consequently, we gain the same speedup.

The strict rule of marking all segments with writing access to references as a global conflict, introduces 448 out of 484 data conflicts. After applying the reference resolving technique,

the number of conflicts is reduced to 196 and 56.25% of false positive conflicts are eliminated. In other words, the number of data conflicts are identical for the non reference and reference version.

Mandelbrot Renderer

Table 3.4 shows the evaluation of the Mandelbrot application and the impact of the reference analysis. First, we notice that without the reference analysis technique (pcp) 4,836 out of 4,900 segments are in conflict with each other which are 99% of the segments. After applying the reference resolving techniques (ref), the number of conflicts is reduced by 4,430 conflicts to 406 conflicts. In other words, more than 90% of the conflicts are false positive conflicts and prevent an effective parallel simulation. The maximum speedup is 3.56x which is similar to the speedup in the previous Section 3.1.1. Also, the communication interfaces use a plain datatype, namely floating point numbers. Passing these values by references don't affect the simulation speed.

Table 3.4: Speedup of the Mandelbrot video renderer

Renderer units	1.pcp	1.ref	2.pcp	2.ref	4.pcp	4.ref	8.pcp	8.ref
Table size	441	441	784	784	1,764	1,764	4,900	4,900
DC	440	154	780	190	1,748	262	4,836	406
Speedup	1.00	1.00	1.00	1.67	1.01	2.87	1.01	3.43

Bitcoin Miner

The Bitcoin application uses channels with plain datatypes as the two previous examples. So, the reference resolving technique produces the same results as in the previous section, namely a speedup of 3.56x. The detailed results are presented in Table 3.6.

Table 3.5: Reduced false positive conflicts and speedup through the PCP with reference analysis of the NoC particle simulator

	2x2.old	2x2.pcp	3x3.old	3x3.pcp
Entries in Table	14,641	14,641	73,441	73,441
Data conflicts (dc)	14,641	1,999	73,441	4,621
Event notifications (en)	843	843	1,893	1,893
False positive dc in number / percent	12,642	86.35%	68,820	93.71%
False positive en in number / percent	0	0.00%	0	0.00%
Execution time seq in sec	1515.42		673.50	
Execution time par in sec	1516.37	766.18	672.17	372.14
Speedup	1.00	1.98	1.00	1.81

	4x4.old	4x4.pcp	5x5.old	5x5.pcp
Entries in Table	231,361	231,361	564,001	564,001
Data conflicts (dc)	231,361	8,303	564,001	13,181
Event notifications (en)	3,363	3,363	5,253	5,253
False positive dc in number / percent	223,058	96.41%	550,820	97.66%
False positive en in number / percent	0	0.00%	0	0.00%
Execution time seq in sec	379.16		242.72	
Execution time par in sec	380.49	129.20	245.14	185.00
Speedup	1.00	2.93	0.99	1.31

	6x6.old	6x6.pcp
Entries in Table	1,168,561	1,168,561
Data conflicts (dc)	1,168,561	19,363
Event notifications (en)	7,563	7,563
False positive dc in number / percent	1,149,198	98.34%
False positive en in number / percent	0	0.00%
Execution time seq in sec	169.08	
Execution time par in sec	177.88	59.19
Speedup	0.95	2.86

Table 3.6: Speedup of the Bitcoin miner example

Worker	1.pcp	1.ref	2.pcp	2.ref	4.pcp	4.ref	8.pcp	8.ref
Table size	100	100	256	256	784	784	2,704	2,704
DC	97	84	253	129	781	237	2,701	501
Speedup	0.98	0.98	1.02	2.01	1.00	3.63	1.00	3.56

Network-on-Chip Particle Simulator

In the NoC particle simulator application, the individual tiles communicate particles and the related particle information through dedicated data structures. The tiles always send a chunk of particles in form of a `std::vector<Particle>` to minimize the communication overhead of single particle communication. In comparison to the previously discussed examples, the communication interfaces of the NoC application do not use plain datatypes, instead they use a `std::vector<Particle>`. So, compared to copy the particles by value, communicating by reference avoids the duplication of particles and the creation of a new vector. This phenomenon can be observed in terms of simulation time in the tables Table 3.3 and Table 3.5. The sequential execution of the 8x8 NoC with references takes 262,58 sec and for the reference version only 129.20 sec. So, communicating them by reference safes simulation time since only an address value is passed.

Table 3.5 shows the experimental results of the particle simulator for grid sizes from 2x2 up to 6x6. The reference analysis reduces eliminates up to 98% of data false positive conflicts and gains a speedup up to 2.93x. The simulation of the parallel version without references for the 4x4 grid takes 195.82 sec, and the parallel version with references needs 59.19 sec. In other words, we gain an additional speedup of 1.51x to the NoC version without passing by reference.

3.2.3 Summary

We a proposed a novel technique to resolve SystemC references which have been marked as race conditions before. We evaluated this new approach on the same application set as the PCP evaluation, however, channels used C++ references. We gained a speedup of up to 2.86x for the parallel version of the particle simulator compared to the sequential version. Additionally, we measured a speedup of 1.51x from the parallel particle simulator with reference version to the parallel particle simulator without references.

Chapter 4

Hybrid Analysis

In the previous chapter, we discussed static analysis techniques to reduce false positive table entires. However, designers tend to provide model information only at run time, for instance, the number of rows and columns of a grid. In this chapter, we discuss a hybrid analysis technique to consider run time information as well. Additionally, we provide an annotation scheme for libraries to support 3rd party source code.

4.1 Introduction

The increasing complexity of embedded systems slows down the design process of new products. Designers use simulation as a tool to validate prototypes. However, the dramatically increasing simulation time has been identified as a bottleneck in the design process. Various approaches have been made to optimize the simulation performance. For instance, the simulation reduced level has been decreased and communication has become more abstract. Although state-of-the-art PCs have multi-core processors, most simulations are still executing sequentially.

SystemC [3] is a widely-used tool for simulating and modeling embedded systems. We advocate an advanced approach to simulate SystemC models fully in parallel without losing accuracy. This is in contrast to other limited techniques. For instance, techniques like time decoupling have been proposed to boost the simulation performance. However, this method results in inaccurate simulation results [25].

Our RISC compiler infrastructure analyzes a given design, identifies potential race conditions, and transforms the sequentially written model into a parallel executable design. The transformation happens automatically and the designer has no burden of partitioning the model. We propose a hybrid analysis to consider all possible aspects of the design including 3rd party libraries.

4.1.1 Problem Definition

SystemC is the de facto standard library for modeling and simulating embedded systems. The official simulation kernel performs the simulation in a sequential fashion. In other words, only one simulation thread is active at any time. If several threads could be simulated in parallel rather than sequentially, then we would obtain a significant decrease in the simulation time.

First attempts have been made to run the simulation in a parallel fashion. A dedicated compiler is used in [16] to analyze the design hierarchy and to identify the potential race conditions among the individual threads. However, in order to perform these advanced simulation techniques, two criteria must be satisfied. First, the entire source code must be available for the static analysis. It cannot be partially provided in a library. Second, the design must be statically explorable to identify the design hierarchy.

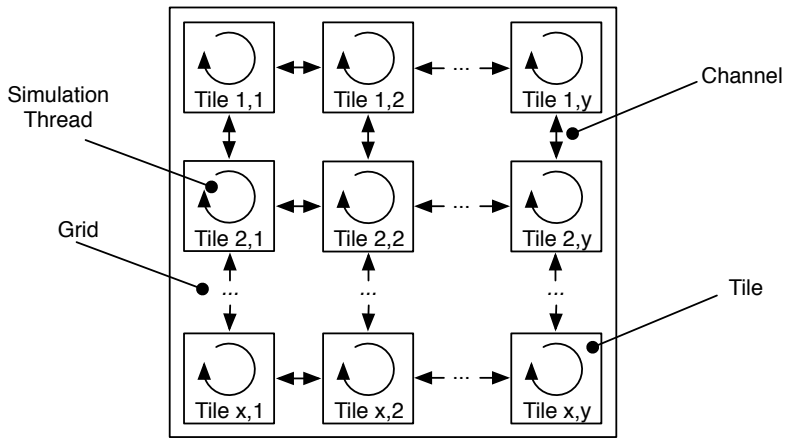


Figure 4.1: General structure of a Network-on-Chip design model

The NoC paradigm is a widely-used design pattern. However, it violates these parallel simulation requirements. Figure 4.1 shows a typical NoC which is assembled through a hosting grid and tiles which are organized in rows and columns. A tile contains Intellectual Property (IP) and communicates with other tiles located to the north, west, south, and east of it. A tile can be a user-defined or a 3rd party IP element which is encapsulated in a library. During the design space exploration, architects test various combinations of these grid sizes and tile components. The number of rows and columns is often defined through command line parameters. Thus, the tiles are allocated in two loops with the `new` operator which is not statically analyzable. Evidently, these two limitations prevent the analysis for an efficient parallel simulation.

In this chapter, we propose a solution in order to simulate designs in an accurate parallel fashion. These models can include 3rd party libraries and code which is not statically analyzable. In the new design flow, we perform a dynamic analysis which is followed by a static analysis. Additionally, we provide the designer with the opportunity to annotate 3rd party libraries. Such annotations are taken into account for the advanced design analysis and allow the simulation of library code in a parallel fashion.

4.1.2 Related Work

Parallel simulation of DES is a well-studied subject. Initial work has been contributed by [23].

The concept of a SG for parallel simulation was first introduced in [16] for synchronous and out-of-order parallel simulation. Later, the SG infrastructure was used for may-happen-in-parallel analysis for safe ESL models in [17]. Both contributions require a complete design model with a statically analyzable module hierarchy. In contrast, our approach supports 3rd party libraries and non-statically analyzable module hierarchies in designs.

Time decoupling is a widely-used method that speeds up the simulation of SystemC models. Parts of the model execute in an unsynchronized manner for a user defined time quantum. However, this strategy is associated with inaccurate simulation results [25]. [54] and [55] propose a technique to parallelize time-decoupled designs. This technique requires the designer to manually partition and instrument the model in parts of time-decoupled zones. In contrast, our approach supports a 100% accurate simulation of designs. Also, our compiler automatically instruments the model.

A tool flow for parallel simulation of SystemC RTL models was proposed in [44]. The model was partitioned according to a modified version of the Chandy–Misra–Bryant algorithm [12]. In contrast, our conflict analysis considers the individual statements of threads. Also, our solution is not restricted to only RTL models.

An approach of static analysis for simulation of SystemC models on GPUs was provided in [51]. In contrast, our approach combines static and dynamic analysis to obtain information for the conflict analysis which is only available at run time.

4.2 Hybrid Analysis

The transition from a sequential simulation towards a parallel simulation is an extensive process. The design must be analyzed and prepared for potential race conditions to avoid unpredictable data corruption. This requires a full understanding of the module hierarchy. One option is to statically extract the module hierarchy and analyze the individual threads. However, in most cases not all of the information can be explored statically. For instance, during the design space exploration, designers test various prototypes to explore an optimal design. Design parameters are passed via the command line to define the number of modules, channels characteristics, and other needed information. The instances of modules, channels, and ports are created through loops in a dynamic fashion. However, the essential parameters are only available at run time, so they cannot be statically analyzed. The result would be an incorrect model transformation which produces wrong simulation results.

Figure 4.2 shows our proposed design flow which supports command-line parameters and the dynamic analysis for a fully accurate parallel simulation. Essentially, the tool flow is split up into two major stages. The first stage performs a *Dynamic Design Analysis* and collects the design hierarchy. The second stage allows the *Static Conflict Analysis* to integrate the obtained data from the previous stage. As a result, the design is transformed into a thread-safe design for fast parallel execution.

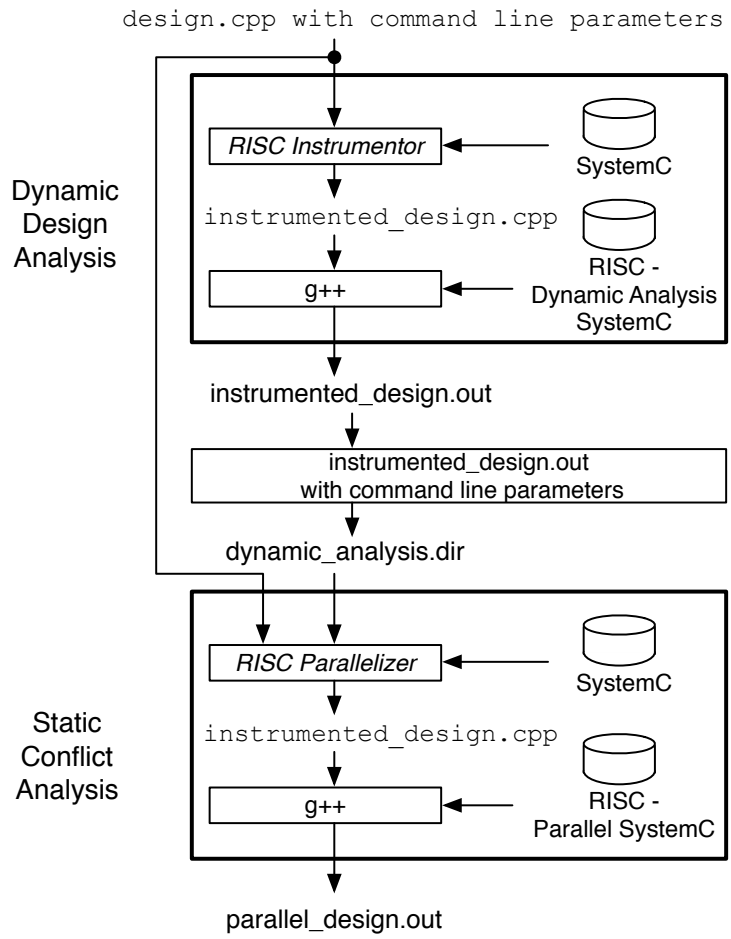


Figure 4.2: Tool flow for the proposed hybrid analysis of design models

4.2.1 Static Conflict Analysis

We use static conflict analysis which is introduced in Section 1.4.

4.2.2 Dynamic Design Analysis

The purpose of the *Dynamic Design Analysis* is to provide data for the *Static Conflict Analysis* which can only be obtained at run time. This data is structural design information which, for example, is dependent on command-line parameters, or hidden deeply in nested

loops. The results of this step are provided in a dynamic internal representation (DIR) file and then used as a lookup table in the *Static Conflict Analysis*.

In detail, the DIR file includes the module hierarchy, the port mapping, and the mapped variables of references. We represent this information as an abstract tree. Specifically, we store the declaration name and the address for all declarations. In addition, we store the address of the bounded channel for ports and the typename of the channel for the channels. Listing 4.1 illustrates such a DIR file. Undoubtedly, we can identify that the reference `ref` is mapped to the variable `var`. Also, we can identify that the port `port` is bound to the channel `chnl1` of the type `MyChannelType`. We can use the address as a lookup because references and ports are only bound once. During the simulation, the specific address might be different. However, the lookup will lead to the same variable.

```
1 top:0x111(
2   chnl1:0x222:MyChannelType, var:0x333
3   mod1:0x444(
4     var:0x555,ref:0x333,port:0x222
5   )
6 )
```

Listing 4.1: Example of a DIR file

We can partially analyze a design in three different ways, but each has its own limitations: A) The C++ language has limited support of introspection. It is possible to identify the type name of a variable at run time. However, it is not possible to identify the declaration name of a variable. B) The SystemC library has an interface for the introspection of a design. It is possible to identify all top modules at run time, and then traverse all the elements which are derived from `sc_object` in a hierarchical fashion. However, it is not possible to identify plain old data types (e.g. `float` and `int`) or their respective declaration name. C) A static analysis of the source code allows to identify modules and analyze all of their members. However, the module hierarchy can only be explored with a severely limiting modeling guide lines.

In our approach, we perform a combined solution of A, B, and, C to generate a DIR file where the designer has no modeling limitations. First, the RISC instrumentor reads the file *design.cpp* and analyzes all the design elements statically. In other words, we have access to the declarations and their names. For each module, we instrument functions to print variables, ports, and other information. For instance, in the function `void print_vars()` {`fprintf("var:%p",&var); ...`}, we use the `typeid` support of C++ to obtain the type name at run time. Through that process, we are able to generate the source code for the instrumented design and build an executable.

Second, we modify the simulation kernel to allow it to undergo the *Dynamic Design Analysis*. The simulation of a SystemC model is split into two major phases. In the first phase, the *elaboration* starts where the module hierarchy and the port binding are established. In the second phase, the *simulation* of the design is performed. The SystemC API provides a hook between these two steps. We simulate the design with all command line parameters until the *elaboration* is completed. Finally, we traverse the module hierarchy via the SystemC introspection API and call the functions for the variable printing.

4.3 Library Handling

The simulator requires the following two bits of information in order to run in a parallel manner. First, it needs to know the race conditions of each individual thread before the simulation starts. Second, the simulator needs to know the current segment ID of each thread. Before the simulator triggers the next segment of a thread, it is essential to check for race conditions with all other active segments. These are two main obstacles for real world SystemC designs which include standard and 3rd party libraries.

The first issue is that the *Static Conflict Analysis* needs access to the function bodies to iden-

tify the race conditions for the individual threads. However, 3rd party intellectual property (IP) is often shipped through libraries. The designer has access to the function signatures, but the implementation is hidden in the library file. In other words, our static analysis cannot identify potential `wait()` calls and race conditions. Consequently, a segment which calls a library function is set in conflict with all other segments. For example, inherent function calls like `printf()` and `sqrt()` would sequentialize the parallel simulation. Therefore, we provide a function annotation scheme to include information about these library functions for the static analysis.

The second issue is that the parallel simulator needs to know which segments are ready to execute. One previously employed strategy was to statically instrument each individual `wait()` call with the associated segment ID, e.g., `wait(event,42);`. The simulator obtains the upcoming segment ID through the `wait()` call. However, this strategy cannot be used for designs with 3rd party libraries. The RISC Parallelizer cannot instrument library files. Instead, we provide a modified RISC simulation kernel to pass the segment ID through the 3rd party library to the simulation kernel.

4.3.1 Function Annotation

We present an annotation scheme for function declarations to provide information for the conflict analysis. Thus, the user can annotate via `pragma` statements two different pieces of information, namely, the conflict status and the type of `wait()` function calls in a function body. We consider a function as conflict-free if the corresponding function body has no read/write access conflicts on any shared state with the other threads in the simulation model. A segment which calls a function with the annotation of *non-conflict-free* results in a conflict with all other segments. The simulator executes this segment sequentially and safeguards a fully accurate simulation.

Figure 4.3 shows four different options of annotating a function declaration with wait information. We designed the scheme to have full support of the SystemC built-in library channels. In the first case, the function has no wait statement. This is the option for non-blocking function calls. The next two cases cover the situation that the function has a conditional or a non-conditional wait, for instance an `sc_buffer`. The last case portrays a more complicated channel type, an `sc_fifo`, where the `wait()` call is in a loop.

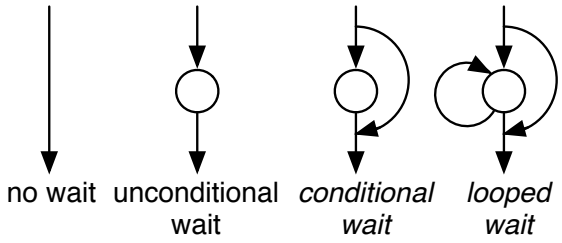


Figure 4.3: Wait annotation for function declarations

The user also has the option not to provide any annotation for library functions. In this case, we assume that the function is conflict-free and there are no wait statements in the function. This behavior is expected for the Standard C Library and Standard Template Library. We decided to make this the default mechanism to avoid annotating all standard library functions.

4.3.2 Segment ID Passing

The parallel simulator needs the active segment ID of each individual thread during the simulation. The approach of instrumenting `wait()` calls with an additional segment ID parameter is only possible if the source code is available for all parts of the design. In other words, we cannot instrument `wait()` calls which occur in the library.

Figure 4.4 illustrates our generalized solution for support of libraries. The thread carries

the upcoming segment ID from the user domain to the parallel RISC SystemC library. We instrument the function call `setID(42)` before the function call in the library domain happens. In the RISC SystemC kernel, we get the segment id via `getID()`. This solution provides the benefit that any 3rd party channel can be used without any modification.

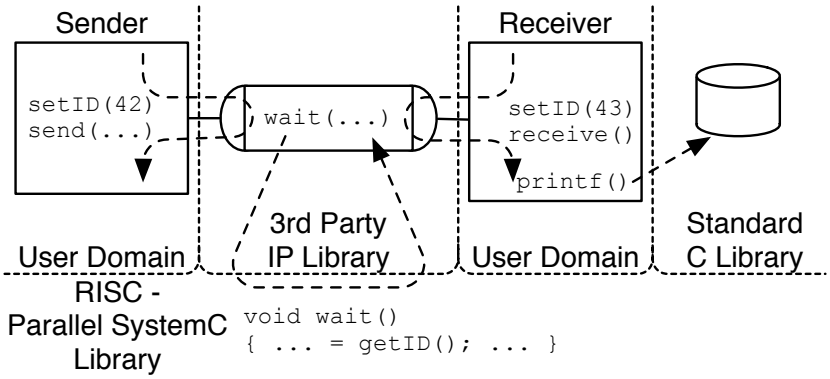


Figure 4.4: Different domains of a design

4.4 Experiments

4.4.1 Producer Consumer Example

We implemented the consumer producer example which is illustrated in Figure 4.4. Both the sender and receiver crunch numbers and exchange them. The number of sender and receiver pairs is scalable via the command line.

The regular RISC Parallelizer — without the *Dynamic Design Analysis* capabilities — cannot process this design and returns an error message. The *Dynamic Design Analysis* creates the correct module hierarchy and identifies the number of modules which is double the number of communication pairs. The obtained speedup depends on the number of modules and the ratio of communication and number crunching.

4.4.2 Network-on-Chip Particle Simulator

We selected as a comprehensive example a NoC particle simulator to demonstrate the parallel simulation capabilities of our hybrid analysis. The abstract architecture of the particle simulator is illustrated in Figure 4.1. The grid is assembled of tiles where each tile communicates bidirectionally with a tile to its north, south, east, or west. For a 8x8 example, we have 64 tiles and one grid module. Each tile has one thread which computes the motion of the individual particles in a certain area of the model. These particles move continuously in 2D space. The moment when a particle crosses the boundary, the responsibility of computing and updating the position of the particles shifts from one tile to its neighbor tile. The entire design can be scaled up to any quadratic size. The user can define via the command line the number of tiles as well as the number of particles, the gravity, and other options. The individual tiles including ports and channels are dynamically created.

At the beginning of the simulation, the grid sends the initial particles to each individual tile. This happens in a purely sequential fashion. Next, all tiles simulate the particles and synchronize with their neighbors. A tile can be blocked due to its communication with one of its neighbors. At the end of the simulation, all the tiles send the particles back to the grid.

Static Design Analysis

The static analysis of the particle simulator causes an error message: "Error: Array of modules in line 231". The RISC Parallelizer detects an array of pointers for tile modules. However, it cannot identify how many instances are created. The same applies to the array of ports and channels. The module hierarchy cannot correctly be extracted and the RISC Parallelizer does not create an executable.

Dynamic Design Analysis

The dynamic analysis is performed for different grid sizes. For the 8x8 particle simulator, 65 modules and 176 channels are correctly identified. The parallel simulation creates the same results as the traditional sequential DES. In other words, the parallel simulation has the same accuracy as the sequential simulation.

Table 4.1: Simulation speedup of the Particle Simulator

	Speedup				Time (in sec.)	
Particles	10k	20k	40k	60k	seq. 60k	par. 60k
5x5	2.56x	3.58x	3.25x	2.97x	160.2	53.77
6x6	2.80x	4.88x	5.04x	4.34x	126.53	29.09
7x7	2.32x	3.91x	5.01x	4.87x	117.46	24.11
8x8	2.07x	4.12x	6.05x	6.39x	108.4	16.96

Table 4.1 shows the simulation speed of the individual particle simulators. We performed all experiments on an Intel Xeon E3-1240 with 4 cores with 2 threads per core. Our test infrastructure provides a theoretical speedup of maximum 8x. The speedup is dependent on the number of particles and the grid size. For a 8x8 and 6x6 grid size a speedup of 6.39x respectively 5.04x is measured. The increasing speedup is dependent on the number of particles in the simulation. More particles increase the number crunching and consequently the execution time of the parallel threads. The sequential communication becomes a minor part of the simulation.

The design has several sequential parts, which cannot be parallelized. One reason is due to the initialization and final synchronization of the individual tiles. A second reason is due to the communication which is performed in a double handshake fashion. This means a tile is blocked until the receiving tile completes the communication.

Table 4.2: Exchanged Particles of the Particle Simulator

	7x7		8x8	
Particles	20k	60k	20k	60k
seq.	467,728	1,321,247	497,111	1,356,083
par.	467,728	1,321,247	497,111	1,356,083

Table 4.2 shows the simulation characteristics of the sequential and parallel simulation to demonstrate the accuracy. All simulations are performed with 20,000 and 60,000 particles for a particle simulator of 6x6 and 8x8 tiles. Both, the sequential and the parallel simulation have identical numbers of communicated particles.

4.5 Summary

In this chapter, we propose an efficient solution for accurate and parallel simulation of SystemC models with 3rd party libraries. Our approach does not trade off simulation speed for simulation accuracy as do time decoupled modeling techniques. In contrast to previous compiler related work, our RISC infrastructure allows to simulate models which are not statically analyzable. Also, we are now able to simulate models in parallel which include 3rd party libraries for the first time. The designer has to provide manual annotation for function declarations which are taken into account by RISC. The annotated functions will be mapped on a dedicated segment to handle the complexity of the unknown segment graph for the function. We demonstrated a simulation speedup of 6.39x while maintaining 100% accuracy.

Chapter 5

Vectorization of System Level Designs

In chapter Chapter 3 and Chapter 4, we discussed static and hybrid analysis techniques to increase the thread level parallelism. Now, we take date level parallelism into account to simulate the individual threads at a higher speed. Specifically, we provide an algorithm to analyze loops and to identify candidates for vectorization.

5.1 Introduction

The functional complexity of embedded systems has increased dramatically over the last years. Additionally, many complex design properties like energy consumption or thermal heating must be considered during the design process. Designers use simulation as a tool to validate all kinds of characteristics of their prototypes. The combination of the increasing complexity and the number of validated properties makes the simulation intensively time-consuming.

SystemC [3] is the de facto standard for modeling, simulating, and validating embedded systems. The Accellera reference implementation performs simulations in a sequential fash-

ion. In other words, at any time of the simulation at most one simulation thread is active. SystemC TLM-2.0 provides the concept of time decoupling to speed up simulations. Unfortunately, the benefit of a speed boost comes at the price of simulation inaccuracy [25]. Other work such as [53] and [54] propose a modified SystemC kernel that supports multithreading. However, designers must manually identify and resolve all potential race conditions in their models. Also, they have to ensure that the design is thread-safe. Consequently, overlooked and not protected race conditions often lead to incorrect simulation results.

We propose an automated compiler approach for parallelizing the simulation using *thread* and *data-level parallelism* to save simulation time. This approach is in contrast with existing works that require manual code transformation. First, our SystemC compiler [30] performs a fully automatic analysis to identify and exploit the available *thread-level parallelism*. Additionally, our compiler performs an analysis for *data-level parallelism* based on our SystemC-aware internal representation. The outcome is a report that lists source code locations for vectorization optimization. Finally, our associated parallel SystemC library simulates the thread-safe design in out-of-order parallel fashion similar to [16]. The data-level analysis is based on and guided by thread-level analysis which is aware of PDES and specifically SystemC semantics. Note that this distinguishes our compiler from general-purpose compilers (such as Intel icpc) that cannot automatically identify SIMD parallelism in the source code.

To the best of our knowledge, this work is the first to apply and exploit SIMD vectorization on top of thread-level parallel SystemC simulation.

5.1.1 Problem Definition

The official simulation kernel of SystemC schedules the individual threads in a sequential fashion although parallel multi- and many-core platforms are available. Consequently, the

simulation time of modern embedded systems becomes a bottleneck in the design flow. Simulations may run for hours until unexpected events occur or the simulation crashes. Designers try to fix the behavior and then they have to rerun the simulation from the beginning. The results of their adjustments are available only after another long wait.

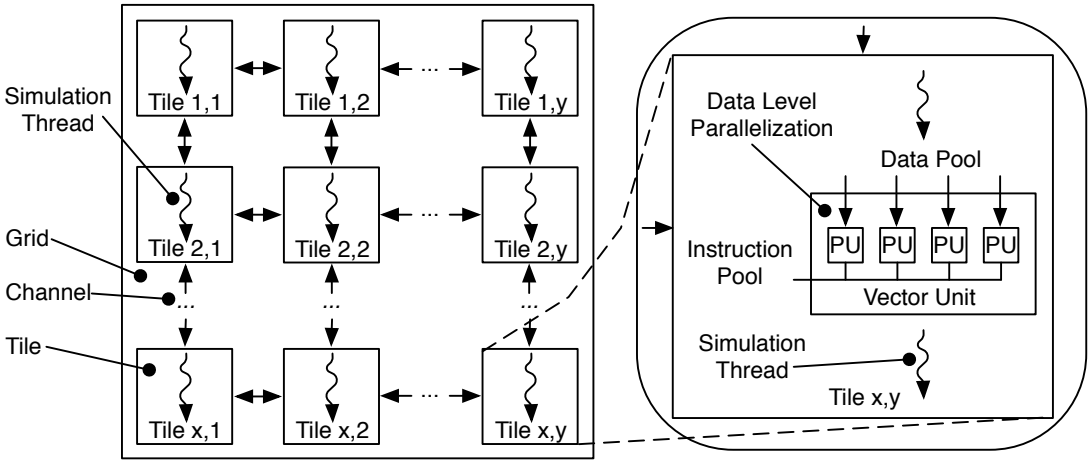


Figure 5.1: General structure of a NoC design model with vector units in tiles

Many attempts have been made to perform SystemC simulations in parallel. Most of these solutions have two major limitations. First, the designer has to manually partition his design into different domains. Through the lack of automation for parallelization, designers merge the orthogonal concepts of design methodology and techniques to save simulation time. Second, traditional parallel discrete-event simulation is limited to thread-level parallelism. Vectorization units for high-performance computing are not utilized although they have been available in standard Personal Computer (PC)s for 8 years. General-purpose compilers like the Intel icpc have significantly higher difficulty in identifying potential parallelism because they are not aware of the intricate SystemC semantics.

NoC design is a popular pattern in the embedded systems domain. Figure 5.1 illustrates a typical NoC platform that is assembled by a hosting grid and tiles which are organized in rows and columns. A tile communicates with the other tiles located to the north, west, south,

and east. This architecture is well-suited for distributed algorithms. Often, tiles follow the same computation pattern but they are computing different parts of the same problem. Such a NoC supports parallel simulation in multiple ways. On one hand, *thread-level parallelism* is given through the individual tiles which process in parallel. On the other hand, each tile can perform faster through *data-level parallelism*.

In this chapter, we propose an automatic solution to run simulations in an ultimate parallel fashion. We extend the SystemC compiler [30] which first reads a design and identifies the associated design hierarchy. Next, it performs conflict analysis to identify potential race conditions and partitions each thread into a set of segments [16]. Each segment considers all statements that can be executed in a scheduling step of a thread. In addition to the thread-level parallelism, we add analysis to identify data-level parallelism. More specifically, our compiler identifies candidates for vectorization in the design by taking the information from the thread-level analysis into account. Finally, the designer can decide whether or not the identified code locations are suitable and worthwhile for vectorization.

5.1.2 Related Work

Parallel discrete-event simulation is a well-studied subject. Initial work has been contributed by [23] on thread-level parallelism.

The concept of a SG for parallel simulation of threads was first introduced in [16] for synchronous and out-of-order parallel simulation. Later, the SG infrastructure was used for may-happen-in-parallel analysis for safe ESL models in [17]. In contrast, we are using the SG to identify and exploit *thread* and *data-level parallelism* to achieve higher simulation performance.

Time decoupling is a widely-used method which can speed up the simulation of SystemC

models. Parts of the model execute in an unsynchronized manner for a user-defined time quantum. However, this strategy generally suffers from inaccurate simulation results [25]. [54] and [55] propose a technique to parallelize time-decoupled designs. This technique requires the designer to manually partition and instrument the model in segments of time-decoupled zones. The authors state "Preparation for time-decoupled parallel simulation was done in less than one person-day." [55]. In contrast, our approach supports a 100% accurate simulation of designs and our compiler automatically instruments the design for parallel execution in minutes. In other words, the designer does not need to be familiar with the design to perform a safe and fast parallel simulation.

A tool flow for parallel simulation of SystemC RTL models was proposed in [44]. The model was partitioned according to a modified version of the Chandy–Misra–Bryant algorithm [12]. In contrast, our conflict analysis considers the individual statements of threads. Also, our solution is not restricted to RTL models.

An approach of static analysis for simulation of SystemC models on Graphics Processing Unit (GPU)s was provided in [51]. In contrast, our approach performs an automatic analysis on the thread *and* data-level parallelism. [11] proposes an automated transformation of RTL applications to GPUs. Here, it is required that the RTL input description is synthesizable.

5.2 Parallel Computation

Parallel computation is a general strategy to optimize the execution time of programs. This technique is applied on various levels e.g. *instruction-level*, *data-level*, and *thread-level parallelism*, as well as through *distributed computing*. The objective of exploiting *instruction-level parallelism* is to maximize instruction throughput. Here, a CPU is organized as a pipeline, such as the traditional RISC hardware architecture. Each instruction is partially executed

at a different stage in the pipeline. *Data-level parallelism* is often associated with the term SIMD. Multiple operations of the same kind are executed in parallel with different data sets. This technique is often used to parallelize low-level loops. At the *thread-level parallelism*, the individual threads of a program are executing in parallel fashion. It must be verified that no race condition occurs among the individual threads. Otherwise, the simulation is compromised and incorrect behavior occurs. Finally, the computation can be distributed over a network of PCs which is called *distributed computing*.

5.2.1 Thread Level Parallelism

The thread-level analysis identifies potential race conditions between the individual threads. In detail, we partition each thread into segments which is more comprehensively introduced in Section 1.4. A segment considers all potentially executed expressions between two scheduling steps. A new scheduling step is triggered with a `wait()` function call which yields control back to the simulation kernel. Figure 5.3 shows a SG of the simple source code in Figure 5.2.

```
0 void foo() {
1   r++;
2   wait();
3   a=b+c;
4   if(condition){
5     i++;
6     wait();
7     j++;
8   } else {
9     b=x+y;
10  }
11  z=z*z;
12  wait()
13  y=z+4; }
```

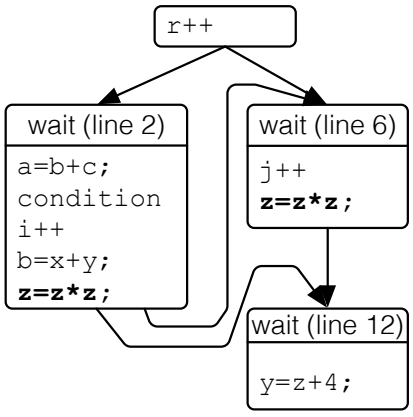


Figure 5.2: Example Source Code Figure 5.3: Segment Graph of Figure 2

The SG has similarities with a control flow graph but differs significantly in semantics. A new

segment is only started when a `wait()` function call occurs. For instance, the exemplary graph in Figure 5.3 has four different segments where the first segment is the initial one and the three others are initiated with `wait()` calls. Also, it is possible that a statement occurs in multiple segments. The segments starting in line 2 and line 6 both include the statement `sqr=sqr*sqr`.

After the SG is created for all threads, the data conflict analysis takes place. All read and written variables are analyzed for each segment. Next, all segments are compared for possible read-after-write and write-after-write conflicts. Finally, we pass the conflict table to the parallel simulator for decision making at run time.

5.2.2 Data Level Parallelism

Vectorization, a.k.a. parallelization utilizes additional hardware units in parallel *lanes* to speedup the computation. Multiple operations of the same type are executed with different data in parallel. For instance, the Intel Advanced Vector Extensions (AVX) have a data width of 256 bit and they can process up to eight 32 bit integer operations or four 64 bit double operations in parallel. Due to some overhead of arranging data in lanes, data-level parallelism pays off best in loops.

Various criteria must be satisfied to vectorize a loop. First, the loop contains only straight-line code. In other words, each lane in the vectorization unit must perform the same operation. On one hand, `goto` and `switch` statements are not allowed because for each loop iteration the control flow would change. On the other hand, `if` statements are allowed, if they can be transformed into *masked* assignments. Second, vectorization includes unrolling of loops. So a loop can be only vectorized if the number of loop iterations is countable. Additionally, no loop index variable can be written in the loop body. Next, no data dependent exit conditions are allowed e.g. `if(...)break;`. Finally, no backward-carried data

dependencies (e.g. `A[i]=A[i-1]+2;`) can be in the loop body. A full list of vectorization requirements is available in [1]. We do not follow function calls of built-in functions that are available as vectorized versions. Built-in mathematical functions, such as `sqrt()` and `sin()` are supported by the Intel *icpc* compiler for vectorization.

Listing 5.1 shows the structure of a *canonical* loop. The code fragment can be easily vectorized. Listing 5.2 shows two nested loops where the inner loop cannot be parallelized due to the data conflicts. On one hand, the inner loop writes two variables. First, the scalar `k` is modified, which cannot be vectorized. The parallel writing of the individual vector units causes conflicts. Second, the writing of `array[i]` cannot be parallelized either. The inner loop iterates over `j` and the writing of `array[i]` is interpreted as writing of a scalar. On the other hand, the outer loop *can* be parallelized. First, the variable `k` is a local variable in the loop body. Consequently, this variable is independent from the outer loop iterations. Second, the inner loop wont be parallelized when the outer loop is parallelized.

```
1 int array[10];
2 for (int i = 0; i < 10; i++) {
3   array[i] = 42;
4 }
```

Listing 5.1: Canonical loop

```
1 int array[10];
2 for (int i = 0 ; i < 10; i++) {
3   int k;
4   for (int j = 0; j < 10; j++) {
5     k++;
6     array[i] = 42;
7   }
8 }
```

Listing 5.2: Nested loop

Our heuristic analysis produces a list of loops which are potential candidates for vectorization. To confirm this, the designer must annotate them in the source code. Specifically, a `#pragma simd` is inserted before the loop to be vectorized. We should emphasize that this manual interaction is necessary because data-level parallelization requires application knowledge that only the designer can provide. The underlying compiler (the Intel icpc) can cover only C++ semantics. Our proposed compiler adds SystemC interpretation, but

application-specific knowledge can only come from the human designer.

Algorithm 3 shows our proposed heuristic to identify data-level parallelism in a SystemC application. The recursive algorithm takes as input a statement s and analyzes if all reachable statements from s are vectorizable. The return value is a tuple where the first element indicates whether or not the statements are vectorizable. The second element is the list of the reachable statements. We provide as input statement the function body of each individual simulation thread. This information is only available with the support of our SystemC-aware compiler which also performs the thread-level analysis.

Algorithm 3 Identification of SIMD loop candidates

```
1: function VECTORIZABLE(s)
   returns (isVectorizable, expressions)
2:     if s ∈ {goto, continue, break, label} then
3:         return (NotVectorizable, ∅)
4:     end if
5:     if s ∈ {variabledeclaration, expression} then
6:         (r1, e1) ← ( Vectorizalbe, EXPR(s))
7:         for all f ∈ FUNCCALLS(s) do
8:             (r2, e2) ← VECTORIZABLE(GETFUNCDEFBODY(f))
9:             (r1, e1) ← (r1 ⊕ r2, e1 ∪ e2)
10:        end for
11:        return (r1, e1)
12:    end if
13:    if s ∈ {if} then
14:        (r1, e1) ← VECTORIZABLE(IFBLOCK(s))
15:        (r2, e2) ← VECTORIZABLE(ELSEBLOCK(s))
16:        return (r1 ⊕ r2, CONDITION(s) ∪ e1 ∪ e2)
17:    end if
18:    if s ∈ {switch} then
19:        for all c ∈ CASES(s) do
20:            VECTORIZABLE(c)
21:        end for
22:        return (NotVectorizable, ∅)
23:    end if
24:    if s ∈ {compound} then
25:        (r1, e1) ← (Vectorizable , ∅)
26:        for all s′ ∈ STATEMENTS(s) do
27:            (r2, e2) ← VECTORIZABLE(s′)
28:            (r1, e1) ← (r1 ⊕ r2, e1 ∪ e2)
29:        end for
30:        return (r1, e1)
31:    end if
```

```
32:     if s ∈ {for, while, do-while} then
33:         if ¬IsCanonical(s) then
34:             return (NotVectorizable, ∅)
35:         end if
36:         (r_b, e_b) ← Vectorizable(Body(s))
37:         r_c ← ConflictCheck(e_b), IncrVar(s))
38:         if (r_b = Vectorizable ∧ r_c = Vectorizable) ∨
39:            (r_c = Vectorizable ∧
40:            r_b ∈ {ScalarConflict, VectorConflict}) then
41:             Recommend this loop for vectorization
42:             return (Vectorizable, e_b)
43:         end if
44:         if r_c = VectorConflict ∧ r_b = VectorConflict then
45:             return (VectorConflict, e_b)
46:         end if
47:         if r_c = ScalarConflict ∧ r_b = ScalarConflict then
48:             return (ScalarConflict, e_b)
49:         end if
50:         return (NotVectorizable, ∅)
51:     end if
52: end function
```

In Algorithm 3, our classification in the first tuple element has four different states which belong to three different classes. The classes are *Vectorizable*, *NotVectorizable*, and *Maybe* where *Maybe* includes *ScalarConflict* and *VectorConflict*. *NotVectorizable* considers the situations where the statement s includes directly or indirectly control flow statements such as `switch`, `goto`, `break`, or `label`. These jump statements cannot be vectorized. The class *Maybe* considers expressions which cannot be vectorized for the current loop but potentially for enclosing loops. `ScalarConflict` reflects the situation in Listing 5.2 in Line 5. The loop writes the scalar `k` and cannot be vectorized. However, a potential outer loop can be vectorized as in Line 4. *VectorConflict* describes a similar situation where an array is written but the loop variable does not match the array index variable, as loop variable `j` and array index `i` in Listing 5.2. The status *Vectorizable* indicates that the statement is suitable for vectorization.

We introduce the operator $\oplus(r_1, r_2) \rightarrow r_3$ where all operands belong to any classification. The result is the more conflicting classification e.g. $Vectorizable \oplus ScalarConflict \rightarrow ScalarConflict$. If the result belongs to the set *Maybe* it can be either *ScalarConflict* or *VectorConflict*. We decided to maintain both types instead of merging them into one, in order to the provide more detailed information to the designer.

The core of the algorithm is the if-clause in Line 32 where we analyze if a loop is applicable for vectorization. Specifically, three conditions must be satisfied. First, the loop iterations must be countable. The function IsCanonical checks if this is the case. Second, all nested statements must be supported. The recursive function call in Line 36 analyzes all nested statements. Third, we have to check if all expressions are conflict-free. The function ConflictCheck analyzes if all write operations target local variables or vectorizable arrays. A loop is vectorizable under two circumstances: a) the second and the third check return *Vectorizable*. b) the second check may result in *ScalarConflict* or *VectorConflict*, but the result for the third is *Vectorizable*. This describes the situation where the inner loop is not vectorizable but the outer is. Otherwise, we distinguish if the result belongs to the class *Maybe* or *NotVectorizable*.

5.2.3 Tool Flow

First, our SystemC compiler reads the design `design.cpp` and translates it into a thread-safe design `risc_design.cpp` to achieve thread-level parallelism. Additionally, our compiler provides to the designer information with candidates for loop vectorization to exploit data-level parallelism in the terminal. After selecting from the provided source code locations, the user inserts the statement `#pragma simd` in front of the chosen loops. Finally, the design `risc_design.cpp` is compiled with the Intel *icpc*.

The feedback of the designer is needed. An example is the following C function:

```
1 void add(float *a,float *b,float *c,int n)
2 {
3   for (int i = 0; i < n; i++) {
4     a[i] = a[i] + b[i] + c[i];
5   }
6 }
```

Listing 5.3: Nested loop

Here, arrays passed as pointers can only be vectorized if the user asserts that there is no vector dependence in the way. This is only possible with application knowledge, not by static compiler analysis. Our proposed compiler, which is aware of SystemC and its concurrent multi-threading semantics, can identify this loop as a potential candidate, but the final data independence assertion must come from the user who knows the application specifics (i.e. the pointers point to non-overlapping arrays).

5.3 Experiments and Results

We have implemented the approach outlined above and demonstrate the combined speedup of *thread* and *data-level parallelism* on two different applications, namely a particle simulator on a NoC and a video Mandelbrot renderer unit. For both applications, we measure the sequential, the sequential with SIMD, the parallel, and the parallel with SIMD simulator run times. On one hand, the experiments execute on an Intel Xeon E3-1240 multi-core processor with 4 CPU cores. Each core has one simulation thread with a vectorization unit of 256 bit width. On the other hand, we use an Intel Xeon Phi™ Coprocessor 5110P many-core architecture. The coprocessor contains 60 cores where each core has a vectorization unit of 512 bit. To obtain unambiguous measurements, we turn CPU frequency scaling off for all experiments.

5.3.1 Network-on-Chip Particle Simulator

As a comprehensive example we select a NoC particle simulator model in SystemC to demonstrate the parallel simulation capabilities of our *thread* and *data-level parallelism* analysis. The abstract architecture of the particle simulator is illustrated in Figure 5.1. The grid is assembled of tiles where each tile communicates bidirectionally with a tile to its north, south, east, or west. For a 4x4 example, we have 16 tiles and one grid module. Each tile has one thread which computes the motion of the individual particles in a certain area of the model. The particles move continuously in 2D space. The moment when a particle crosses the boundary, the responsibility of computing and updating the position of the particles shifts from one tile to its neighbor tile. The entire design can be scaled up to any quadratic size. The user can configure via the command line the number of tiles as well as the number of particles, the gravity, and other options.

At the beginning of the simulation, the testbench sends the initial particles to each individual tile. This happens in a purely sequential fashion. Next, all tiles simulate the particles and then synchronize with their neighbor tiles. A tile is blocked when it communicates with one of its neighbors. At the end of the simulation, all the tiles send the particle positions back to the testbench.

First, our compiler performs a thread-level analysis of the design. Each thread is identified and analyzed for race conditions with all other threads in the design. The resulting conflict table is then passed to our parallel SystemC library which performs out-of-order scheduling.

Next, our infrastructure creates and inspects the call graph of all possible threads for vectorization. The *data-level parallelism* analysis generates a list of potential locations for the vectorization. After inspection of the list, the function `apply_force()` is selected for parallelization. This function computes the gravity influence of the individual particles on each other. The Intel *icpc* compiler cannot directly identify this function for vectorization. Our

analysis builds a call graph for each thread and filters it for possible candidate locations. Suitable candidates are vetted by the designer using his application knowledge. To confirm vectorization, the designer adds `#pragma simd` before the for-loop in question.

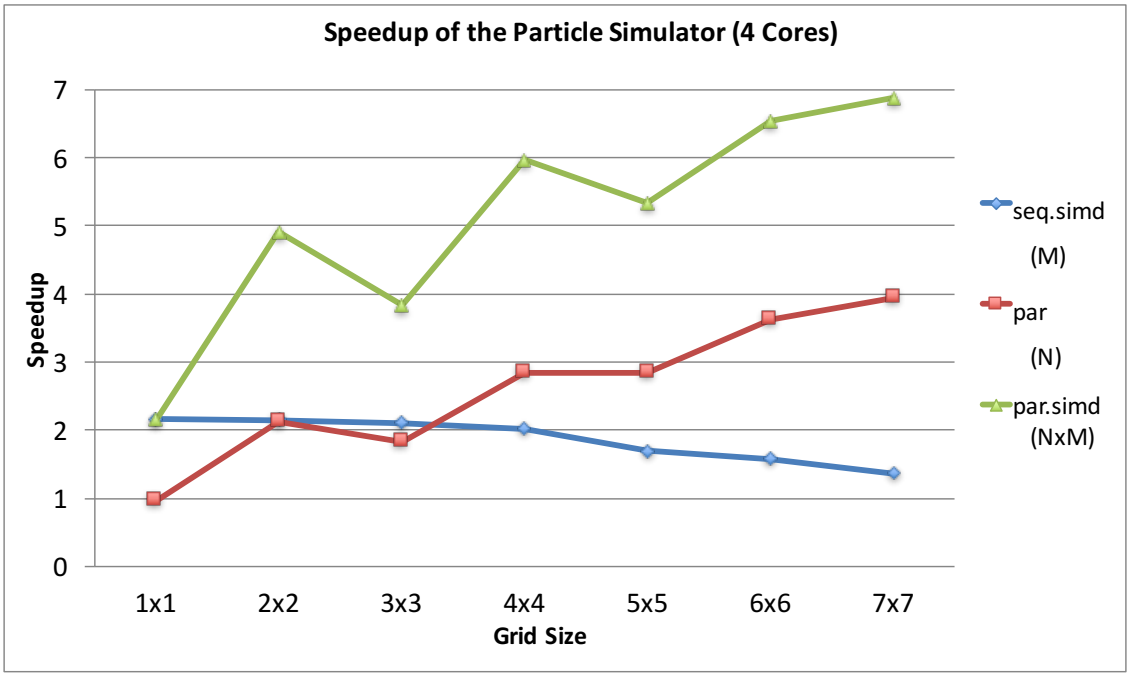


Figure 5.4: Speedup of the NoC Particle Simulator

Experiment on the Multi-Core Host

Figure 5.4 shows the speedup for the particle simulator on the four core machine. First, the blue line (diamonds) shows the speedup M for the sequential SIMD simulation between 1.6x and 2.1x. This confirms our SIMD discussion above where the maximal speedup of 4x cannot be reached due to the needed overhead of packing and unpacking the lanes. The increasing communication of particles among tiles results in lower parallelism (Amdahl's law) which explains the decreasing speedup. For a 7x7 grid size, 49 threads are active and communicating synchronously bidirectional to the associated neighbor threads. So, intensive core to core communication limits the parallelism.

The red line (squares) shows the thread-level speedup N, generally increasing with higher

grid sizes. The measurement and the resulting total speedup ($N \times M$) shown in green (triangles) show a zig-zag pattern. Grid sizes with an even number of rows and columns perform better than the grid sizes with odd number of rows and columns. This phenomenon is due to the implementation of the particle simulator, in particular due to its communication characteristics. Figure 5.5 compares the communication pattern of a 3x3 and 4x4 grid.

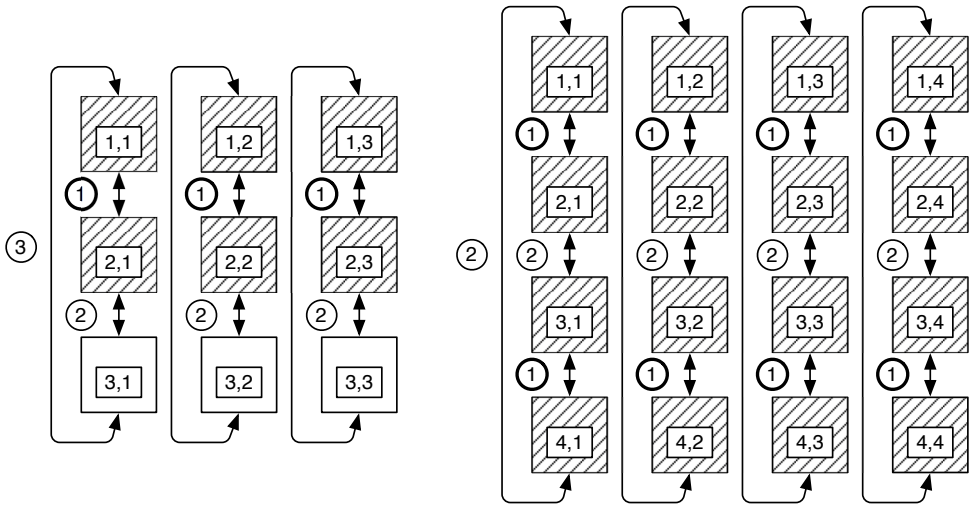


Figure 5.5: Communication pattern of the Particle Simulator

First, for the 4x4 grid the odd rows are communicating with the neighbors to the south (indicated as ①). In this case, eight communication pairs are active at the same time. Next, the even rows communicate to their south neighbors. Here, the last row communicates with the first row. Again, eight pairs are available for the parallel simulation (indicated as ②). So, at any point all four cores are fully utilized. The same applies for the horizontal communication.

For the 3x3 grid, the simulation has different characteristics. First the odd rows start communicating which is only the first row ①. Three pairs are available for execution which means that only three out of four cores are used. Next, the odd rows communicate ②. Again, only three pairs are available, one core stays idle. Finally, the last row communicates with the first row ③. In all phases, only three out of four cores are utilized. All over, the

4x4 grid can gain higher speed up due to the higher core utilization.

This explanation applies for the other gird sizes as well. For instance, the 6x6 grid size with 36 threads can be better utilized than a 5x5 with 25 threads on a 4 core machine.

Finally, the green (triangle) line shows the combination of the parallel and SIMD technique essentially the product of $N \times M$. The product of the sequential with SIMD and parallel simulation show the combined speedup. The maximal speedup is 6.8x which is impressive on 4 cores.

Experiment on the Many-Core Host

We simulated the particle simulator on the many-core architecture as well. The Intel Xeon Phi 5110P Coprocessor hardware has a ring architecture of cores. Two cores are communicating over a third core which hosts as a so-called tag directory. This communication scheme causes a high traffic congestion. In our simulations, the speedup is marginal (below 5x) and constantly decreases with the increasing number of threads. Similar results and the importance of a sophisticated thread to core mapping have been shown in [31] for this specific architecture. So, we decide not to investigate the NoC benchmark further on this platform.

5.3.2 Mandelbrot Renderer

The Mandelbrot renderer is a parallel video application to compute the Mandelbrot set [32]. Basically, the DUT hosts a number of renderer units. Each unit computes a different slice of the Mandelbrot image. At compile time, the user defines how many slices are available. During the simulation, the DUT provides coordinates to the individual slices. A slice computes the Mandelbrot set for the given coordinates and sends the results back. The DUT receives the data from each unit and stores them. Finally, new coordinates are provided

to all slices for the next frame. In contrast to the NoC particle simulator where tiles are intensively communicating, the individual renderer units are fully independent.

Our compiler automatically applies the *thread-level parallelism* to the individual threads of the renderer units. The *data-level parallelism* analysis identifies the central loop in the function `mandel_row()` as a candidate for vectorization, which we confirm.

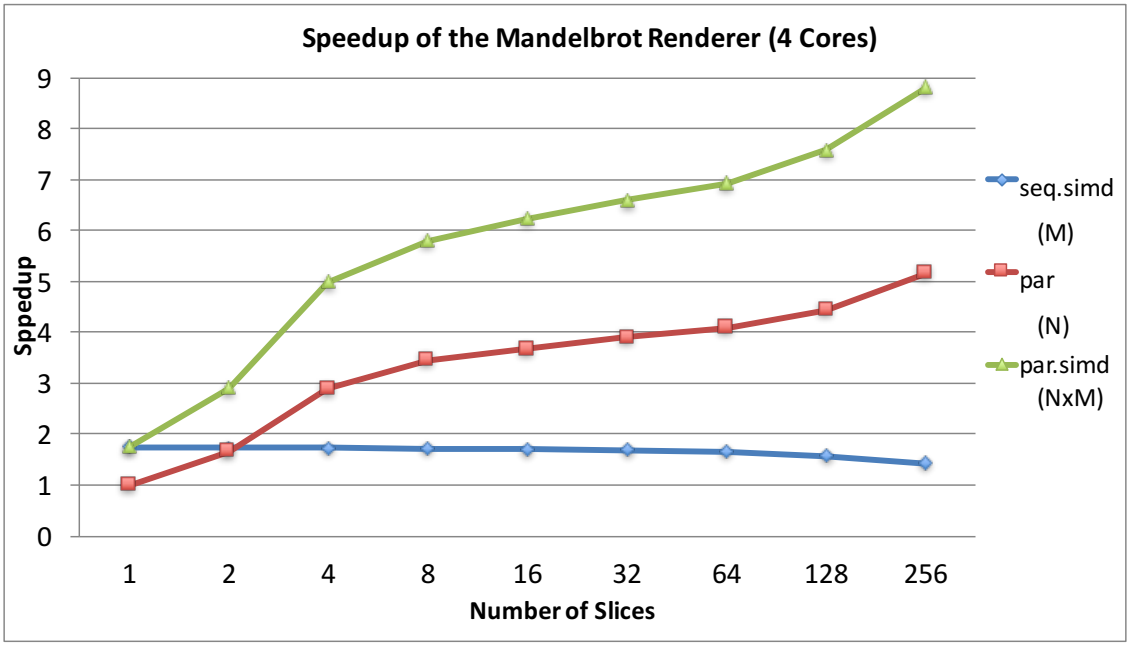


Figure 5.6: Speedup of the Mandelbrot Renderer

Experiment on the Multi-Core Host

Figure 5.6 shows the simulation speedup for the Mandelbrot renderer with up to 256 different units on the 4 core machine. First, the sequential simulation with SIMD support (blue diamonds) achieves a speedup M of about 1.7x. The increasing number of slices slightly affect the speedup. Through the increasing number of threads in combination with the data need of the vectorization units, higher memory traffic occurs. Next, the *thread-level parallelism* (red squares) provides a speedup N of nearly 5.1x. This super-linear speedup is possible due to the poor cache utilization of sequentially scheduling 256 threads on the 4-core

machine. Table 5.1 shows the increase from 41.5 seconds to 59 seconds for this sequential reference case. Finally, the combination of the *thread* and *data-level parallelism* $N \times M$ reach a total of up to 8.8x.

Table 5.1: Simulation speedup for the Mandelbrot renderer on 4 core host architecture.

Slices	4 Core Machine										
	Execution Time / CPU Utilization								Speedup		
	seq		seq.simd		par		par.simd		seq.simd	par	par.simd
1	41.53	100%	23.80	99%	41.74	99%	23.88	99%	1.74	0.99	1.74
2	41.66	99%	23.99	99%	25.14	165%	14.32	166%	1.74	1.66	2.91
4	41.91	99%	24.22	99%	14.53	297%	8.39	294%	1.73	2.88	5.00
8	42.36	99%	24.61	99%	12.28	354%	7.31	341%	1.72	3.45	5.79
16	43.07	99%	25.23	99%	11.74	372%	6.90	361%	1.71	3.67	6.24
32	44.58	99%	26.51	99%	11.44	381%	6.76	369%	1.68	3.90	6.59
64	46.69	99%	28.05	99%	11.41	382%	6.73	371%	1.66	4.09	6.94
128	50.85	99%	32.42	99%	11.46	381%	6.70	373%	1.57	4.44	7.59
256	58.99	99%	41.37	99%	11.43	382%	6.69	374%	1.43	5.16	8.82

Experiment on the Many-Core Host

Finally, we simulate the Mandelbrot renderer on the Intel Xeon Phi many-core architecture. Figure 5.7 shows the simulation results. Due to the minimal communication needs compared to the particle simulator, highest speedups are reached. The vectorization unit with 512 bit can execute up to eight double-precision floating-point operations in parallel. A speedup M of 6.9x is achieved. The thread-level parallelization increases strongly on the 60 cores with a speedup N of 50x. Afterwards, the speed slows down. Due to the 60 physical cores and use of hyper threads. Table 5.2 shows the increase from 393.97 seconds to 1.85 seconds for this sequential reference case. Finally, the combination of the thread and data level parallelization $N \times M$ generates a speedup of up to 212x.

Table 5.2: Simulation speedup for the Mandelbrot renderer on 60 core host architecture.

Slices	60 Core Machine with 4 Hyperthreads Each						
	Execution Time				Speedup		
	seq	seq.simd	par	par.simd	seq.simd	par	par.simd
1	393.76	56.94	393.52	56.70	6.92	1.00	6.94
2	393.75	56.92	234.15	33.44	6.92	1.68	11.77
4	393.79	56.92	129.45	18.58	6.92	3.04	21.19
8	393.75	56.92	67.36	9.82	6.92	5.85	40.10
16	393.77	56.93	34.62	5.43	6.92	11.37	72.52
32	393.80	56.97	18.47	2.87	6.91	21.32	137.21
64	393.90	57.12	9.59	1.89	6.90	41.07	208.41
128	393.97	57.14	8.51	1.85	6.89	46.29	212.96
256	394.20	57.35	7.90	2.03	6.87	49.90	194.19

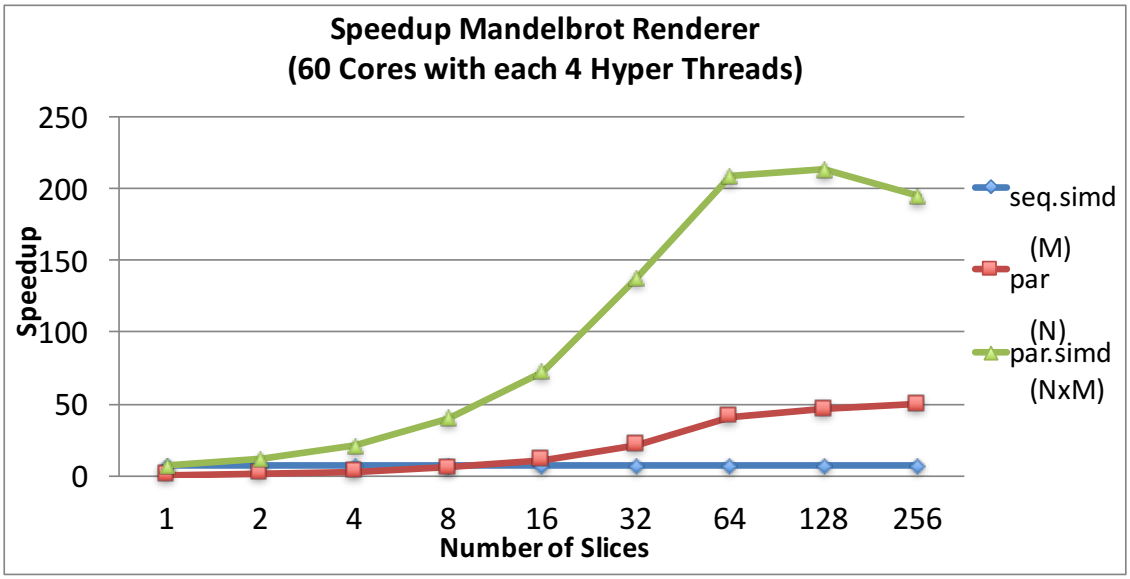


Figure 5.7: Speedup of the Mandelbrot renderer on a many core architecture

5.4 Summary

In this chapter, we present our compiler infrastructure for the automatic parallelization of SystemC models at the *thread level*. Additionally, our infrastructure performs an analysis to identify source locations that are suitable for *data-level parallelization*. The vectorization

analysis depends strongly on our SystemC aware compiler which identifies the simulation threads of the individual models and channels. To the best of our knowledge, this work is the first to apply and exploit SIMD vectorization on top of *thread-level* parallel SystemC simulation. We demonstrated our techniques on NoC particle simulator and Mandelbrot renderer SystemC benchmarks on a multi- and many-core architecture. On the 4 core machine, we achieved a speedup of up to 8.8x through the combination of *thread* and *data-level parallelism*. On the 60 core architecture, we gained up to 212x of simulation speedup.

Chapter 6

Conclusion

In this chapter, we discuss the contributions of this dissertation and briefly summarize them. Additionally, we discuss the future work of the RISC compiler infrastructure.

6.1 Contributions

The key contributions of this dissertation are the following:

1. Creation of an open source compiler framework for analysis and parallel simulation of IEEE SystemC models
2. Automatic generation of SystemC thread communication charts to visualize thread and module dependencies [48]
3. Advanced conflict analysis for channel instance IDs and C++ references to distinguish variables for reduced false positive conflicts [47]
4. Hybrid analysis approach to support 3rd party library code as well as dynamic elabo-

ration of module hierarchies [50]

5. Identification and exploitation of opportunities to exploit thread and data level parallelism for PDES [49]

6.1.1 Thread Communication Graphs

In Chapter 2, we addressed the time-consuming task of refining and documenting legacy and 3rd party source code. We proposed to automatically extract graphical charts from given SystemC code to ease the understanding of the source code with a visual representation. Specifically, we extracted the communication flow between the threads from the design model by use of an automatic SystemC compiler infrastructure that statically analyzes the code.

6.1.2 Improved Static Analysis

In Chapter 3, we examined the effect of false positive conflicts in the data and event notification table for the simulation speed. Specifically, we discussed the treatment of channel variables and references. For channel variables we introduced the concept of the PCP to identify the instance ID of individual channels. For references, we provided a combination of AST annotation and the PCP analysis to determine the mapped reference variables. Through these techniques we were able to eliminate up to 98% of false positive conflicts and gain a speedup of 4x in simulator performance.

6.1.3 Hybrid Analysis

In Chapter 4, we discussed the need for a hybrid analysis in context of static analysis for SystemC models. On one hand, the source code of libraries is often not available, however, it

is needed for the static analysis. We provided an annotation scheme for libraries to provide information for the static analysis. On the other hand, during the design space exploration designers tend to provide model configurations dynamically. For instance, the number of rows and columns of a NoC architecture is provided via command line arguments. We proposed a dynamic analysis to extract structural information and used this information later in the static analysis. We gained a speedup of up to 6x for a model which is not analyzable without our newly introduced hybrid analysis.

6.1.4 Thread and Data Level Parallelism

In Chapter 5, we extended the traditional PDES and added data level parallelism to it. Our RISC compiler infrastructure analyzes the model for potential loops for vectorization. As the result, the designer gets a list of potential loops for vectorization and has to decide which loops are applicable. We gained a speedup of 8x on a multi-core architecture with 4 cores and a speedup of 212x on a many-core architecture with 60 cores.

6.1.5 Open Source Release

Unfortunately, in academia open source releases are rare because they are time costly and require a high level of maintenance. In contrast, this dissertation provides all individual contributions in an open source software package. Based on the given Linux C++ platform and the 3rd party ROSE compiler infrastructure, we have implemented the SystemC IR, the Segment Graph Generator, and the Backend Analysis software layers shown in Figure 1.5. The RISC compiler framework had four open source releases with associated technical reports over the last years. The current version is freely available at [30] and is still under maintenance and extension by continuing group members.

Comprehensive Test Suite

Another major engineering contribution of this dissertation is the testing infrastructure. In collaboration with undergraduate students hundreds, of test cases for the SG generation have been created. These test cases consider all kinds of loops, if-else statements, nested control flow statements, member function calls, recursive function calls, global function calls, and many other aspects. Additionally, hundreds of test cases for the data conflict and event notification table have been created. These test cases consider variables in many contexts e.g. function parameter references, module references, global and static variables, local and member variables, namespace variables, variables in context of instance IDs, and other situations. All these test cases run automatically and are used as regressions tests to maintain the correct functionality of the RISC software package with every release.

6.2 Future Work

The overview and the future work of the RISC compiler infrastructure is sketched in Figure 1.15 on page 23.

In the current release, the RISC compiler does not support TLM 2.0 models which contain, among other things, direct memory accesses. Additionally, the concept of channels disappeared because they became modules which are connected to each other directly. This technique aims at reducing the amount of context switches and gain simulation speed in a sequential simulator. The RISC compiler requires communication between modules through channels right now. The diagnostic for the conflict analysis should be extended to simulate TLM 2.0 models in parallel as well.

Another aspect of future work is the handling of *partial* SGs. So far, the entire SystemC model must be in one single file for instrumentation. However, library providers do not

provide all of their source code and keep it protected. Thus it is needed to provide partial SGs along with header files which are integrated into one complete SG later.

Bibliography

[1] Documentation: Intel Corporation - Requirements for Vectorizable Loops. https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops. Accessed: 2016-08-09.

[2] IEEE POSIX 1003.1c standard. *IEEE Standard 1003.1c*, 1995.

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

[4] Press Release: Qualcomm Datacenter Technologies Announces Commercial Shipment of Qualcomm Centriq 2400 – The World's First 10nm Server Processor and Highest Performance Arm-based Server Processor Family Ever Designed. https://www.qualcomm.com/news/releases/2017/11/08/qualcomm-datacenter-technologies-announces-commercial-shipment-qualcomm, November 2017.

[5] Accellera Systems Initiative Webpage. http://www.accellera.org, 2018.

[6] Dot language. http://www.graphviz.org/doc/info/lang.html, April 2018.

[7] Doxygen. http://www.stack.nl/~dimitri/doxygen, April 2018.

[8] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[9] D. C. Black and J. Donovan. *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, December 2005.

[10] N. Blanc, D. Kroening, and N. Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *Proceedings of the 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary*, March 2008.

[11] N. Bombieri, F. Fummi, and S. Vinco. On the Automatic Generation of GPU-oriented Software Applications from RTL IPs. In *Proceedings of The International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS), Montreal, Canada*, September 2013.

[12] K. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Proceedings of the IEEE Transactions on Software Engineering*, (5):440–452, September 1979.

[13] W. Chen. Out-of-order Parallel Discrete Event Simulation for Electronic System-Level Design, University of California Irvine, USA. In *Ph.D. Thesis*, March 2013.

[14] W. Chen and R. Dömer. An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASPDAC), Sydney, Australia*, January 2012.

[15] W. Chen and R. Dömer. Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions. In *Proceedings of the 16th Design, Automation and Test in Europe (DATE) Conference, Grenoble, France*, March 2013.

[16] W. Chen, X. Han, and R. Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proceedings of the 15th Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2012.

[17] W. Chen, X. Han, and R. Dömer. May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models. In *Proceedings of the 17th Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2014.

[18] W. Chen, X. Han, and R. Dömerömer. Multicore Simulation of Transaction-Level Models Using the SoC Environment. *IEEE Transactions on Design & Test of Computers*, 28:20–31, May 2011.

[19] Z. Cheng and R. Dömer. A SystemC Model of a Bitcoin Miner. University of California Irvine, USA, September 2016.

[20] The Cilk Project. `http://supertech.csail.mit.edu/cilk/`, April 2018.

[21] R. Dömer. System-level Modeling and Design with the SpecC Language, University of Dortmund, Germany. In *Ph.D. Thesis*, April 2000.

[22] R. Dömer, G. Liu, and T. Schmidt. *Handbook of Hardware/Software Codesign*, chapter Parallel Simulation. Springer Netherlands, June 2017.

[23] R. M. Fujimoto. Parallel Discrete Event Simulation. *Commun. ACM*, 33(10), October 1990.

[24] A. Gerstlauer, R. Dömer, J. Peng, and D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, Bosten, Dordrecht, London, June 2001.

[25] G. Glaser, G. Nitschey, and E. Hennig. Temporal Decoupling with Error-Bounded Predictive Quantum Control. In *Proceedings of the Forum on specification and Design Languages (FDL), Barcelona, Spain*, September 2015.

[26] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[27] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably Distributed SystemC Simulation for Embedded Applications. In *Proceedings of the 3rd International Symposium on Industrial Embedded Systems (SIES), La Grande Motte, France*, June 2008.

[28] A. Kaushik and H. D. Patel. Systemc-clang: An Open-source Framework for Analyzing Mixed-abstraction Systemc Models. In *Proceedings of the Forum on specification and Design Languages (FDL), Paris, France*, September 2013.

[29] G. Liu. Optimizing Many-Threads-to-Many-Cores Mapping in Parallel Electronic System Level Simulation, University of California Irvine, USA. In *Ph.D. Thesis*, March 2017.

[30] G. Liu, T. Schmidt, and R. Dömer. RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset. Center for Embedded and Cyber-physical Systems (CECS), University of California Irvine, USA, July 2017.

[31] G. Liu, T. Schmidt, R. Dömer, A. Dingankar, and D. Kirkpatrick. Optimizing Thread-to-Core Mapping on Manycore Platforms with Distributed Tag Directories. In *Proceedings of the 20th Asia and South Pacific Design Automation Conference (ASPDAC), Chiba/Tokyo, Japan*, January 2015.

[32] B. Mandelbrot. Fractal aspects of the iteration of $z \rightarrow \lambda z(1-z)$ for complex λ and z. Annuals of the New York Academy of Sciences, 1980.

[33] K. Marquet and M. Moy. PinaVM: A SystemC Front-End Based on an Executable Intermediate Representation. In *Proceedings of the 10th International Conference on Embedded Software (EMSOFT)*, October 2010.

[34] S. Mauw and M. Reniers. High-level Message Sequence Charts. In *Proceedings of the 8th SDL Forum, Evry, France*, September 1997.

[35] M. Moy, F. Maraninchi, and L. Maillet-Contoz. Pinapa: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip. In *Proceedings of the 5th International Conference on Embedded Software (EMSOFT)*, September 2005.

[36] W. Mueller, R. Dömer, and A. Gerstlauer. The Formal Execution Semantics of SpecC. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, October 2002.

[37] D. Nicol and P. Heidelberger. Parallel Execution for Serial Simulators. *ACM Transactions on Modeling and Computer Simulation*, 6(3):210–242, July 1996.

[38] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[39] OpenMP. `https://computing.llnl.gov/tutorials/openMP/`, April 2018.

[40] H. D. Patel, D. Mathaikutty, D. Berner, and S. K. Shukla. CARH: Service-Oriented Architecture for Validating System-Level Designs. *Processings of the IEEE Transactions on CAD of Integrated Circuits and Systems*, 25(8):1458–1474, August 2006.

[41] Posix Threads Programming. `https://computing.llnl.gov/tutorials/pthreads/`, April 2018.

[42] D. J. Quinlan. ROSE: Compiler Support for Object-Oriented Frameworks. *Parallel Processing Letters*, 10(2/3), 2000.

[43] S. Roopak, R. Parthasarathi, and B. Samik. *Correct-by-Construction Approaches for SoC Design*. Springer, New York, Berlin, Paris, 2004.

[44] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker. Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures. In *Processdings of the 17th Euromicro Conference Digital System Design (DSD)*, August 2014.

[45] B. C. Schäfer and A. Mahapatra. S2CBench: Synthesizable Systemc Benchmark Suite for High-Level Synthesis. *Embedded Systems Letters*, 6(3):53–56, 2014.

[46] T. Schmidt. A Program State Machine Based Virtual Processing Model in SystemC, University of Oldenburg, Germany. In *Master Thesis*, March 2013.

[47] T. Schmidt, Z. Cheng, and R. Dömer. Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation. In *Proceedings of the 21st Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2018.

[48] T. Schmidt, G. Liu, and R. Dömer. Automatic Generation of Thread Communication Graphs from SystemC Source Code. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES), Sankt Goar, Germany*, May 2016.

[49] T. Schmidt, G. Liu, and R. Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In *Proceedings of the 54th Design and Automation Conference (DAC), Austin, USA*, June 2017.

[50] T. Schmidt, G. Liu, and R. Dömer. Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation. In *Proceedings of the 22nd Asia and South Pacific Design Automation Conference (ASPDAC), Chiba/Tokyo, Japan*, January 2017.

[51] R. Sinha, A. Prakash, and H. D. Patel. Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs. In *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASPDAC), Sydney, Australia*, January 2012.

[52] Supercomputing Technologies Group MIT Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, 1998.

[53] N. Ventroux and T. Sassolas. A New Parallel SystemC Kernel Leveraging Manycore Architectures. In *Proceedings of the 19th Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2016.

[54] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann. SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments. In *Proceedings of the 19th Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2016.

[55] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto. Time-Decoupled Parallel SystemC Simulation. In *Proceedings of the 17th Design, Automation and Test in Europe (DATE) Conference, Dresden, Germany*, March 2014.