UNIVERSITY OF CALIFORNIA,
IRVINE

Out-of-order Parallel Discrete Event Simulation for Electronic System-Level Design

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Weiwei Chen

Dissertation Committee:
Professor Rainer Doemer, Chair
Professor Daniel D. Gajski
Professor Brian Demsky

2013

© 2013 Weiwei Chen

DEDICATION

To my parents.

Contents

LIST OF FIGURES vi

LIST OF TABLES viii

ACKNOWLEDGMENTS xii

CURRICULUM VITAE xiv

ABSTRACT OF THE DISSERTATION xviii

1 Introduction 1
- 1.1 System-level Design . . . 4
  - 1.1.1 Levels of Abstraction . . . 6
  - 1.1.2 The Y-Chart . . . 7
  - 1.1.3 System-level Design Methodologies . . . 8
  - 1.1.4 Electronic System-Level Design Process . . . 11
- 1.2 Validation and Simulation . . . 14
  - 1.2.1 Language Support for System-level Design . . . 14
  - 1.2.2 System Simulation Approaches . . . 17
  - 1.2.3 Discrete Event Simulation . . . 18
- 1.3 Dissertation Goals . . . 21
- 1.4 Dissertation Overview . . . 22
- 1.5 Related Work . . . 23
  - 1.5.1 The SpecC Language . . . 24
  - 1.5.2 The SystemC Language . . . 31
  - 1.5.3 The System-on-Chip Environment (SCE) Design Flow . . . 32
  - 1.5.4 Multi-core Technology and Multithreaded Programming . . . 35
  - 1.5.5 Efficient Model Validation and Simulation . . . 37

2 The ConcurrenC Model of Computation 40
- 2.1 Motivation . . . 40
- 2.2 Models of Computation (MoCs) . . . 42
- 2.3 ConcurrenC MoC . . . 44
  - 2.3.1 Relationship to C-based SLDLs . . . 45
  - 2.3.2 ConcurrenC Features . . . 46
  - 2.3.3 Communication Channel Library . . . 47

2.3.4 Relationship to KPN and SDF . . . 48
2.4 Case Study . . . 49

3 Synchronous Parallel Discrete Event Simulation 52
3.1 Traditional Discrete Event Simulation . . . 52
3.2 SLDL Multi-threading Semantics . . . 55
3.2.1 Cooperative multi-threading in SystemC . . . 55
3.2.2 Pre-emptive multi-threading in SpecC . . . 56
3.3 Synchronous Parallel Discrete Event Simulation . . . 58
3.4 Synchronization for Multi-Core Parallel Simulation . . . 59
3.4.1 Protecting Scheduling Resources . . . 59
3.4.2 Protecting Communication . . . 60
3.4.3 Channel Locking Scheme . . . 62
3.4.4 Automatic Code Instrumentation for Communication Protection . . . 63
3.5 Implementation Optimization for Multi-Core Simulation . . . 66
3.6 Experiments and Results . . . 67
3.6.1 Case Study on a H.264 Video Decoder . . . 67
3.6.2 Case Study on a JPEG Encoder . . . 72

4 Out-of-order Parallel Discrete Event Simulation 74
4.1 Motivation . . . 74
4.2 Out-of-order Parallel Discrete Event Simulation . . . 76
4.2.1 Notations . . . 76
4.2.2 Out-of-order PDES Scheduling Algorithm . . . 79
4.3 Out-of-order PDES Conflict Analysis . . . 81
4.3.1 Thread Segments and Segment Graph . . . 81
4.3.2 Static Conflict Analysis . . . 86
4.3.3 Dynamic Conflict Detection . . . 91
4.4 Experimental Results . . . 92
4.4.1 An Abstract Model of a DVD Player . . . 93
4.4.2 A JPEG Encoder Model . . . 94
4.4.3 A Detailed H.264 Decoder Model . . . 95

5 Optimized Out-of-order Parallel Discrete Event Simulation 97
5.1 Optimized Compiler Using Instance Isolation . . . 98
5.1.1 Motivation . . . 98
5.1.2 Instance Isolation without Code Duplication . . . 101
5.1.3 Definitions for the Optimized Static Conflict Analysis . . . 103
5.1.4 Algorithm for Static Conflict Analysis . . . 104
5.1.5 Experimental Results . . . 107
5.2 Optimized Scheduling Using Predictions . . . 109
5.2.1 State Prediction to Avoid False Conflicts . . . 109
5.2.2 Static Prediction Analysis . . . 112
5.2.3 Out-of-order PDES scheduling with Predictions . . . 117

5.2.4 Optimized out-of-order PDES Scheduling Conflict Checking with a Combined Prediction Table . . . 118
5.2.5 Experimental Results . . . 119

6 Comparison and Outlook **123**
6.1 Experimental Setup . . . 123
6.1.1 Experimental Environment Setup . . . 123
6.1.2 The Parallel Benchmark Models . . . 125
6.1.3 The Embedded Applications . . . 127
6.2 Parallel Discrete Event Simulation Overlook . . . 131

7 Utilizing the Parallel Simulation Infrastructure **136**
7.1 Introduction . . . 137
7.2 Overview and Approach . . . 138
7.2.1 Creating Parallel System Models . . . 138
7.2.2 Shared Variables and Race Conditions . . . 139
7.3 Automatic Race Condition Diagnosis . . . 141
7.4 Race Condition Elimination Infrastructure . . . 143
7.4.1 Static Code Analysis . . . 143
7.4.2 Dynamic Race Condition Checking . . . 146
7.5 Experiments and Results . . . 147
7.5.1 Case study: A Parallel H.264 Video Decoder . . . 147
7.5.2 Case study: A Parallel H.264 Video Encoder . . . 149
7.5.3 Additional Embedded Applications . . . 150
7.6 Conclusions . . . 151

8 Conclusions **152**
8.1 Contributions . . . 153
8.1.1 A model of computation for system-level design . . . 153
8.1.2 A synchronous parallel discrete event simulator . . . 153
8.1.3 An advanced parallel discrete event simulation approach . . . 154
8.1.4 An infrastructure for increasing modeling observability . . . 156
8.2 Future Work . . . 156
8.2.1 Model Parallelization . . . 156
8.2.2 Multithreading Library Support . . . 157
8.2.3 Extension to the SystemC SLDL . . . 157
8.2.4 Parallel Full System Validation . . . 157
8.3 Concluding Remarks . . . 158

Bibliography **159**

List of Figures

1.1 Hardware and software design gaps versus time (source [11]) 5
1.2 Levels of abstraction in SoC design (source [18]) 6
1.3 System-level design in the Y-Chart (source [17]) 7
1.4 System-level design methodologies (source [13]) 10
1.5 Scheduler for discrete event simulation 19
1.6 The block diagram and source code of a SpecC model 25
1.7 SpecC behavioral hierarchy block diagram and syntax [18] 26
1.8 Compilation flow for a SpecC Design (source [45]) 31
1.9 SystemC language architecture (source [38]) 32
1.10 Refinement-based design flow in the SCE framework (source [46]) 33

2.1 Relationship between C-based SLDLs SystemC and SpecC, and MoC ConcurrenC 45
2.2 Visualization of a ConcurrenC Model in three spatial and one temporal dimensions 46
2.3 Proposed H.264 Decoder Block Diagram 50

3.1 Traditional sequential Discrete Event simulation scheduler 54
3.2 Synchronous parallel Discrete Event simulation scheduler 59
3.3 Protecting synchronization in channels for multi-core parallel simulation 61
3.4 Synchronization protection for the member functions of communication channels. 63
3.5 Channel definition instrumentation for communication protection 64
3.6 Life-cycle of a thread in the multi-core simulator 65
3.7 Optimized synchronous parallel discrete event simulation scheduler 67
3.8 A parallelized H.264 decoder model. 68
3.9 Parallelized JPEG encoder model. 72

4.1 High-level DVD player model with video and audio streams. 75
4.2 Scheduling of the high-level DVD player model 76
4.3 States and transitions of simulation threads (simplified). 77
4.4 Example source code in SpecC 82
4.5 The control flow graph, segment graph, conflict tables, and time advance tables for the simple design example in Fig. 4.4 84
4.6 Write-after-write (WAW) conflict between two parallel threads. 87
4.7 The abstract DVD player example 93

5.1 Example source code in SpecC 99
5.2 Simple design example with isolated instances. 100

5.3 Function-local segment graphs and variable access lists for the example in Fig. 5.2. 106
5.4 Simple design example. 111
5.5 Out-of-order PDES scheduling. 112
5.6 A partial Segment Graph with Adjacency Matrix and Data Conflict Table. . . . 114
5.7 Data structures for optimized out-of-order PDES scheduling. 115
5.8 An video edge detector example 120
5.9 Simulation run time and compilation time for OoO PDES with predictions . . . 122

6.1 Simulation results for highly parallel benchmark models. 128

7.1 Validation and debugging flow for parallel system models. 140
7.2 Reusing essential tools from a parallel simulation infrastructure to diagnose race conditions in parallel models. 140
7.3 Tool flow for automatic race condition diagnosis among shared variables in parallel system models. 142
7.4 A parallel design example with a simple race condition. 144
7.5 Parallel simulation algorithm with dynamic race condition checks. 146

List of Tables

2.1 System-level design in comparison with the well-established RTL design 41
2.2 Parameterized Communication Channels 48
2.3 Simulation Results, H.264 Decoder modeled in *ConcurrenC* 51

3.1 Simulation results of a well balanced H.264 Decoder ("Harbour", 299 frames 4 slices each, 30 fps). 70
3.2 Comparison with optimized implementation. 71
3.3 Simulation speedup with different h264 streams (spec model). 71
3.4 Simulation results of JPEG Encoder 73

4.1 Time advances at segment boundaries. 90
4.2 Examples for direct and indirect timing hazards 90
4.3 Experimental results for the abstract DVD player example 94
4.4 Out-of-order PDES statistics for JPEG and H.264 examples 95
4.5 Experimental results for the JPEG Image Encoder and the H.264 Video Decoder examples 96

5.1 Experimental Results for H.264 Decoder Models with Different Degree of Isolation. 100
5.2 Compilation time for a Set of JPEG Image Encoder and H.264 Video Decoder Models. 108
5.3 Simulation run time for a Set of JPEG Image Encoder and H.264 Video Decoder Models. 109
5.4 Experimental results for embedded applications 122

6.1 Experimental results for embedded application examples using standard algorithms. 131
6.2 Comparison of traditional sequential, synchronous parallel, and out-of-order parallel discrete event simulation 132

7.1 Experimental results on tool execution time for diagnosing race conditions in embedded multi-media applications 147
7.2 Experimental results on diagnosed race conditions in embedded multi-media applications 148

List of Acronyms

AHB Advanced High-performance Bus. System bus definition within the AMBA 2.0 specification. Defines a high-performance bus including pipelined access, bursts, split and retry operations.

AMBA Advanced Microprocessor Bus Architecture. Bus system defined by ARM Technologies for system-on-chip architectures.

ASIC Application Specific Integrated Circuit. An integrated circuit, chip, that is custom designed for a specific application, as supposed to a general-purpose chip like a microprocessor.

Behavior An encapsulating entity, which describes computation and functionality in the form of an algorithm.

BFM Bus Functional Model. A pin-accurate and cycle-accurate model of a bus (see also PCAM).

CAD Computer Aided Design. Design of systems with the help of and assisted by computer programs, i.e. software tools.

CE Communication Element. A system component that is part of the communication architecture for transmission of data between PEs, e.g. a transducer, an arbiter, or an interrupt controller.

Channel An encapsulating entity, which abstractly describes communication between two or more partners.

DE Discrete Event Simulation.

DFG Data Flow Graph. An abstract description of computation capturing operations (nodes) and their dependencies (operands).

DSP Digital Signal Processor. A specialized microprocessor for the manipulation of digital audio and video signals.

EDA Electronic Design Automation. A category of software tools for designing electronic systems.

ESL Electronic System Level. Electronic system level design and verification is an emerging electronic design methodology that focuses on the higher abstraction level concerns first and foremost.

FPGA Field Programmable Gate Array. An integrated circuit composed of an array of configurable logic cells, each programmable to execute a simple function, surrounded by a periphery of I/O cells.

FSM Finite State Machine. A model of computation that captures an algorithm in states and rules for transitions between the states.

FSMD Finite State Machine with Datapath. Abstract model of computation describing the states and state transitions of an algorithm like a FSM and the computation within a state using a DFG.

GPU Graphics Processing Unit.

HCFSM Hierarchical Concurrent Finite State Machine. An extension of the FSM that explicitly expresses hierarchy and concurrency.

HDL Hardware Description Language. A language for describing and modeling blocks of hardware.

HW Hardware. The tangible part of a computer system that is physically implemented.

ISO International Organization for Standardization

ISS Instruction Set Simulator. Simulates execution of software on a processor at the ISA level.

KPN Kahn Process Network. A deterministic model of computation where processes are connected by unbounded FIFO communication channels to form a network

MoC Model of Computation. A meta model that defines syntax and semantic to formally describe any computation, usually for the purpose of analysis.

MPSoC Multi-Processor System-on-Chip. A highly integrated device implementing a complete computer system with multiple processors on a single chip.

OoO PDES Out-of-order Parallel Discrete Event Simulation.

OS Operating System. Software entity that manages and controls access to the hardware of a computer system. It usually provides scheduling, synchronization and communication primitives.

PCAM Pin-accurate and Cycle-Accurate Model. An abstract model that accurately captures all pins (wires) and is cycle timing accurate.

PDES Parallel Discrete Event Simulation.

PE Processing Element. A system component that provides computation capabilities, e.g. a custom hardware or generic processor.

PSM Program State Machine. A powerful model of computation that allows programming language constructs to be included in leaf nodes of a HCFSM.

RTL Register Transfer Level. Description of hardware at the level of digital data paths, the data transfer and its storage.

RTOS Real-Time Operating System. An operating system that responds to an external event within a predictable time.

SCE SoC Environment. A set of tools for the automated, computer-aided design of SoC and computer systems.

SHIM Software/hardware integration medium. A concurrent asynchronous deterministic model which is essentially an effective KPN with rendezvous communication for heterogeneous embedded systems.

SLDL System-Level Design Language. A language for describing a heterogeneous system consisting of hardware and software at a high level of abstraction.

SIR Syntax-Independent Representation. The intermediate representation of SpecC SLDL.

SoC System-On-Chip. A highly integrated device implementing a complete computer system on a single chip.

SW Software.

TLM Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.

Acknowledgements

The six years for Ph.D. study is such a wonderful and enjoyable journey which makes me a much better person in every possible way. I would like to take this opportunity to thank my professors, family, colleagues and friends who have encouraged and supported me in this process.

First and foremost, I would like to gratefully and sincerely thank my advisor, Prof. Rainer Doemer, for his guidance, understanding, patience and support during my doctorate study at UC Irvine. His technical insights, organization talents, amiable personality, and witty sense of humor have tremendously inspired and shaped me to be a good scholar and person. I enjoy our meetings and conversations which always bring me the morale, courage, and new ideas to identify and solve research and teaching problems. I also appreciate our discussions on philosophical topics in terms of research, teaching, work and living which I believe will benefit me for my whole life.

I would like to thank Prof. Daniel D. Gajski for serving on my committee and supporting me to pursue my career goal. His critical and visionary opinions and expertise has provided me a thorough and comprehensive view in the area of embedded computer systems. It is also so pleasant to have Prof. Gajski around in the same working area who cheers us up every day. In addition, I would also like to thank Prof. Brian Demsky for serving on my committee and for his valuable suggestions to improve this dissertation.

I would like to thank the members from the Laboratory for Embedded Computer Systems (LECS) in the Center for Embedded Computer Systems (CECS). In particular, I would like to thank Xu Han, Che-Wei Chang and Yasaman Samei for the group discussions and their contributions on the experimental models that have been used in this work. Many thanks to Tim Schmidt and Steffen Peter for the great research discussions, serving as the critical readers of this dissertation, and providing many constructive critiques. Also, special thanks to the CECS "Game Night" group, including Tim, Xu, Che-Wei, Steffen, Yasaman and I, which grants us the opportunity to know each other outside of the normal work setting and undoubtedly improves our nerdy social life. I am so fortunate to have their support and friendship in these years.

I would like to thank Dr. Christopher O'Neal from the Teaching, Learning and Technology Center (TLTC) at UC Irvine who has supervised me in the Pedagogical Fellowship (PF) program. His knowledge, passion and enthusiasm has opened my eyes towards the scholarship of teaching. I am grateful to his trust and support which encourages me to push myself out of my comfort zone to pursue excellence. I would also like to thank my fellow Pedagogical Fellows (PFFs) for their brilliant diverse mind sets which has greatly enriched my PF experience and their precious friendship.

This work has been supported in part by funding from the National Science Foundation (NSF) under research grant NSF Award #0747523 led by Prof. Doemer. I would like to thank the NSF for the valuable support.

Last but not least, I would like to express my deepest gratitude to my parents, Guoyao and Xiuhua, for their unconditional and endless love and faith in me, the very source of motivation to the pursuit of my dream.

CURRICULUM VITAE

Weiwei Chen

EDUCATION

Ph.D. in Electrical and Computer Engineering **2013**
University of California, Irvine

M.S. in Computer Engineering **2007**
Shanghai Jiao Tong University, China

B.Eng. in Computer Science and Engineering **2004**
Department of Computer Science and Engineering
Teaching Reform Class, Shanghai Jiao Tong University, China

International Exchange Student **2003**
School of Electrical and Computer Engineering
Purdue University, West Lafayette, Indiana

High School Graduation **2000**

EXPERIENCE

University of California, Irvine Irvine, CA
Graduate Student Researcher 2007 - 2013
Center for Embedded Computer Systems
Department of Electrical Engineering and Computer Science

Pedagogical Fellow 2012 - 2013
Teaching, Learning and Technology Center (TLTC)
The Henry Samueli School of Engineering

Teaching Assistant 2008 - 2013
Department of Electrical Engineering and Computer Science

Shanghai Jiao Tong University Shanghai, China
Graduate Research Assistant 2004 - 2007
School of Microelectronics

Teaching Assistant 2005 - 2006
School of Microelectronics

Microsoft Seattle, WA
Software Development Engineer Intern 2011

IBM China System and Technology Lab (CSTL) Shanghai, China
R&D Engineer Intern 2006 - 2007

PUBLICATIONS

Journal Articles (peer reviewed)

J3. **Weiwei Chen**, Xu Han, Che-Wei Chang, Rainer Doemer, "Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models", in preparation for submission to *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*

J2. **Weiwei Chen**, Xu Han, Che-Wei Chang, Rainer Doemer, "Advances in Parallel Discrete Event Simulation for Electronic System-Level Design", accepted for publication in *IEEE Design & Test of Computers*, vol.30, no.1, pp.45-54, January/February 2013

J1. **Weiwei Chen**, Xu Han, Rainer Doemer, "Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment", *IEEE Design & Test of Computers*, vol.28, no.3, pp.20-31, May-June 2011

Conference Papers (peer reviewed)

C12. Xu Han, **Weiwei Chen**, Rainer Doemer, "Designer-in-the-Loop Recoding of ESL Models using Static Parallel Access Conflict Analysis", in Proceedings of *the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, St. Goar, Germany, June 2013

C11. **Weiwei Chen**, Rainer Doemer, "Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions", in Proceedings of *the Design, Automation and Test in Europe Conference (DATE)*, pp.3-8, Grenoble, France, March 2013

C10. **Weiwei Chen**, Che-Wei Chang, Xu Han, Rainer Doemer, "Eliminating Race Conditions in System-Level Models by using Parallel Simulation Infrastructure", in Proceedings of *the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp.118-123, Huntington Beach, USA, November 2012

C9. **Weiwei Chen**, Xu Han, Rainer Doemer, "Out-of-order Parallel Simulation for ESL design", in Proceedings of *the Design, Automation and Test in Europe Conference (DATE)*, pp.141-146, Dresden, Germany, March 2012

C8. Rainer Doemer, **Weiwei Chen**, Xu Han, "Parallel Discrete Event Simulation of Transaction Level Models", invited paper, in Proceedings of *the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp.227-231, Sydney, Australia, January 2012

C7. **Weiwei Chen**, Rainer Doemer, "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 (ASP-DAC)*, pp.461-466, Sydney, Australia, January 2012

C6. Rainer Doemer, **Weiwei Chen**, Xu Han, Andreas Gerstlauer, "Multi-Core Parallel Simulation of System-Level Description Languages", invited paper, in Proceedings of

the 16th Asia and South Pacific Design Automation Conference (ASP-DAC), pp.311-316, Yokohama, Japan, January 2011

C5. **Weiwei Chen**, Xu Han, Rainer Doemer, "ESL Design and Multi-Core Validation using the System-on-Chip Environment", in Proceedings of *the 15th IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pp.142-147, Anaheim, USA, June 2010

C4. **Weiwei Chen**, Rainer Doemer, "A Fast Heuristic Scheduling Algorithm for Periodic ConcurrenC Models", in Proceedings of *the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp.161-166, Taipei, Taiwan, January 2010

C3. Rongrong Zhong, Yongxin Zhu, **Weiwei Chen**, Mingliang Lin, Weng Fai Wong, "An Inter-core Communication Enabled Multi-core Simulator Based on SimpleScalar", in Proceedings of *the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW)*, pp.758-763, Niagara Falls, Canada, April 2007

C2. Guoyong Shi, **Weiwei Chen**, C.-J. Richard Shi, "A Graph Reduction Approach to Symbolic Circuit Analysis", in Proceedings of *the 12th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp.197-202, Yokohama, Japan, January 2007

C1. **Weiwei Chen**, Guoyong Shi, "Implementation of a Symbolic Circuit Simulator for Topological Network Analysis", in Proceedings of *the IEEE Asia Pacific Conference on Circuit and System (APCCAS)*, pp.1368-1372, Singapore, December 2006

Book Chapters

B2. **Weiwei Chen**, Rainer Doemer, "ConcurrenC: A New Approach towards Effective Abstraction of C-based SLDLs", *Analysis, Architectures and Modeling of Embedded Systems (ed. A. Rettberg, M. Zanella, M. Amann, M. Keckeisen, F. Rammig)*, Springer, 2009, ISBN 978-3-642-04283-6

B1. **Weiwei Chen**, Guoyong Shi, "Symbolic Analysis of Analog Integrated Circuits", *Embedded Systems and Materials Research for Advanced Applications, the 1st Chinese-German Summer School in Shanghai*, September, 2006, ISBN-10: 3-00-019576-9 / ISBN-13: 978-3-00-019576-1

Technical Reports

TR5. Xu Han, **Weiwei Chen**, Rainer Doemer, "A Parallel Transaction-Level Model of H.264 Video Decoder", *TR-11-03*, Center for Embedded Computer Systems, UC Irvine, June 2011

TR4. **Weiwei Chen**, Rainer Doemer, "A Distributed Parallel Simulator for Transaction Level Models with Relaxed Timing", *TR-11-02*, Center for Embedded Computer Systems, UC Irvine, May 2011

TR3. **Weiwei Chen**, Rainer Doemerr, "ConcurrenC: A Novel Model of Computation for Effective Abstraction of C-based SLDLs", *TR-09-07*, Center for Embedded Computer

System, UC Irvine, May 2009

TR2. **Weiwei Chen**, Siwen Sun, Bin Zhang, Rainer Doemer, "System Level Modeling of a H.264 Decoder", *TR-08-10*, Center for Embedded Computer System, UC Irvine, August 2008

TR1. **Weiwei Chen**, Rainer Doemer, "System Specification of a DES Cipher Chip", *TR-08-01*, Center for Embedded Computer System, UC Irvine, January 2008

Poster Presentations

P3. **Weiwei Chen**, Rainer Doemer, "Out-of-order Parallel Simulation for Electronic System-Level Design", in the *EDAA/ACM SIGDA PhD Forum at the Design, Automation and Test in Europe Conference (DATE)*, Grenoble, France, March 2013

P2. **Weiwei Chen**, Rainer Doemer, "Out-of-order Parallel Discrete Event Simulation for ESL Design", *Graduate Student Poster Presentation*, Faculty Retreat, Department of Electrical Engineering and Computer Science, UC Irvine, September 2012

P1. **Weiwei Chen**, Rainer Doemer, "Parallel Discrete Event Simulation for ESL Design", in *SIGDA Ph.D. Forum at the Design Automation Conference (DAC)*, San Francisco, USA, June 2012

HONORS AND AWARDS

- Pedagogical Fellowship, UC Irvine, 2012-13 academic year
- Henry Samueli Endowed Fellowship, The Henry Samueli School of Engineering, UC Irvine, 2007
- Travel grants, Design Automation Conference (DAC) 2012 and Asia and South Pacific Design Automation Conference (ASP-DAC) 2010
- Young Student Support Award, Design Automation Conference (DAC), Anaheim, CA, 2010
- Excellent Teaching Assistant Award, School of Microelectronics, SJTU, 2006
- National Scholarship for Academic Excellence, China, 2006
- *Morgan Stanley* Scholarship, *Infineon* Scholarship, SJTU, 2006
- *Guanghua* Scholarship, SJTU, 2005
- First-round and second-round Exceptional Undergraduate Student, SJTU, 2001, 2003
- People's Scholarship for Academic Excellence, SJTU, 2000 - 2003
- *Pan Wen-yuan Foundation* Scholarship, China, 2001
- *Shuping* Scholarship, 7 years since the 9th grade

ABSTRACT OF THE DISSERTATION

Out-of-order Parallel Discrete Event Simulation for Electronic System-Level Design

By

Weiwei Chen

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2013

Professor Rainer Doemer, Chair

The large size and complexity of the modern embedded systems pose great challenges to design and validation. At the so called electronic system level (ESL), designers start with a specification model of the system and follow a systematic top-down design approach to refine the model to lower abstraction levels step-by-step by adding implementation details. ESL models are usually written in C-based System-level Description Languages (SLDLs), and contain the essential features, such as clear structure and hierarchy, separate computation and communication, and explicit parallelism. The validation of ESL models typically relies on simulation. Fast yet accurate simulation is highly desirable for efficient and effective system design.

In this dissertation, we present *out-of-order parallel discrete event simulation (OoO PDES)*, a novel approach for efficient validation of system-level designs by exploiting the parallel capabilities of todays multi-core PCs for system level description languages. OoO PDES breaks the global simulation-cycle barrier of traditional DE simulation by localizing the simulation time into each thread, carefully delivering notified events, and handling a dynamic management of simulation sets. Potential conflicts caused by parallel accesses to shared variables and out-of-order thread scheduling are prevented by an advanced predictive static model analyzer in the

compiler. As such, OoO PDES allows the simulator to effectively exploit the parallel processing capability of the multi-core system to achieve fast speed simulation without loss of simulation and timing accuracy.

We perform simulation experiments on both highly parallel benchmark examples and real-world embedded applications, including a JPEG image encoder, an edge detector, a MP3 audio decoder, a H.264 video decoder, and a H.264 video encoder. Experimental results show that our approach can achieve significant simulation speedup on multi-core simulation hosts with negligible compilation cost.

Based on our parallel simulation infrastructure, we then propose a tool flow for dynamic race condition detection to increase the observability for parallel ESL model development. This helps the designer to quickly narrow down the debugging targets in faulty ESL models with parallelism. This approach helps to reveal a number of risky race conditions in our in- house embedded multi-media application models and enabled us to safely eliminate these hazards. Our experimental results also show very little overhead for race condition diagnosis during compilation and simulation.

Overall, our work provides an advanced parallel simulation infrastructure for efficient and effective system-level model validation and development. It helps embedded system designers to build better products in shorter time.

1 Introduction

Embedded computer systems are special purpose information systems that are embedded in larger systems to provide operations without human intervention [1]. They are pervasive and ubiquitous in our modern society with a wide application domain, including automotive and avionic systems, electronic medical equipments, telecommunication devices, industrial automation, energy efficient smart facilities, mobile and consumer electronics, and others. Embedded computer system is one of the most popular computational systems in our current information era [1]. In 2002, more than 98 percent of all the microprocessors produced worldwide are used for embedded computer systems [2]. The global market for embedded technologies is also tremendously big and increasing rapidly. Ebert et al. stated in [3] that *"the worldwide market for embedded systems is around 160 billion euros, with an annual growth of 9 percent"*. We can expect that more and more products we use in our daily life will be based on embedded computer systems in the future.

Following the "Dortmund" definition of embedded computer systems in [1], both hardware and software are critical for embedded system design.

The hardware for embedded systems is usually less standard than general purpose computer systems [1]. Embedded computer systems often consist of a large set of input devices, heterogeneous processors, and customized memory components. A huge range of input devices are used in embedded systems to collect the information from the user and the external environment. To name a few, *sensor* devices detect the physical qualities of the external environment, such as

temperature, humidity, velocity, acceleration, brightness, etc; *analog-to-digital converters* discretize continuous physical quantities into values in the discrete domain; and *sample-and-hold circuits* convert continuous time into discrete signals [1]. The input information is then sent to the processing units in the system for computation and decision making to generate system intelligence and automation.

Modern embedded system platforms usually compose various types of processing elements into the system, including general-purpose CPUs, application-specific instruction-set processors (ASIPs), digital signal processors (DSPs), dedicated hardware accelerators implemented as application-specific integrated circuits (ASICs), and intellectual property (IP) components. Moreover, embedded systems usually use customized memory hierarchies due to its tight performance, power, and size constraints. Since embedded systems are usually designed to a specific target application, the memory system is allowed to be tailored accordingly by using customized memory components and architectures, such as caches with different configurations and management strategies, scratch-pad memory, streaming buffers, DRAM, and multiple SRAMs [4]. As a whole system, the hardware components are connected in a network and communicate with each other via different communication protocols, such as time triggered protocol (TTP/C) [5] for safety-critical systems, controller area network (CAN) for automation systems [6], ARINC 629 for civil aircraft databuses [7], FlexRay for automotive communication network [8], Advanced Micro-controller Bus Architecture (AMBA) for system-on-chip design [9], and Time-Triggered Ethernet (TTEthernet) for predictable and deterministic real-time communication [10]. With the advance in semiconductor technology, many embedded systems can now be integrated entirely onto a single die which result in complex System-on-Chip (SoC) architectures [11, 12].

The software of modern embedded computer systems is often tightly coupled with the underlying heterogeneous hardware platform and external physical processes [13]. Timeliness, concurrency, and constraints on the available resources are several most important features for embedded

software. In order to interact with the physical world, embedded software usually needs to process and response in real-time. Correctness is not only depends on the functionality but also on the time frame. With multiple components integrated in one system and the external physical processes which are fundamentally running in parallel, embedded software also needs to handle concurrency of the whole system, such as monitoring multiple sensors to get inputs and processing information on different computational components at the same time. Furthermore, embedded software also need to take resource constraints into consideration in order to meet the product requirement. Resource constraints include power consumption, memory size, thermal sensitivities, available computing capabilities, and so on.

The general embedded software stack usually consists of multiple components including application software, middleware or adapter layer, operating system or real-time operating system (RTOS), device drivers, boot firmware, communication protocol stacks, and hardware abstraction layer [11]. Compared to the traditionally constraint embedded applications, large application software, which is written in complex programming languages such as C/C++ and Java, is commonly used in today's embedded systems. The programming flexibility introduced by the high-level programming languages results in higher software productivity as well as product innovation. Embedded operating systems enable and manage the concurrency, process scheduling, and resource sharing in the whole system. A real-time operating system (RTOS) , which is widely used in nowadays' embedded systems, facilitates the construction of the systems under certain real-time constraints in line with the timeliness requirement of the design. The rest of the software components are often categorized as Hardware-depend Software (HdS) due to the fact that they interact closely with the underlying hardware platform to provide configurability and flexibility. As was defined in [11] by Ecker et al., HdS is specifically built for a particular hardware block, implements a system's functionality together with the hardware, and provides the application software with an interface to easily access the hardware features.

1.1 System-level Design

The large size, complexity and heterogeneity of today's embedded systems imposes both theoretical and engineering challenges on system modeling, validation, debugging, synthesis, and design space exploration, in order to build up a satisfying system within a short time-to-market design period.

Fig. 1.1 shows the design challenge statistically . Moore's law predicts that the capability of hardware technology doubles every eighteen months, whereas the hardware design productivity in the past few years is estimated to increase at 1.6x over the same period of time. On the other hand, since the hardware productivity improved over he last several years by putting multiple cores on to a single chip, the additional software required for hardware (HdS) doubles over every ten months. However, the productivity especially for HdS is far behind which is doubling only every five years. There are growing gaps between the technology capabilities and the real hardware and software productivities over the years. This tells that System-on-Chip design productivity cannot follow the pace of the nano-electronics technology development which is characterized by Moore's Law. Hence, design methodologies become a popular research topic to tackle those design challenges of embedded systems in the recent decade.

System-level Design is a promising solution to improve the design productivity [14]. The 2004 edition of the International Technology Roadmap for Semiconductors (ITRS) places system-level as "*a level above RTL including both hardware and software design*" [15]. Here, *"Hardware" (HW)* consists processing elements, buses, hardwired circuits and reconfigurable cells. It implements the circuit element in the system. *"Software" (SW)* includes embedded codes in programming languages, configuration data, etc. It defines the functionality that is performed on the underlying hardware. ITRS also states that there are two independent degrees of design freedom in system-level, *behavior* which defines the system functionality and *architecture* which defines the system platform [16].

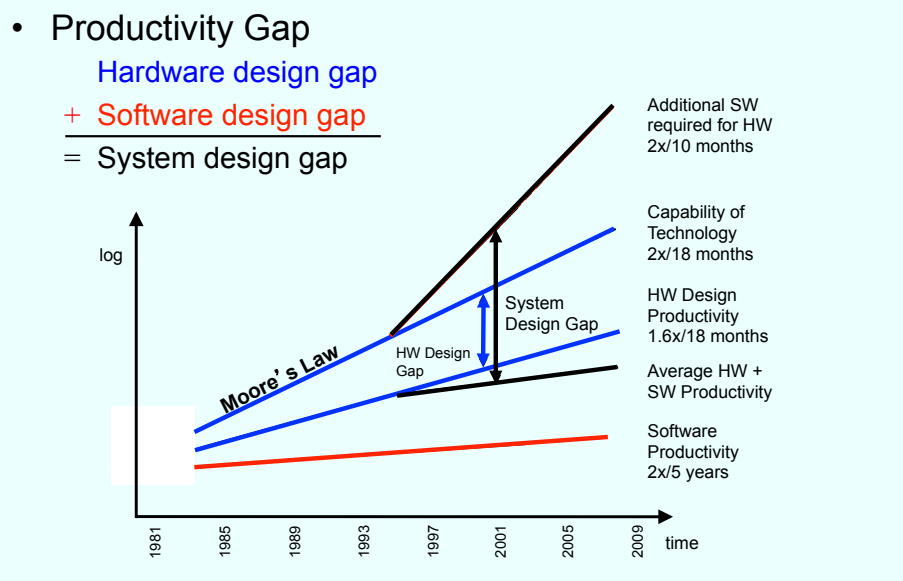


Figure 1.1: Hardware and software design gaps versus time (source [11])

System-level design is a holistic approach to address the design challenges that are brought by the radical increase in the complexity of both embedded hardware and software. At the system-level, the designer treats the intended system as a whole entity and focuses on the functional specification or algorithm which is independent from the hardware platform or software implementations. With a complete picture of the entire system, the hardware and software can be designed jointly (co-design) at the same time. It also makes global system optimization possible and more efficient. Moreover, system-level design can often rely on guided automatic approaches to add more details to get the final implementation. For instance, the designer can map a functional module in the system-level model to a processor and run it as regular software program or synthesize the module into a specific hardware unit by using high-level synthesis tools. With the help of designer guided refinement tools, system-level design can therefore increase productivity dramatically.

1.1.1 Levels of Abstraction

Abstraction is a natural process in system-level design which allows the designers to handle the complexity of the entire system without the distraction of the low-level implementation details. According to Doemer in [17], "*a well-known solution for dealing with complexity is to exploit hierarchy and to move to higher levels of abstraction*".

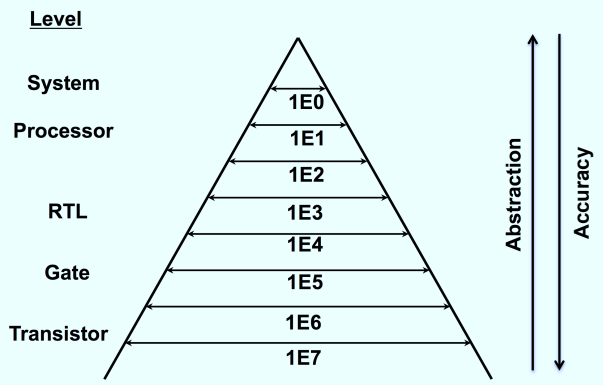


Figure 1.2: Levels of abstraction in SoC design (source [18])

As shown in Fig. 1.2, raising the level of abstraction results in fewer numbers of components to handle. In particular, an implemented system may contain tens of millions of transistors. Gate-level abstracts the transistor-level by using logic gates and flip-flops which are composed of specific sets of transistors. Then, register-transfer level (RTL) typically reduces the number of components to several thousands. This may further be represented by a couple of processor elements which construct one system. The reducing number of components in higher abstraction levels helps to maintain a system-level overview in a simple way. Ideally, the designer can start at the pure functional level and gradually refine the design to lower abstraction levels with more implementation details with the help of system-level design tools.

The idea of abstraction levels is realized by using *models* in the design process. ***Models*** are the abstraction of the reality. They are built for different abstraction levels to reflect the features that are critical to the corresponding level.

The advantage of having different levels of abstraction allows the designer to only focus on the critical features of a certain abstraction level and ignore the unneeded lower-level details. However, this cannot be achieved without cost. Hiding more implementation details can in turn cause the loss of *accuracy*, such as timing, number of pins and interrupts. It is a trade-off between more focused view of design and accuracy for timing and physical design metrics.

1.1.2 The Y-Chart

Gajski's *Y-Chart* [19] is a conceptual framework to coordinate the abstraction levels in three design domains (the axes), i.e. *behavior (functionality)*, *structure (architecture)*, and *physical layout (implementation)*. The **behavioral** domain defines the functionality of the system, i.e. the output in terms of the input over time. The **structural** domain describes the architecture of the hardware platform including processing elements (PEs) and communication elements (CEs). The **physical** domain specifies the implementation details of the structure, such as the size and position of the components, the printed circuit board, and the port connection between components.

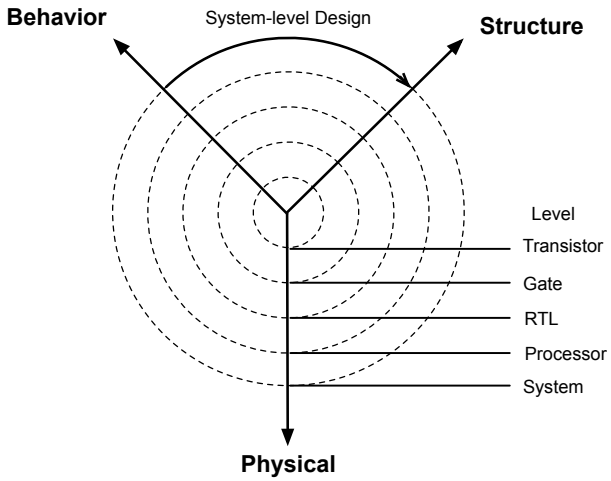


Figure 1.3: System-level design in the Y-Chart (source [17])

The level of abstractions are identified by the dashed concentric circles, increasing from tran-

sistor level to system-level in the *Y-Chart*. They are orthogonal to the design domains which are instead represented as axes.

Design flows are illustrated as paths in the *Y-Chart*. For instance, the behavior of the design can first be mapped to the structure description at the system-level, and then be refined to lower abstraction levels along the *structure* axis to a final implementation. Another design flow may start from the system-level function description in programming, refine the design to behavioral processor-level first with controllers and datapaths, and follow the structural synthesis flow down to physical layouts.

The heart of system-level design is the platform mapping from the behavior onto the structure [20]. This is illustrated as the arrow on the outmost circle in Fig. 1.3.

1.1.3 System-level Design Methodologies

Computer Aided Design (CAD) tools are introduced to automate the design process which is too complex for the designers to do manually due to the advances in technologies. CAD tools usually follow sets of stringent rules and use dedicated component libraries to make the design process more efficient and manageable. The aggregation of models, components, guidelines and tools is called design *methodology* [13]. Design methodologies evolve with technologies, applications, as well as design group foci.

The **bottom-up** design methodology starts from the lowest abstraction level, and generates libraries for the next higher level by each level (Fig. 1.4a). For example, the *transistor* level generates the library for the logic gates and flip-flops in the *logic* level with N-type and P-type transistors. Floorplan and layout are needed for all the levels by using this approach. The advantage of this methodology is that each abstraction level has its own library with accurate metric estimation. The disadvantage is that optimal library for a specific design is difficult to achieve since parameters need to be tuned for all the library components at each level.

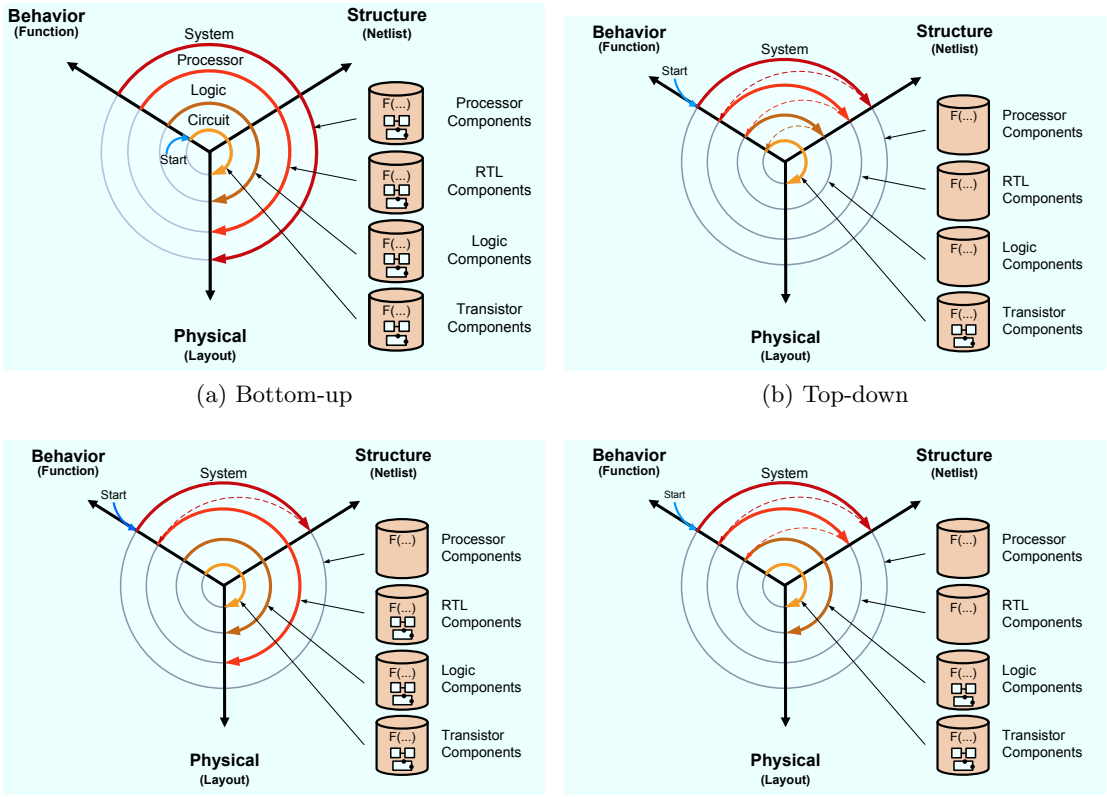
The **top-down** design methodology starts with the highest abstraction level to convert the functional description of the system into a component netlist at each abstraction level (Fig. 1.4b). Component functions are decomposed with further details step by step down to the lower abstraction levels. System layout is only needed at the lowest transistor level. The advantage of this approach is that high-level customization is relatively easy without implementation details, and only a specific set of transistors and one layout is needed for the whole process. The disadvantage is that it is difficult to get the accurate performance metrics at the high abstraction levels without the layout information.

The **meet-in-the-middle** design methodology is a combination of the *bottom-up* and *top-down* approaches. In this flow, the three design aspects, i.e. behavior, structure, and physical, meet at different abstraction levels. For example, as shown in Fig. 1.4c, the functional specification is synthesized down into processor components; whereas RTL components from the library construct the processor components up with a system layout. The RTL components from the library are generated through a bottom-up process with their own structure and layout. Therefore, the three design domains meet at the *processor* level. Physical design and layout are required for three times in this flow: first for the standard cells, second for RTL components, and last for the system.

Fig. 1.4d shows an alternative flow where the three domains meet at the *RTL* level. In this flow, the system behavior description is mapped onto a certain architecture with PEs and CEs; then the behavior models of the PEs and CEs are synthesized into RTL components; whereas the RTL components are synthesized with logic components. The system layout can be generated by combining the logic components through floor planing and routing, since logic components has already have their own layout with standard cells. This flow requires physical design and layout for two times, one for the standard cells, and the other for the system.

The advantage for the *meet-in-the-middle* design methodology is that it can provide more accurate metrics at high abstraction levels than the *top-down* approach; and requires less layout

and libraries than the *bottom-up* one. The disadvantage is that while the metrics accuracy is increasing, the easiness for optimization decreases; and more than one layout and libraries are still required.



(a) Bottom-up

(b) Top-down

(c) Meet-in-the-middle (meet at the Processor level)

(d) Meet-in-the-middle (meet at the RTL level)

Figure 1.4: System-level design methodologies (source [13])

Other design methodologies, such as *platform methodology*, *system methodology*, and *FPGA methodology* that have been discussed in [13] are more product oriented. These methodologies are commonly used in companies and design groups for building products with their own platform components.

1.1.4 Electronic System-Level Design Process

In this dissertation, we will focus on the so called *Electronic System-Level (ESL) Design* which aims at a systematic *top-down* design methodology to successively transform a given high-level system description into a detailed implementation. There are three backbone aspects for ESL design: *specification, validation, and refinement.*

The starting point for ESL design is the ***specification*** of the system. The specification is a pure functional model in which the algorithm of the intended design is captured and encapsulated in computation modules, i.e. *behaviors*, and the communication between the behaviors is described through abstract communication modules, i.e. *channels*. The specification model should be *complete* with all features that are essential to system-level design, but also *abstract* without unnecessary implementation details such as timing and interrupts. It also needs to be *executable* so that the designer can verify the model with the design expectations. Typically, specification models are written in programming languages and can be compiled for execution. As the initial formal description of the system, the specification model serves as the golden reference to the other models that will be refined at lower abstraction levels.

Once having the system specification, ***validation*** is needed to check its correctness. Typically, ESL design models are validated through *simulation*. *Simulation* is a dynamic process to validate the functionality and the metrics of the model in terms of the execution output for given input vectors. The speed of simulation on the simulation host is often slower than the execution speed of the real implemented system. Moreover, since simulation is driven by input vectors, unless exhaustive simulation with all the possible input vectors are performed, simulation based validation cannot cover all the cases.

Validation can also be performed through static model analysis, which is often called *verification*. Verification usually refers to the approach of using formal methods to prove certain characteristics of the model. In contrast to *simulation*, *verification* can yield 100% coverage on

all the possible cases. However, the complexity of verification can grow exponential while the size of the model grows. Thus, it is only possible to verify small designs.

Refinement is the process of adding more implementation details into the model, including architecture mapping, scheduling strategies, network connections, communication protocols, and so on. In ESL design, *refinement* is often guided by the designers' decisions and performed automatically with the help of CAD tools.

A sequence of ELS model refinement at different abstraction levels is described in the following paragraphs.

After building and validating the *specification* model, the architecture of the system will be determined with respect to the number and types of processing elements, memory components, and their connectivities. PEs include programmable and nonprogrammable component. Programmable PEs are generic embedded processors like ARM [21], MIPS [22], LEON [23], ColdFire [24], Blackfin [25], and MicroBlaze [26] processors, as well as digital signal processors (DSPs). Non-programmable elements are customized hardware accelerators or non-instruction set computer components [27]. In the *architecture mapping* step, the designer first decide the HW/SW partition among the functional modules. Performance-critical modules are usually implemented as dedicated HW units and the rest of them are compiled and executed as SW on programmable PEs. It is a trade-off between high-speed high-cost HW implementation and low-cost less efficient SW execution. Also, the parameters for the PEs, including clock frequency, memory size, width of the address bus, are adjusted in this step for specific applications.

The second refinement step is to decide the *scheduling strategies* on programmable processing elements. Multiple behaviors can be mapped onto a single host PE and run as software. Proper scheduling mechanism is needed to decide the execution order of the tasks on the PEs since the computational resources available can be fewer than the number of mapped behaviors. Scheduling can be static or dynamic. In case of static scheduling, the constraints such as behavior workload and dependencies should be known by analyzing the model beforehand and

will not change during execution time. In case of dynamic scheduling, operating system is often used to map behaviors onto tasks and manage them under priority-based or round-robin style scheduling schemes.

The third refinement step is *network* allocation. This step determines the connectivities among the PEs in the system. For example, there is a generic processor and several HW accelerators in the system. They communicate with each other via the CPU bus of the processor. Thus, the processor will be specified as the master of the bus while the HW accelerators as the slaves. Port interfaces also need to be allocated for each PEs so as to properly connect to the bus.

The next refinement is for *communication*. The communication of the system is described and encapsulated in the *channel* modules in the *specification* model, and is mapped to virtual buses in the *architecture* model. In this step, the virtual buses are replaced with actual ones for specific protocols. The communication protocols are then implemented by inlining the communication code into the PEs, i.e. bus drivers in the operating system. If two buses with different protocols need to exchange data, transducers need to be inserted to bridge the gap between them.

The system refinement process is completed with the *backend* tasks. The goal of the *backend* is to create a final optimized *implementation* for each component in the design. The customized hardware units and transducers need to be synthesized into RTL netlists written in *VHDL* or *Verilog*. The software code needs to be generated, either in C or assembly, and compiled for the corresponding processors. The *implementation* model is at the lowest abstraction level and contains all the information, including functionality, structure, communication and timing with respect to the actual implementation. It is bus-cycle accurate for the communication and clock-cycle accurate for the computation on the processors.

Moreover, in ESL design, *estimation* techniques are often used to measure the metrics of the system models, such as timing, power consumption, workload, etc. Similar to *validation*, *estimation* can be performed statically by model analysis, or dynamically by simulation and profiling. There is also a trade-off between the estimation accuracy and speed. Estimation is faster but

less accurate on the models at higher abstraction levels than on the ones at lower abstraction levels. It should be emphasized that *validation* and *estimation* needs to be performed at the end of each aforementioned refinement step to ensure that the refined model still meets the design requirements.

1.2 Validation and Simulation

In system-level design, abstract models are built for better understanding on the system, and refinement and optimization for the final implementation. *Simulation* is critical in the design process for both *validation* and *estimation* which check the models according to the intended design constraints.

1.2.1 Language Support for System-level Design

System-level models need to be executable for efficient *validation* and *estimation*. They are typically described in programming languages. Programming languages are designed for the convenience of making the computer systems behave according to human beings' thoughts. Various languages are proposed to improve the work of system design. We will review some of those design languages for embedded system in this section.

Hardware Description Languages (HDLs)

The goal of a hardware description language is to concisely specify the connected logic gates. It is originally targeted at simulation and a mixture of structural and procedural modeling.

VHDL [28, 29] and ***Verilog*** [30, 31] are the most popular languages for hardware modeling and description. Both of them have discrete event semantics and ignore the idle portions of the system for efficient simulation. They describe the hardware system in hierarchy by partitioning the

system into functional blocks which could be primitives, other blocks, or concurrent processes. HDLs often describe the system as finite-state machines (FSMs), datapath and combinatorial logic. The described system could be simulated to verify its correctness and be synthesized into real hardware circuits.

HDLs support the lower level description of an embedded system. However, they could not ignore those implementation details like pins, wires, clock cycles. Thus, it is too complicated to describe a whole system and the simulation will take a very long time.

Software Languages

Software languages describe sequences of instructions for a processor to execute. Most of them use sequences of imperative instructions which communicate through the *memory*, an array of numbers that hold their values until changed.

Machine instructions are typically limited to do simple work, say, adding two numbers. Therefore, high-level software languages aim to specify many instructions concisely and intuitively. The ***C programming language*** [32] provides expressions, control-flow constructs, such as conditionals, loops and recursive functions. The ***C++ programming language*** [33] adds classes to build new data types, templates for polymorphic code, exceptions for error handling, and a standard library for common data structures. The ***Java programming language*** [34] provides automatic garbage collection, threads, and monitors for synchronization. It is quite suitable for writing applications on different embedded system platforms [35].

However, software language does not have the ability to explicitly describe hardware behaviors like signals, events, and timing.

System-level Description Languages (SLDLs)

System-level description languages (SLDLs), such as ***SpecC*** [36, 18, 37], ***SystemC*** [38, 39, 40] and ***SystemVerilog*** [41], are available for modeling and describing an embedded system at different abstraction levels.

SpecC is a system-level description language as well as a system level design methodology. It is based on the C language and a true superset of ANSI-C [42]. In addition to the constructs in ANSI-C, SpecC has the a minimal set of extensions to support the requirements of system-level modeling, such as structural and behavioral hierarchy, concurrency, communication, synchronization, timing, exception handling, and explicit state transitions.

SpecC uses the same discrete-event simulation semantics as *Verilog* and *VHDL* [43]. It is executable and synthesizable [18].

SystemC is a C++ library which facilitates both system and RTL modeling. It builds systems from *Verilog*- and *VHDL*-like modules, each of which has a collection of I/O ports, instances of other modules or processes written in C++. SystemC is the de-facto language for system-level design in industry.

SystemC and SpecC are conceptually equivalent. However, the two languages have some distinctions in semantics and implementation details. We will discuss the SpecC and SystemC SLDLs in more detail in Section 1.5.

SystemVerilog is the industry's first unified hardware description and verification language (HDVL) standard. It is a major extension of the *Verilog* language. *SystemVerilog* supports modeling and verification at the "transaction" level of abstraction by the support of assertion-based verification (ABV). It also provides a set of extensions to address advanced design requirements, like modeling interfaces, removing os restrictions on module port connections, allowing any data type on each side of the port, etc. Object-oriented techniques are new features for modeling

hardware and testbenches.

1.2.2 System Simulation Approaches

Typically, system-level design validation is based on simulation. The functional behavior of the model is defined by the software semantics of the underlying C/C++ language, and the synchronization and communication is defined by the execution constructs for the system-level or hardware description language.

Simulation is an approach to perform experiments using the computer implementation of a model. There are three types of computer-based simulation in classical thinking: *discrete event*, *continuous*, and *MonteCarlo*. Nance articulated the definitions of these simulations in [44] as follows:

> *"**Discrete event simulation** utilizes a mathematical/logical model of a physical system that portrays state changes at precise points in simulated time. Both the nature of the state change and the time at which the change occurs mandate precise description.*
>
> ***Continuous simulation** uses equational models, often of physical systems, which do not portray precise time and state relationships that result in discontinuities [...]. Examples of such systems are found in ecological modeling, ballistic reentry, or large scale economic models.*
>
> ***Monte Carlo simulation** [...] utilizes models of uncertainty where representation of time is unnecessary [...]. Typical of Monte Carlo simulation is the approximation of a definite integral by circumscribing the region with a known geometric shape, then generating random points to estimate the area of the region through the proportion of points falling within the region boundaries."*

In short, the notion of time is emphasized and the simulation states evolve only at discrete time

points in *discrete event (DE)* simulation. DE simulation can be *deterministic* with no degree of randomness, or *stochastic* with some random variables. *Continuous* simulation changes the value of the variables continuously with respect to time. The system state is usually tracked based on differential equations. *Monte Carlo* simulation approximates the probability of outcomes for a system with some degree of randomness. It often requires repetitive trials, and the representation of time is not important.

1.2.3 Discrete Event Simulation

Digital systems are naturally discrete. The behavior of a system is usually described in state transitions at discrete time points, i.e. clockcycles. Therefore, both system-level description languages (SLDLs) and hardware description languages (HDLs) use *discrete event (DE) semantics* for simulating the communication and synchronization in the system. A simulation is *deterministic* if multiple simulation runs using the same input value always yield the same results.

Discrete event simulation generally operates on a sequence of events which happen at discrete instants in time. The state of the system is changed when an event occurs at a particular time. No state change is assumed to happen between two consecutive events so that the simulation can be performed in a discrete way. In SLDL simulation, the simulation state is updated when two types of *"events"* occur:

- A synchronization event is notified to trigger the behavior (module) that is waiting on it;
- The simulation time is advanced.

The simulation cycle reflects the timing information of the model. It is represented by a two-tuple of ***(time, delta)*** which is global to all the behaviors (modules). The ***time-cycle*** represents the exact timing such as delay or execution time in the system model. The ***delta-cycle*** is a special notion in SLDLs and HDLs to interpret the *zero-delay* semantic in digital systems.

As is defined in [38], *delta-cycle* lasts for an infinitesimal amount of time and is used to impose a partial order of simultaneous actions. *Delta-cycles* do not advance the actual time, but reflect the order of event deliveries and signal updates within a particular *time-cycle*. There can be multiple *delta-cycles* in one *time-cycle*.

It should be emphasized that the discrete event semantics define the execution order of the behaviors (modules) in SLDL models. Conceptually, the behaviors within the same simulation cycle, i.e. *(time, delta)* are allowed to simulate concurrently. However, the thread execution order in the same *delta-cycle* can be non-deterministic.

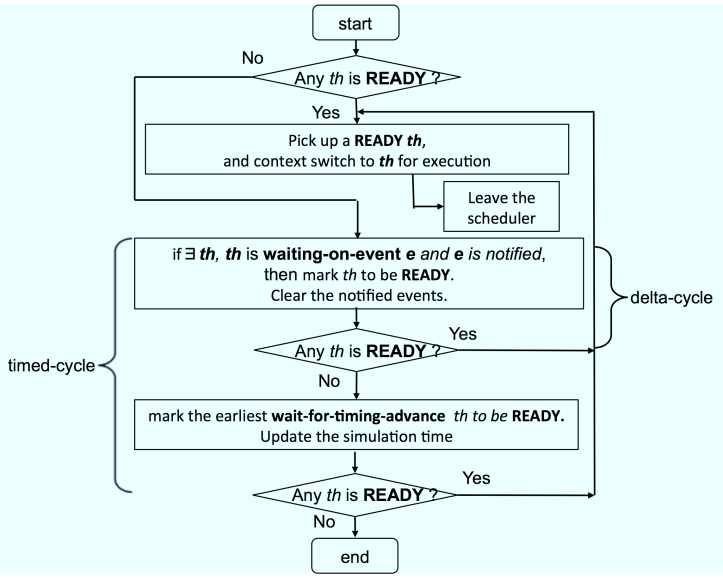


Figure 1.5: Scheduler for discrete event simulation

During simulation, the behaviors (modules) in the models are mapped to *working threads* (processes) with respect to their behavioral structure (i.e. sequential execution, parallel execution, pipeline execution, or finite-state machine execution, etc.) and start to execute the code. The threads suspend themselves when they need to wait for an event to happen or the simulation time advances to a specific point. A thread can also suspend itself after forking some children threads for execution and wait for them to join. When all the threads in simulation are suspended, the DE scheduler will start to work.

The ***scheduler*** is the heart of a discrete event simulation. Fig. 1.5 shows the basic scheduling flow for the SLDL discrete event simulation. The steps are summarized as follows:

1. The scheduler checks if there is any thread *th* that is ready to resume the execution. If so, proceed to *step 2*; otherwise, goto *step 3*.

2. The scheduler picks up a thread *th* that is ready to resume its execution, and issues the thread *th* to run. Scheduling stops.

3. If there is a thread *th* that is waiting on an event *e* and the event *e* is notified, mark the thread *th* as *READY* to resume its execution. Check all the *wait-on-an-event* threads and mark them to be *READY* if the waited events are notified. Then, clear all the notified events and increment the *delta-cycle*. Proceed to *step 4*.

4. If there is any thread *th* that is ready to resume its execution, goto *step 2*; otherwise, proceed to *step 5*.

5. The scheduler picks up the *wait-for-timing-advance* threads with the earliest time advance *t*, mark them as *READY* to resume execution, and updates the simulation time to *t* (update the *time-cycle*). Proceed to *step 6*.

6. If there is any thread *th* that is ready to resume its execution, goto *step 2*; otherwise, simulation finishes. *Deadlock* exists in the model if there are still some suspended threads.

Note that there are two loops in the scheduling process. The inner loop which mainly covers *step 3* and *step 4* manages the *delta-cycle* advances. The outer one which includes the inner loop, *step 5* and *step 6* further takes care of the *time-cycle* advances. The two loops are implemented in this way since there can be multiple *delta-cycles* within a particular *time-cycle*.

The ***scheduler*** can be implemented as a dedicated thread that runs infinitely until the simulation stops, i.e. the function *exit(0)* is called in the working thread. The working threads and the scheduler can work alternatively during simulation through thread synchronization.

The ***scheduler*** can also be implemented as a function and called by the working threads when scheduling is needed. This can reduce the overhead of context switching that is unavoidable by using a dedicated thread (Section 3.5).

1.3 Dissertation Goals

The large size and complexity of modern embedded systems with heterogeneous components, complex interconnects, and sophisticated functionality pose enormous challenges to system validation and debugging. While system-level design methodology provides a systematic flow for modeling and refinement, fast yet accurate simulation is a key to enabling effective and efficient model validation, performance estimation, and design space exploration.

The multi-core technology allows multiple threads in one program to run concurrently. It holds the promise to map the explicit parallelism in system-level models onto parallel cores in the multi-core simulation host so as to reduce the simulator run time.

However, the reference DE simulation kernels for both SpecC and SystemC are using cooperative multi-threading library which is not scalable on multi-core simulation platforms. Moreover, the DE execution semantics tend to run each module in a certain order so as to properly update the shared global simulation time. The logic parallelism expressed in the model cannot exploit the concurrency during simulation. In other words, discrete event simulation for system-level description languages is slow due to its kernel implementation and the sequential execution semantics.

This dissertation aims at leveraging multi-core parallel technology for fast discrete event simulation of system-level description languages. The following aspects are the goals for the work in this dissertation:

- define a model of computation which can capture system-level features, such as parallel

execution semantics, and can be expressed precisely in both SystemC and SpecC SLDLs.

- propose and design a SLDL simulation framework that:
  - can utilize multi-core computational resource effectively
  - is safe for multi-thread synchronization and communication
  - is transparent for the designer to use
  - respect the execution semantics defined in the SLDL language reference manual (LRM)
- improve observability of parallel execution and the debugging process for building system-level models with parallelism

1.4 Dissertation Overview

This dissertation is organized as follows:

After the introduction to the research background of this work in Chapter 1, a new model of computation for system-level design, namely ConcurrenC, is defined in Chapter 2. The essential features for system-level design can be reflected in the ConcurrenC model and be expressed in system-level description languages. Chapter 2 lays the research context for the work in this dissertation.

Chapter 3 describes the simulator kernel extension for synchronous parallel discrete event simulation (SPDES). The cooperative user level threads are replaced with operating system kernel level threads so as to allow real parallelism during simulation. The SpecC simulator kernel is extended for scheduling concurrent threads with respect to the discrete event execution semantics. More importantly, thread synchronization and communication are protected by automatic code instrumentation which is transparent to the users.

Chapter 4 proposes the idea of the out-of-order parallel discrete event simulation (OoO PDES). This advanced simulation approach breaks the global simulation time barrier to allow threads in different simulation cycles to run in parallel. With the help of code analysis performed statically at compile time and conflict-aware scheduling at run time, out-of-order parallel simulation can significantly increase simulation parallelism without loss of simulation correctness or timing accuracy. The *segment graph* data structure is highlighted in this chapter as the backbone for OoO PDES.

Chapter 5 focuses on optimizations for the out-of-order parallel discrete event simulation. First, the static code analysis algorithm is optimized by using the concept of instance isolation. Then, an optimized scheduling approach using prediction is discussed. These two optimizations lead to significant simulation speedup with negligible compilation overhead.

Chapter 6 gives a comparison among the three discrete event simulation kernels for system-level description languages, i.e. the traditional sequential, the synchronous parallel, and the out-of-order parallel simulation kernels. An outlook on the experimental results is also presented.

Chapter 7 extends the utilization of the parallel simulation infrastructure to detect race conditions in parallel system-level models. Parallel accesses to shared variables can be captured during simulation and reported to the designers to narrow down the debugging targets. A tool flow for this debug process is proposed in this chapter.

Finally, Chapter 8 summarizes the contributions of the work in this dissertation and concludes with a brief discussion on the future work.

1.5 Related Work

We will describe the relevant research work in this section. A brief overview on the SpecC and SystemC languages are given in Section 1.5.1 and Section 1.5.2. Then, the SpecC design

flow, the System-on-Chip Environment (SCE), is discussed in Section 1.5.3. Multi-core technology and multithreaded programming is briefly reviewed in Section 1.5.4. Finally, the related research work for efficient system-level model validation and simulation will be presented in Section 1.5.5.

1.5.1 The SpecC Language

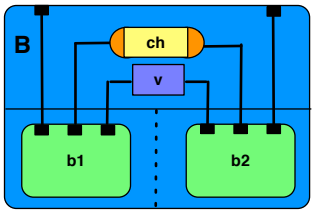
The SpecC language is a system-level description language as well as a design methodology. It was invented by D. Gajski's group from the Center for Embedded Computer Systems at the University of California, Irvine [18]. SpecC is an superset of the ANSI-C language with extended constructs to describe hardware and system features. We will give an overview on these constructs as follows:

- **Structural Hierarchy**

  A SpecC program is a collection of classes of type ***behavior***, ***channel***, and ***interface***. ***Behavior*** is the class for describing the computation in the design. The definition of it consists of a set of ports, member variables, member methods, instances of child behaviors, and a mandatory *main* method. ***Interface*** is the abstract class for the declaration of communication functions. ***Channel*** is the class which implements the *interface* functions for the communication in the design.

  Fig. 1.6 shows the block diagram of a SpecC model with two behaviors b_1 and b_2 nested in the parent behavior b, and communicating via channel ch through interface ports and a shared variable v through variable ports. The three types of SpecC classes can directly reflect the structure of the design in the form of a hierarchical network of behaviors and channels. The description of computation and communication can also be easily separated.

  Basically, all C programs are valid SpecC descriptions. However, instead of starting from the *main* function, the SpecC program execution starts from the *main* method in the



(a) Block diagram of a SpecC model (source [17])

```
behavior B(in int p1, out int p2)
{
  int v;
  double_handshake_channel ch;
  B1 b1 (p1, v, ch);
  B2 b2 (p2, v, ch);

  void main()
  {
      par{ b1.main();
           b2.main(); }
  }
};
```

(b) The SpecC source code

Figure 1.6: The block diagram and source code of a SpecC model

behavior named *Main*.

- **Types**

  SpecC supports all the data types in the C language:

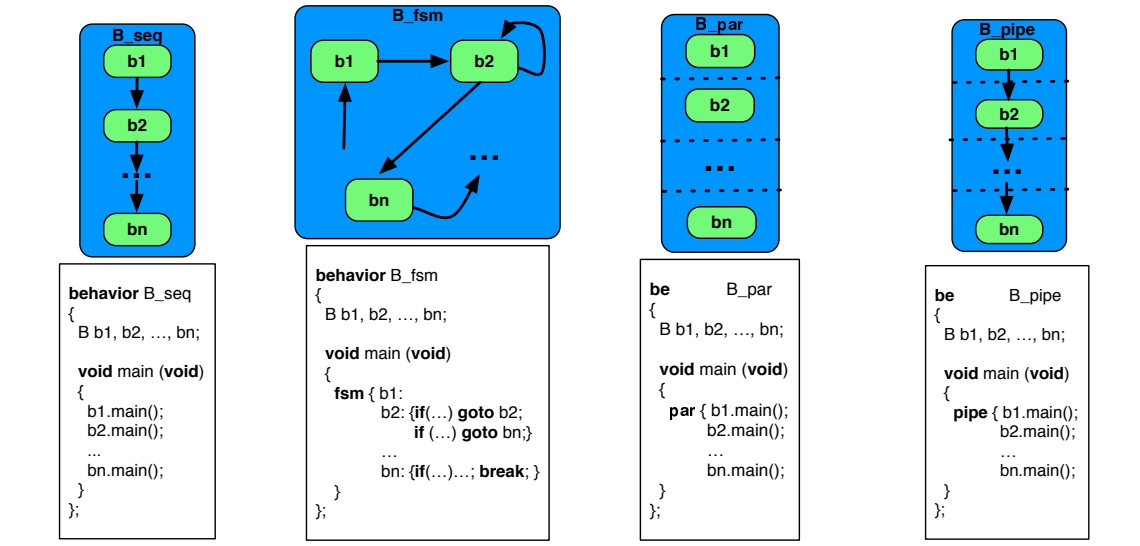
  - basic types, such as *char*, *int* and *double*;
  - composite types such as *arrays* and *pointers*;
  - user-defined types such as *struct*, *union* and *enum*.

  Moreover, SpecC also supports type *bool* for boolean data, type *bit vectors* for describing hardware features, and type *event* for synchronization and exception handling.

- **Behavioral Hierarchy**

  Behavioral hierarchy describes the dynamic execution order of the child behaviors in time. SpecC supports two types of composition of child behaviors: *sequential* and *concurrent*. *Sequential* execution can be specified as a set of sequential statements (Fig. 1.7a), or as a finite state machine (FSM) with a set of explicit state transitions (Fig. 1.7b). *Concurrent* execution can be specified as a set of behavior instances running in parallel (Fig. 1.7c) or

in a pipelined fashion (Fig. 1.7d).



(a) Sequential Execution (b) FSM Execution (c) Parallel Execution (d) Pipelined Execution

Figure 1.7: SpecC behavioral hierarchy block diagram and syntax [18]

More specifically, according to [18], the execution semantics for behavioral hierarchy is defined as follows :

- **Sequential**: Behavior *B_seq* has n child behaviors (b_1, b_2, ..., b_n) running in a sequential order indicated as the arrows in Fig. 1.7a. The execution of *B_seq* starts from the execution of b_1 and terminates when b_n finishes. The execution of the child behaviors has a total order starting from b_1, then b_2, ..., and b_n at last. Here, b_i starts its execution right after b_{i-1} finishes. The execution of the child behaviors do not overlap. Only one behavior is running at a time.
- **Finite state machine**: a special case of sequential execution where the execution order of the child behaviors are described by state transitions. The state transitions are determined dynamically at run time by the state condition variables that are specified in the SpecC program.

As shown in Fig. 1.7b, behavior *B_fsm* is a FSM with n states (b_1, b_2, ..., b_n) wrapped in the **fsm** keyword. The initial state is b_1 and the final state is b_n. One state transits to another which is specified after the *goto*-statement when the condition in the *if*-statement is *true*. The states in FSM also run sequentially without overlap.

– **Parallel**: Behavior *B_par* has n concurrent child behaviors (b_1, b_2, ..., b_n) separated with dotted lines. Syntactically, parallel composition is specified by using the **par** keyword.

  The child behaviors are logically running in parallel and mapped onto separate threads during simulation. They start altogether when *B_par* starts. *B_par* terminates when all the child behaviors complete their execution. *Parallel* execution in SpecC is a powerful construct for modeling explicit concurrency in the system-level models. While SystemC supports explicit expression of concurrency by using processes of type *sc_method*, the execution semantics for parallel execution is slightly different between SpecC and SystemC. We will discuss the difference in more detail in Section 3.2.

– **Pipeline**: a special form of parallel execution and unique in SpecC. It is specified with the *pipe* keyword wrapping around the child behaviors that are running in a pipelined fashion. Similar to parallel execution, all the child behaviors are mapped to separate threads during simulation. However, they will be executed as a pipeline.

  As shown in Fig. 1.7d, when behavior *B_pipe* starts its first iteration, only b_1 is executed. When b_1 finishes, b_1 and b_2 will be executed in parallel in the second iteration. The second iteration completes when both b_1 and b_2 finish their execution. Then, b_3 joins b_1 and b_2 in the third iteration when the three of them run in parallel. This keeps going with b_i joins b_1 to b_{i-1} in the ith iteration and run in parallel. Finally, after the nth iteration, all the child behaviors run in parallel and continue to execute till all the behaviors finish their execution in the previous iteration. The

child behaviors in pipelined execution are separated via dotted lines in the SpecC block diagram, and the order of joining execution is indicated with the arrows.

The total number of iteration for pipeline execution can be specified as infinite or a fixed positive integral number.

- **Communication**

  Communication in SpecC can be described in **shared variables** or **channels** between behaviors.

  As shown in Fig. 1.6, the communication variable v is defined in behavior B and is connected to the ports of behavior b_1 and b_2. *Shared variables* represent the wire for communication when connected to the ports of behaviors. The value will be hold as in memory and can be accessed by either of the connected behaviors.

  The communication in SpecC can also be represented by **leaf channels** or **hierarchical channels**. *Leaf channels* are virtual high-level channels which are not related to a real implementation, such as a bus protocol. *Hierarchical channels* are channels which contain children channel instances. They can represent more complex channels, such as a communication protocol stack.

  The channels usually have their own member variables and functions for communication. Typically, the communication functions in the channels are available to the connected behaviors through the ports of type *interface*.

- **Synchronization**

  Synchronization is required for the cooperation and communication among concurrent behaviors. SpecC defines the data-type ***event*** as the basic unit of synchronization, and uses them in the ***wait***, ***notify*** and ***notifyone*** statement.

  The *wait* statement suspend the current thread and does not resume the thread until the event that is waited on is notified. The *notify* statement notifies one or multiple events so

as to trigger all the threads that are waiting on those events. The *notifyone* is a variant of *notify*. It also notifies one or multiple events but only triggers one thread that are waiting on the events. Which suspended thread to trigger by the *notifyone* statement depends on the implementation of the SpecC simulator.

- **Exception Handling**

  SpecC supports two types of exception handling: ***interrupt*** and ***abortion***.

  When *interrupt* happens, the interrupted behavior will suspend to let the interrupt handling behavior to take over the execution. After the handling behavior completes, the interrupted behavior will resume its execution. In contrast to *interrupt*, an *aborted* behavior will not resume its execution after the exception handling behavior finishes.

  Syntactically, *interrupt*s and *abortion*s are implemented by using the *try-interrupt* and *try-trap* constructs respectively. The interrupted (aborted) behavior is wrapped in the *try* block. It is sensitive to a list of events, each of them is associated with an exception handling behavior and wrapped in the *interrupt* (*trap*) block.

- **Timing**

  There are two types of timing specification in SpecC: ***exact timing*** and ***timing constraints***.

  The *exact timing* specifies the delay and execution time in the model. Syntactically, it is implemented by using the *waitfor* construct with an integral argument which indicates the amount of time to wait.

  The *timing constraints* specify the design expectation on the duration of a set of labelled statements or behaviors. The constraints do not affect the simulation scheduling. They are only validated dynamically at runtime and fail if the logic timing does not meet the specified constraints. Syntactically, *timing constraints* are implemented by using the *do-timing* and *range* constructs.

- **Persistent Annotation**

  Persistent annotation is specified by using the *note* construct in SpecC. It is similar to a *comment* in the C language which adds additional annotations in the program to increase the readability. However, in contrast to *comments*, *persistent annotations* will not be removed by the SpecC compiler so that the other synthesis tools can utilize them as guidelines from the designer.

- **Library Support**

  For flexible component handling and speedy compilation, SpecC supports the concept of library. Precompiled design modules can be added into a SpecC design by using the *import* construct.

Fig. 1.8 shows the compilation flow for a SpecC design. The SpecC compiler (***scc***) starts with a SpecC source code file (Design.sc) with suffix **.sc** be processed by the *Preprocessor* generating an output of **.si** file (Design.si). Next, the preprocessed SpecC design is fed into the *Parser* to build the *SpecC internal representation (SIR)* data structure in the memory. Then, the *Translator* generates the C++ description of the design (Design.cc and Design.h). Finally, the C++ program is compiled and linked by the standard C++ compiler to build the executable binary image for simulation.

The *SIR* data structure is the computer representation of the SpecC design. The *Refinement tools* for SpecC design are based on the SIR data structure, and can modify it based on the designers' requirements. The *Exporter* can dump the SIR in the memory into a binary **.sir** file. The *Importer* can read a **.sir** file and build the SIR data structure in the memory.

The work in this dissertation is using the SpecC language. We will extend the simulation kernel and the compiler support for SpecC. More details on the syntax and features of SpecC can be found in [18].

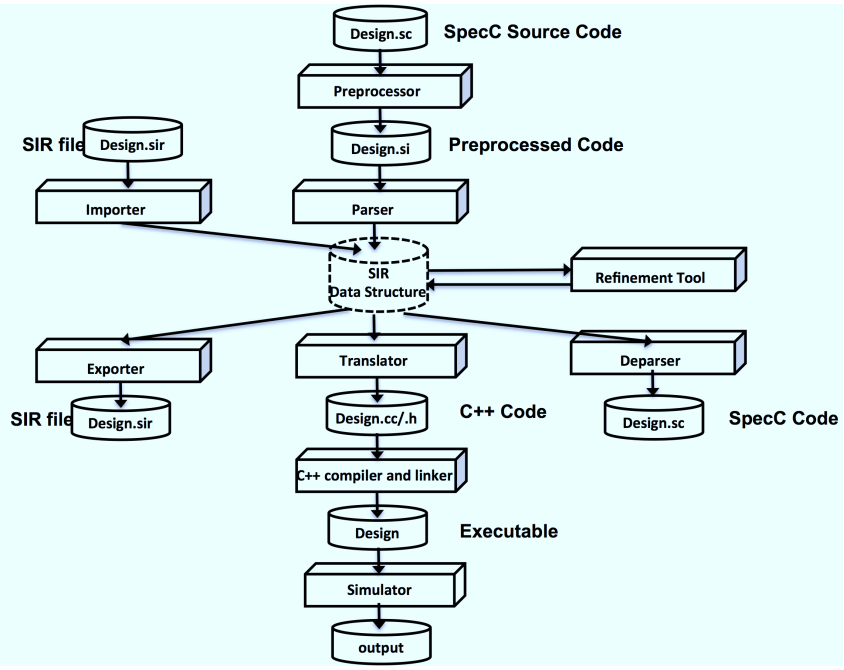


Figure 1.8: Compilation flow for a SpecC Design (source [45])

1.5.2 The SystemC Language

SystemC has a layered language architecture. As shown in Fig. 1.9 [1], SystemC is built entirely on the C++ standard. The *core language* is composed of a discrete-event simulation kernel and some abstract constructs for describing system elements, such as modules, ports, processes, events, interfaces, channels, etc. Alongside the *core language* is a set of basic *Data-Types* for logic values, bits and bit-vectors, fixed-pointer numbers, and so on. The elementary channels are depicted above the *core language* and *data-types* for generic applicable models, such as signal, semaphores, and first-in-first-out buffers (FIFOs). More standard and methodology-specific channels can be built on top of the elementary channels.

Similar as SpecC, SystemC also has the constructs for system-level design, such as time model, module hierarchy for structure and connectivity management, concurrency models, and com-

[1] Not all the blocks in Fig. 1.9 are considered as part of the SystemC standard. Only the one with double borders belong to the SystemC standard.

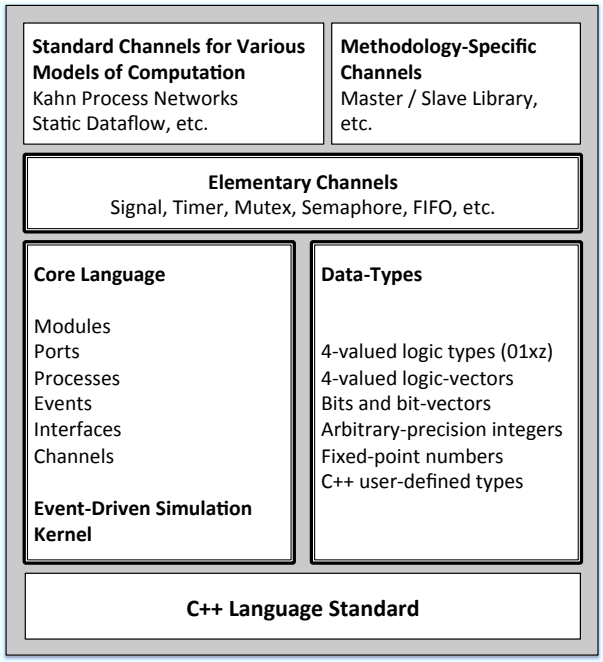


Figure 1.9: SystemC language architecture (source [38])

munication management between concurrent units, all of which are not normally available in software languages [39]. In particular, the base class ***sc_module*** is the base class for all the SystemC modules which specifies the structure and hierarchy in the system. ***SC_METHOD*** and ***SC_THREAD*** are the macros to define simulation processes. ***SC_METHOD*** is a member function of an ***sc_module*** without simulation time (cycle) advance, whereas ***SC_THREAD*** is the one which allows time to pass and behaves similar to a traditional software thread. ***sc_port***, ***sc_interface***, and ***sc_channel*** are the constructs for describing communication. Last but not least, SystemC also supports the concepts of ***event (sc_event)***, ***sensitivity***, and ***notification*** for the synchronization among concurrent SystemC threads.

1.5.3 The System-on-Chip Environment (SCE) Design Flow

The experimental applications used in this dissertation are developed and refined in the System-on-Chip Environment (SCE) framework [46].

The SCE framework supports a systematic top-down flow based on the *specify-explore-refine* methodology [13]. It is a framework for SpecC-based heterogenous MPSoC design which may comprise embedded software processors, custom hardware components, dedicated intellectual property blocks, and complex communication stacks.

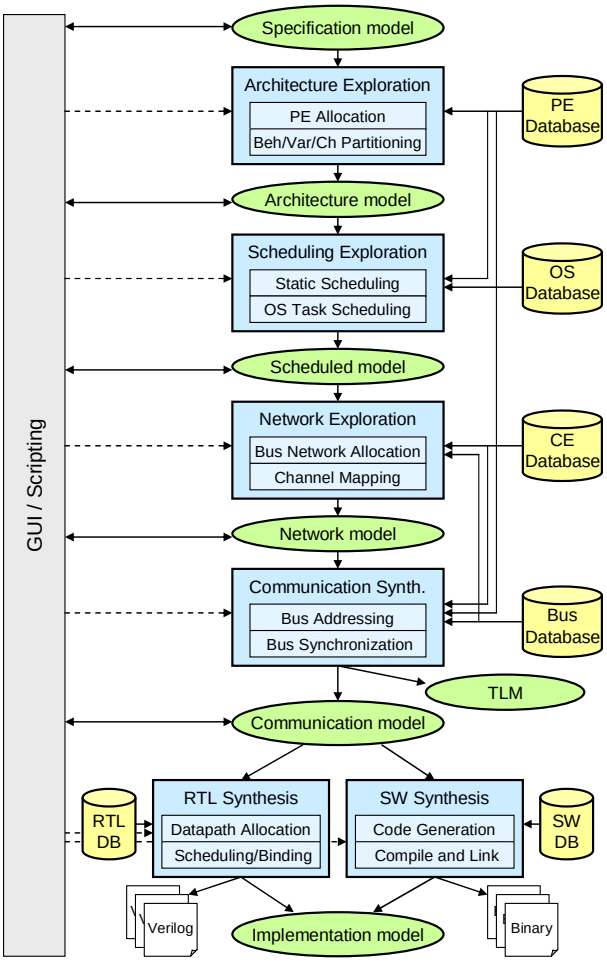


Figure 1.10: Refinement-based design flow in the SCE framework (source [46])

The SCE flow starts with a specification model of the intended design, and then refines the model with increasing amount of detail to a final implementation which is pin- and cycle-accurate. The model refinement is performed by a set of tools that are integrated in the SCE framework with a large database library of pre-designed components. Most importantly, the SCE flow keeps

the designer in the loop to make critical design decisions through an interactive graphical user interface with powerful scripting capabilities.

As an overview shown in Fig. 1.10, the SCE framework has four refinement stages and one backend stage for refinement synthesis:

- *Architecture Exploration* allows the designer to allocate the intended hardware platform with software processors, hardware accelerators, and communication units that are provided by the SCE library. The designer can then map the modules in the specification onto different components in the allocated platform. After the decisions for architecture allocation and mapping are made, the refinement tool in SCE automatically partitions the specification and generates a new model to reflect the architecture mapping.

- *Scheduling Exploration* allows the designer to choose the scheduling strategies for the software processors. SCE offers both dynamic and static scheduling strategies such as priority-based, round-robin, or first-come-first-serve algorithms. The SCE refinement tool will then automatically insert abstract RTOS models which support real-time scheduling, tasks management and synchronization, preemption, and interrupt handling, into the refined model.

- *Network Exploration* is the stage where the designer specifies the communication topology of the entire system and maps the abstract channels onto network busses and bridges from the SCE library. At the end of the network exploration phase, the SCE tool will automatically replace the channel modules with the communication components from the database, and establish the point-to-point communication links.

- *Communication Synthesis* automatically refines the point-to-point links into an implementation with actual bus protocols and wires. As a result, a pin- and bit-accurate communication model is obtained. In addition to the pin-accurate model (PAM), SCE can alternatively generate a corresponding Transaction Level Model (TLM) with abstractions

for bus transactions for the purpose of fast simulation.

In the exploration phases, the designers are highly involved in the process for making design decisions. The SCE tool takes care of the automatic model refinement based on the designer's choices. Moreover, the SCE tool validates the models after each exploration stage and provides estimation numbers of the platform. The designer can then tune their design choices for further optimization.

After the system refinement steps, the SCE tool will perform the *backend* tasks as follows:

- *RTL Synthesis* implements the hardware units in the model. The SCE tool automatically produces the structural RTL model from the behavior model of the hardware components. The RTL description can either be in *Verilog* HDL for further logic synthesis or in SpecC SLDL for system simulation.
- *Software Synthesis* generates the program code running on the software processors. The SCE tool supports a layer-based software stack executing on the programmable software processors. The software stack includes the application code, the configured RTOS, and the inlining communication code. The software is then cross-compiled and linked as a binary image which can be executed on the final target platform as well as in the instruction-set simulator within the system context.

The SCE framework is based on the SpecC SLDL and uses discrete event (DE) simulation for model validation and performance estimation after each exploration stage.

1.5.4 Multi-core Technology and Multithreaded Programming

Moore's law predicts that the chip performance doubles every 18 months. While the increasing clock frequency, the shrinking size of the transistors, and the reducing operating voltage were the major contributing factors to chip performance improvement, they are no longer easy to achieve due to material physical limitations, such as the power consumption and heat dissipation

issues [47].

Multi-core technology has become more and more popular in the recent one decade as an alternative way to achieve more computational capability without hitting the "power wall" [2] or breaking the physical limitations. Basically, a **multi-core processor** is a computing component with two or more central processing units (CPUs) in a single physical package. The CPUs on a multi-core chip may have their own or shared cache units, register sets, floating-point processing units, and other resources. They share the main memory. In contrast to several single core chips, a multi-core processor is easier to cool down since the CPUs are usually simpler with fewer transistors. As such, multi-core processors can provide more processing capability with less heat dissipation.

The biggest challenge for utilizing the multi-core technology is how to program the software that can run efficiently on the underlying platform. The programming philosophy for most of the existing software languages is fundamentally sequential. It is not trivial to rewrite sequential program with parallelism, nor to write parallel ones from scratch. Moreover, high multi-core CPU utilization requires balanced workload among the cores. This is very difficult to achieve since the partitioning problem is known to be NP-complete [49].

Multithreaded programming is widely used for abstracting concurrency in the software. In computer science, a *process* is an instance of computer program that can be executed, and a *thread* is a light-weight *process*. Processes usually have their own address space and resources. On the other hand, threads within one process share resources such as memory, code text, global data, and files.

A multithreaded program is a process which has multiple active threads at run time. Each thread executes its own set of sequential instructions and can run in parallel with the other threads. Multithreaded program can have true concurrent execution on multi-core computer platforms.

[2] As defined in [48], "power wall" means the limit on the amount of power a microprocessor chip could reasonably dissipate.

In such cases, programming discretions need to be taken to manage threads synchronization properly, avoid race conditions on shared variables, and avoid execution deadlocks.

Multithreaded programming is supported as libraries and constructs in some capacity in most high-level programming languages. *C/C++* supports standardized multithreaded API libraries, such as *Posix Threads* [50, 51], *OpenMP* [52], and *Message Passing Interface* [53]. Accesses to the native threads APIs are also available on different operating systems, such as *Native Posix Thread Library* (NPTL) on Linux [54] and *Win32 Threads* on Windows [55]. *Java* has language-level multithreading support, such as the *Runnable* interface, the *Thread* class, and the "synchronized" key word. The logic threads in Java programs are mapped to native ones in the underlying operating system through the *Java Virtual Machine* (JVM). Some interpreted languages, such as *Ruby* and *Python*, support limited multithreading with a *Global Interpreter Lock* (GIL) to prevent concurrent interpreting on multiple threads. There are also languages that are particularly designed for parallel programming, such as *Erlang* [56], *Cilk* [57, 58], *CUDA* [59, 60], and so on.

1.5.5 Efficient Model Validation and Simulation

Parallel Discrete Event Simulation (PDES) is a well-studied subject in the literature [63, 64, 65]. Two major synchronization paradigms can be distinguished, namely *conservative* and *optimistic* [64]. *Conservative* PDES typically involves dependency analysis and ensures in-order execution for dependent threads. In contrast, the *optimistic* paradigm assumes that threads are safe to execute and rolls back when this proves incorrect. Often, the temporal barriers in the model prevent effective parallelism in conservative PDES, while rollbacks in optimistic PDES are expensive in implementation and execution.

Distributed parallel simulation, such as [63] and [66], is a natural extension of PDES. Distributed simulation breaks the design model into modules, dispatches them on geographically distributed hosts, and then runs the simulation in parallel. However, model partitioning is difficult and the

network speed becomes a simulation bottleneck due to the frequently needed communication in system-level models.

Related work on improving simulation speed in the broader sense can be categorized into software modeling and specialized hardware approaches.

Software techniques include the general idea of transaction-level modeling (TLM) [67], which speeds up simulation by higher abstraction of communication; and source-level [68] or host-compiled simulation [69] which abstract the computation from the target platform. Generally, these approaches trade-off simulation speed against a loss in accuracy, for example, approximate timing due to estimated or back-annotated values.

Temporal decoupling proposed by SystemC TLM [70] also trades off timing accuracy against simulation speed. Simulation time is incremented separately in different threads to minimize synchronization, but accuracy is reduced. Our approach proposed in Chapter 4 also localizes simulation time to different threads, but fully maintains accurate timing.

Specialized hardware approaches include the use of Field-Programmable Gate Array (FPGA) and Graphics Processing Unit (GPU) platforms. For example, [71] emulates SystemC code on FPGA boards and [72] proposes a SystemC multithreading model for GPU-based simulation. [73] presents a methodology to parallelize SystemC simulation across multi-core CPUs and GPUs. For such approaches, model partitioning is difficult for efficient parallel simulation on the heterogeneous simulator units.

Other simulation techniques change the infrastructure to allow multiple simulators to run in parallel and synchronize as needed. For example, the Wisconsin Wind Tunnel [74] uses a conservative time bucket synchronization scheme to synchronize simulators at a predefined interval. Another example [75] introduces a simulation backplane to handle the synchronization between wrapped simulators and analyzes the system to optimize the period of synchronization message transfers. Both techniques significantly speedup the simulation, however, at the cost

of timing accuracy.

2 The ConcurrenC Model of Computation

Embedded system design in general can only be successful if it is based on a suitable Model of Computation (MoC) that can be well represented in an executable System-level Description Language (SLDL) and is supported by a matching set of design tools. While C-based SLDLs, such as SystemC and SpecC, are popular in system-level modeling and validation, current tool flows impose serious restrictions on the synthesizable subset of the supported SLDL. A properly aligned and clean system-level MoC is often neglected or even ignored.

In this chapter, we propose a new MoC, called *ConcurrenC*, that defines a system-level of abstraction, fits system modeling requirements, and can be expressed in both SystemC and SpecC SLDLs [76].

2.1 Motivation

For system-level design, the importance of abstract modeling cannot be overrated. Proper abstraction and specification of the system model is a key to accurate and efficient estimation and the final successful implementation.

Register-Transfer Level (RTL) design is a good example to show the importance of a well-defined model of computation. Designers describe hardware components in hardware description languages (HDL), i.e. VHDL and Verilog. Both languages have strong capabilities to support different types of hardware structures and functionalities. By using the HDL, designers use

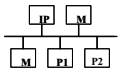
Abstraction Level	Schematics	Language	MoC	Tool
RTL		VHDL, Verilog	FSM, FSMD	Synopsys Design Compiler Cadence RTL Compiler ...
ESL		SpecC, SystemC	PSM, TLM (?) *ConcurrenC !*	SoC Environment [46] Synopsys System Studio ...

Table 2.1: System-level design in comparison with the well-established RTL design

Finite State Machines (FSMs) to model controllers or other parts of their design. Thus, FSM plays a crucial role as a formal model behind the languages. In other words, the FSM model in the mind of the designer is described syntactically in the VHDL or Verilog language to implement a hardware design.

Note that commercial computer aided design (CAD) tools cannot synthesize all the VHDL / Verilog statements. Instead, special design guidelines are provided to restrict the use of specific syntax elements, or to prevent generation of improper logics, e.g. latches.

The importance of the model in system design is the same as in RTL. Table 2.1 compares the situation at the system-level against the mature design methodology at the RTL. RTL design is supported by the strong MoCs of FSM and FSMD, and well-accepted *coding guidelines* exist for the VHDL and Verilog languags, so that established commercial tool chains can implement the described hardware. It is important to notice that here the MoC was defined first, and the coding style in the respective HDL followed the needs of the MoC.

At the Electronic System Level (ESL), on the other hand, we have the popular C-based SLDLs SystemC and SpecC which are more or less supported by early academic and commercial tools [46, 77]. As at RTL, the languages are restricted to a (small) subset of supported features, but these *modeling guidelines* are not very clear. Moreover, the MoC behind these SLDLs is unclear. SpecC is defined in context of the Program State Machine (PSM) MoC [36], but so is SpecCharts [78] whose syntax is entirely different. For SystemC, one could claim Transaction

Level Model (TLM) as its MoC [38], but a wide variety of interpretations of TLM exists.

We can conclude that in contrast to the popularity of the C-based SLDLs for ESL modeling and validation, and the presence of existing design flows implemented by early tools, the use of a well-defined and clear system-level MoC is neglected. Instead, serious restrictions are imposed on the usable (i.e. synthesizable and verifiable) subset of the supported SLDL. Without a clear MoC behind these syntactical guidelines, computer-aided system design is difficult. Clearly, a well-defined and formal MoC is needed to tackle the ESL design challenge.

2.2 Models of Computation (MoCs)

Edwards et al. argue in [79] that the design approach should be based on the use of formal methods to describe the system behavior at a higher level of abstraction. A Model of Computation (MoC) is such formal method for system design. MoC is a formal definition of the set of allowable operations used in computation and their respective costs [80]. This defines the behavior of the system is at certain abstract level to reflect the essential system features. Many different models of computation have been proposed for different domains. An overview can be found in [78] and [81].

Kahn Process Network (KPN) is a deterministic MoC where processes are connected by unbounded FIFO communication channels to form a network [82]. *Dataflow Process Network (DFPN)* [82], a special case of *KPN*, is a kind of MoC in which dataflow can be drawn in graphs as process network and the size of the communication buffer is bounded. *Synchronous dataflow (SDF)* [83], *Cyclo-static dataflow (CSDF)* [84], *Heterochronous dataflow (HDF)* [85], *Parameterized Synchronous dataflow (PSDF)* [86], *Boolean dataflow (BDF)* [87], and *Dynamic dataflow (DDF)* [88] are extended MoCs from the *DFPN* to provide the features like static scheduling, predetermined communication patterns, finite state machine (FSM) extension, reconfiguration, boolean modeling, and dynamic deadlock and boundedness analysis. These MoCs are popular

for modeling signal processing applications but not suited for controller applications.

Software/hardware integration medium (SHIM) [89] is a concurrent asynchronous deterministic model which is essentially an effective KPN with rendezvous communication for heterogeneous embedded systems.

Petri Net [90], an abstract, formal model of information flow, is a state-oriented hierarchical model, especially for the systems that being concurrent, distributed, asynchronous, parallel, non-deterministic or stochastic activities. However, it is uninterpreted and can also quickly become incomprehensible with any system complexity increase.

Dataflow Graph (DFG) and its derivatives are MoCs for describing computational intensive systems [36]. It is very popular for describing digital signal processing (DSP) components but is not suitable to represent control parts which are commonly found in most programming languages.

Combined with **Finite State Machine (FSM)** which is popular for describing control systems, *FSM* and *DFG* form *Finite State Machine with Datapath (FSMD)* in order to describe systems requiring both control and computation. *Superstate Finite-State Machine with Datapath (SFSMD)* and *hierarchical concurrent finite-state machine with Datapath(HCFSMD)* are proposed based on FSMD to support the hierarchical description ability with concurrent system features for behavioral synthesis.

Program State machine (PSM) [78] is an extension of *FSMD* that supports both hierarchy and concurrency, and allows states to contain regular program code.

Transaction-level modeling (TLM) [38] is a well-accepted approach to model digital systems where the implementation details of the communication and functional units are abstracted and separated. TLM abstracts away the low level system details so that executes dramatically faster than synthesizable models. However, high simulation speed is traded in for low accuracy, while a high degree of accuracy comes at the price of low speed. Moreover, TLM does not specify a

well-defined MoC, but relies on the system design flow and the used SLDL to define the details of supported syntax and semantics.

2.3 ConcurrenC MoC

Although we have the mature and industrially used system-level description languages, like SpecC and SystemC, we do not have a formal model for both of them. In this section, we will discuss the close relationship and tight dependencies between SLDLs (i.e. syntax), their expressive abilities (i.e. semantics), and the abstract models they can represent. In contrast to the large set of models the SLDL can describe, the available tools support only a subset of these models. To avoid this discrepancy that clearly hinders the effectiveness of any ESL methodology, we propose a novel MoC, called *ConcurrenC*, that fits the system modeling requirements and the capabilities of the supporting tool chain and languages.

Generally speaking, *ConcurrenC* should be a system-level FSM extension with support for concurrency and hierarchy. As such, it falls into the PSM MoC category. The *ConcurrenC* model needs clear separation of concerns on computation and communication. In the realm of structure abstraction, a *ConcurrenC* model consists of blocks, channels and interfaces, and fully supports structural and behavioral hierarchy. Blocks can be flexibly composed in space and time to execute sequentially, in parallel/pipelined fashion, or by use of state transitions. Blocks themselves are internally based on C, which the most popular programming language for embedded applications. In the realm of communication abstraction, we intentionally use a set of predefined channels that follow a typed message passing paradigm rather than using user-defined freely programmable channels.

2.3.1 Relationship to C-based SLDLs

More specifically, *ConcurrenC* is tailored to the SpecC and SystemC SLDLs. *ConcurrenC* abstracts the embedded system features and provides clear guidelines for the designer to efficiently use the SLDLs to build a system. In other words, the *ConcurrenC* model can be captured and described by using the SLDLs.

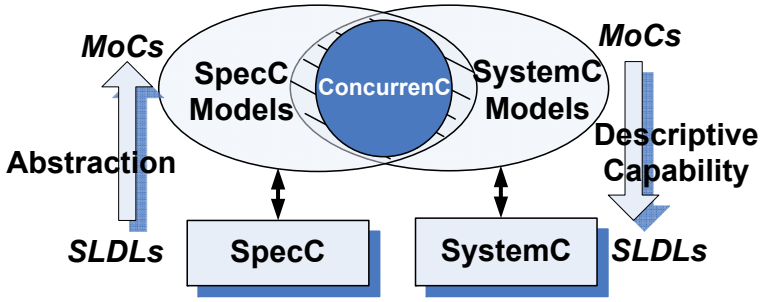


Figure 2.1: Relationship between C-based SLDLs SystemC and SpecC, and MoC ConcurrenC

Fig. 2.1 shows the relationship between the C-based SLDLs, SystemC and SpecC, and the MoC, *ConcurrenC*. *ConcurrenC* is a true subset of the models that can be described by SpecC and SystemC. This implies that *ConcurrenC* contains only the model features which can be described by both languages. For example, exception handling, i.e. interrupt and abortion, is supported in SpecC by using the *try-trap* syntax, but SystemC does not have the capability to handle such exceptions. On the other hand, SystemC supports the feature for waiting a certain time *and* for some events at the same time, which SpecC does not support. As shown in Fig. 2.1, features that are only supported by one SLDL will not be included in the *ConcurrenC* model.

Moreover, *ConcurrenC* excludes some features that both SpecC and SystemC support (the shadow overlap area in Fig. 2.1). We exclude these to make the *ConcurrenC* model more concise for modeling. For example, *ConcurrenC* will restrict its communication channels to a predefined library rather than allowing the user to define arbitrary channels by themselves. This allows tools to recognize the channels and implement them in an optimal fashion.

2.3.2 ConcurrenC Features

A *ConcurrenC* Model can be visualized in four dimensions as shown in Fig. 2.2. There are three dimensions in space, and one in time. The spatial dimensions consist of two dimensions for structural composition of blocks and channels and their connectivity through ports and signals (X, Y coordinates), and one for hierarchical composition (Z-axis). The temporal dimension specifies the execution order of blocks in time, which can be sequential or FSM-like (thick arrows), parallel (dashed lines), or pipelined (dashed lines with arrows) in Fig. 2.2.

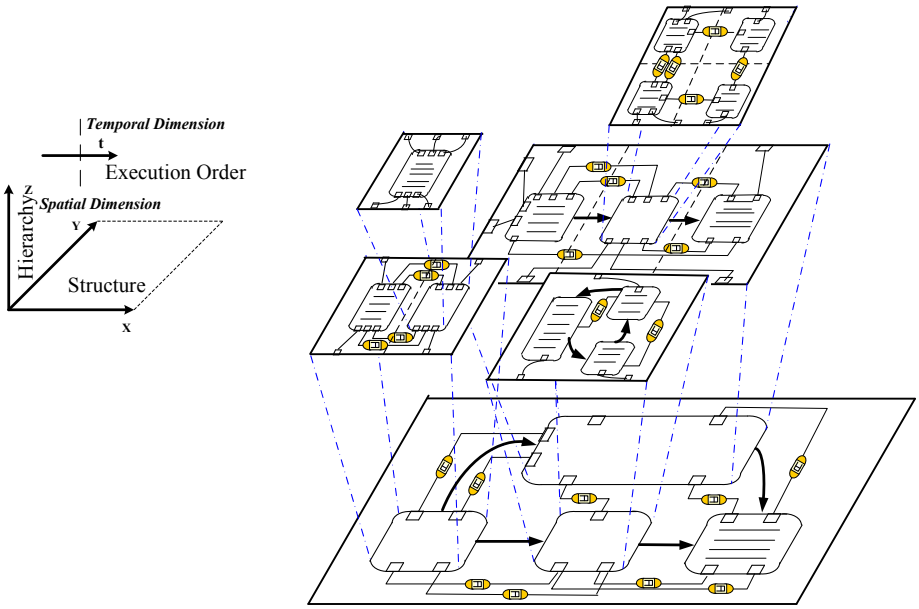


Figure 2.2: Visualization of a ConcurrenC Model in three spatial and one temporal dimensions

The detailed features of the proposed *ConcurrenC* MoC are listed below:

- **Communication & Computation Separation** Separating communication from computation allows "plug-n-play" features of the embedded system [36]. In *ConcurrenC*, the communication contained in channels is separated from the computation part contained in blocks so that the purpose of each statement in the model can be clearly identified whether it is for communication or computation. This also helps for architecture refinement and

hardware/software partitioning.

- **Hierarchy** Hierarchy eliminates the potential explosion of the model size and significantly simplifies comprehensible modeling of complex systems.
- **Concurrency** The need for concurrency is obvious. A common embedded system will have multiple hardware units work in parallel and cooperate through specified communication mechanisms. *ConcurrenC* also supports pipelining in order to provide a simple and explicit description of the pipelined data flow in the system.
- **Abstract Communications (Channels)** A predefined set of communication channels is available in *ConcurrenC*. We believe that the restriction to predefined channels not only avoids coding errors by the designer, but also simplifies the later refinement steps, since the channels can be easily recognized by the tools.
- **Timing** The execution time of the model should be evaluable to observe the efficiency of the system. Thus, *ConcurrenC* supports wait-for-time statements in similar fashion as SystemC and SpecC.
- **Execution** The model must be executable in order to show its correctness and obtain performance estimation. Since a *ConcurrenC* model can be converted to SpecC and SystemC, the execution of the model is thus possible.

2.3.3 Communication Channel Library

For *ConcurrenC*, we envision two type of channels, channels for synchronization and data transfer. For data transfer, *ConcurrenC* limits the channel to transfer data in FIFO fashion (as in KPN and SDF). In many cases, these channels make the model deterministic and allow static scheduling. For KPN-like channels, the buffer size is infinite ($\mathbf{Q}_\infty$) which makes the model deadlock free but not practical. For SDF-like channels, the buffer size is fixed ($\mathbf{Q}_n$). Double-handshake mechanism, which behaves in a rendezvous fashion, is also available as a FIFO with

buffer size of zero ($\mathbf{Q}_0$). Signals are needed to design a 1-N (broadcasting) channel. Furthermore, shared variables are allowed as a simple way of communication that is convenient especially in software. Moreover, FIFO channels can be used to implement semaphore which is the key to build synchronization channels. In summary, *ConcurrenC* supports the predefined channel library as shown in Table 2.2.

Channel Type	Receiver	Sender	Buffer Size
Q_0	Blocking	Blocking	0
Q_n	Blocking	Blocking	n
Q_∞	Blocking	–	∞
Signal	Blocking	–	1
Shared Variable	–	–	1

Table 2.2: Parameterized Communication Channels

2.3.4 Relationship to KPN and SDF

With the features we discussed above, it is quite straightforward to convert the major MoCs, KPN and SDF into *ConcurrenC*.

The conversion rules from KPN (SDF) to *ConcurrenC* are:

- $\forall$ processes $\in$ KPN (SDF): convert into *ConcurrenC* blocks.
- $\forall$ channels $\in$ KPN (SDF): convert into *ConcurrenC* channels of type Q_∞ (Q_n).
- Keep the same connectivity in *ConcurrenC* as in KPN (SDF).
- If desired, group blocks in hierarchy and size the KPN channels for real-world implementation.

The conversion rules from SDF to *ConcurrenC* are:

- $\forall$ actors $\in$ SDF: convert into *ConcurrenC* blocks.

- $\forall$ arcs $\in$ SDF: convert into *ConcurrenC* channels of type Q_n where n is the size of the buffer.
- keep the same connectivity in *ConcurrenC* as in SDF.
- If desired, group blocks in hierarchy.

As such, *ConcurrenC* is essentially a superset MoC of KPN and SDF. Also it becomes possible to implement KPN and SDF into SpecC and SystemC by using *ConcurrenC* as the intermediate MoC. Moreover, note that *ConcurrenC* inherits the strong formal properties of KPN and SDF, such as static schedulability and deadlock-free guarantees.

2.4 Case Study

In order to demonstrate the feasibility and benefits of the *ConcurrenC* approach, we use the Advanced Video Coding (AVC) standard H.264 decoding algorithm [91] as a driver application to evaluate the modeling features. Our H.264 decoder model is of industrial size, consisting of about 40 thousand lines of code. The input of the decoder is an H.264 stream file, while the output is a YUV file.

ConcurrenC features can be easily used to model the H.264 decoder, see Fig. 2.3.

- **Hierarchy**: At the top level of the *ConcurrenC* model, there are three behavioral blocks: **stimulus, decoder, and monitor**. The **stimulus** reads the input YUV file, while the **monitor** receives and displays the decoded stream including signal-to-noise ratio (SNR), system time, and writes the reconstructed frames into the output file. **Decoder** contains multiple blocks for concurrent slice decoding. A stream processing block prepares the settings, n decode units decode slices in parallel, and the decoding synchronizer combines the decoded slices for output by the monitor. The number of the slice decoders is scalable depending on the number of slices contained in one frame of the input stream file. Inside

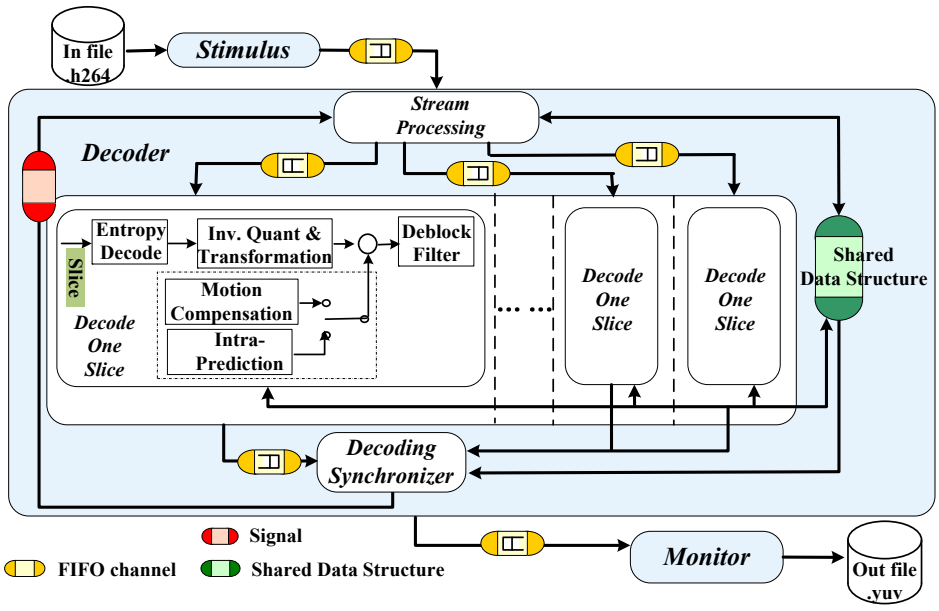


Figure 2.3: Proposed H.264 Decoder Block Diagram

the slice decode blocks, functional sub-blocks are modeled for the detailed decoding tasks. Hierarchical modeling allows convenient and clear system description.

- ***Concurrency***: [92] confirms that multiple slices in one frame are possible to be decoded concurrently. Consequently, our H.264 decoder model consists of multiple blocks for concurrent slice decoding in one picture frame[1].

- **Communication**: FIFO channels and shared variables are used for communication in our H.264 decoder model. FIFO queues are used for data exchange between different blocks. For example, the decoder synchronizer sends the decoded frame via a FIFO channel to the monitor for output. Shared variables, i.e. reference frames, are used to simplify the coordination for decoding multiple slices in parallel.

- **Timing**: The decoding time can be observed by using wait-for-time statements in the modeled blocks. We have obtained the estimated execution time for different hardware

[1]We should emphasize that this potential parallelism was not apparent in the original C code. It required serious modeling effort to parallelize the slice decoders for our model.

architectures by using simulation and profiling tools of the SLDLs.

- **Execution**: We have successfully converted and executed our model in SpecC using the SoC Environment [46].

filename	boat.264				coastguard.264			
# macroblocks/frame	396				396			
# frames	73 (2.43 secs)				299 (9.97 secs)			
# slices/frame	4		8		4		8	
max # macroblocks/slice	150		60		150		60	
model type	seq	par	seq	par	seq	par	seq	par
host sim time (s)	4.223	4.258	4.557	4.550	12.191	12.197	12.860	12.846
estimated exec time (s)	11.13	4.43	11.49	**1.80**	18.78	**7.20**	20.31	**3.33**
speedup	1	2.51	1	6.38	1	2.61	1	6.10

Table 2.3: Simulation Results, H.264 Decoder modeled in *ConcurrenC*

Table 2.3 shows the simulation results of our H.264 decoder modeling in *ConcurrenC*. The model is simulated on a PC machine with Intel(R) Pentium(R) 4 CPU at 3.00GHz. Two stream files, one with 73 frames, and the other with 299 frames are tested. For each test file, we created two types of streams, 4 slices and 8 slices per frame. We run the model by decoding the input streams in two ways: slice by slice (seq model), and slices in one frame concurrently (par model). The estimated execution time is measured by annotated timing information according to the estimation results generated by SCE with a ARM7TDMI 400 MHz processor mapping. Our simulation results show that the parallelism of the application modeled in *ConcurrenC* is scalable. We can expect that it is possible to decode three of the test streams in real-time (bold times).

3 Synchronous Parallel Discrete Event Simulation

Effective Electronic System-Level (ESL) design frameworks transform and refine high-level designs into various transaction level models described in C-based System-level Description Languages (SLDLs) and rely on simulation for validation. The traditional cooperative Discrete Event (DE) simulation of SLDLs is not effective and cannot utilize any existing parallelism in todays' multi-core CPU hosts.

In this chapter, we will present the parallel extension for the DE simulator to support efficient simulation on multi-core hosts [94, 93, 95, 96].

3.1 Traditional Discrete Event Simulation

In both SystemC and SpecC, a traditional DE simulator is used. Threads are created for the explicit parallelism described in the models (e.g. *par*{} and *pipe*{} statements in SpecC, and *SC_THREADS* and *SC_CTHREADS* in SystemC). These threads communicate via events using *wait-for-event* construct, and advance simulation time using *wait-for-time* construct.

To describe the simulation algorithm[1], we define the following data structures and operations:

[1]The formal definition of execution semantics can be found in [97].

1. Definition of queues of threads th in the simulator:

   - **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **COMPLETE**}
   - **READY** = {th | th is ready to run}
   - **RUN** = {th | th is currently running}
   - **WAIT** = {th | th is waiting for some events}
   - **WAITFOR** = {th | th is waiting for time advance}
   - **COMPLETE** = {th | th has completed its execution}

2. Simulation invariants:

   Let **THREADS** = set of all threads which currently exist. Then, at any time, the following conditions hold:

   - **THREADS** = **READY** $\cup$ **RUN** $\cup$ **WAIT** $\cup$ **WAITFOR** $\cup$ **COMPLETE**.
   - $\forall$ A, B $\in$ **QUEUES**, A $\neq$ B : A $\cap$ B = $\varnothing$.

3. Operations on threads th:

   - **Go**(th): let thread th acquire a CPU and begin execution.
   - **Stop**(th): stop execution of thread th and release the CPU.
   - **Switch**(th_1, th_2): switch the CPU from the execution of thread th_1 to thread th_2.

4. Operations on threads with set manipulations:

   Suppose th is a thread in one of the queues, A and B are queues $\in$ **QUEUES**.

   - th = **Create**(): create a new thread th and put it in set **READY**.
   - **Delete**(th): kill thread th and remove it from set **COMPLETE**.
   - th = **Pick**(A, B): pick one thread th from set A (formal selection/matching rules can be found in [97] section 4.2.3) and put it into set B.

- **Move**(th, A, B): move thread th from set A to B.

5. Initial state at beginning of simulation:

   - **THREADS** = $\{th_{root}\}$.
   - **RUN** = $\{th_{root}\}$.
   - **READY** = **WAIT** = **WAITFOR** = **COMPLETE** = $\varnothing$.
   - $time = 0$.

DE simulation is driven by events and simulation time advances. Whenever events are delivered or time increases, the scheduler is called to move the simulation forward.

Fig. 3.1 shows the control flow of the traditional scheduler. At any time, the scheduler runs a single thread which is picked from the **READY** queue. Within a delta-cycle, the choice of the next thread to run is non-deterministic (by definition). If the **READY** queue is empty, the scheduler will fill the queue again by waking threads who have received events they were waiting for. These are taken out of the **WAIT** queue and a new delta-cycle begins.

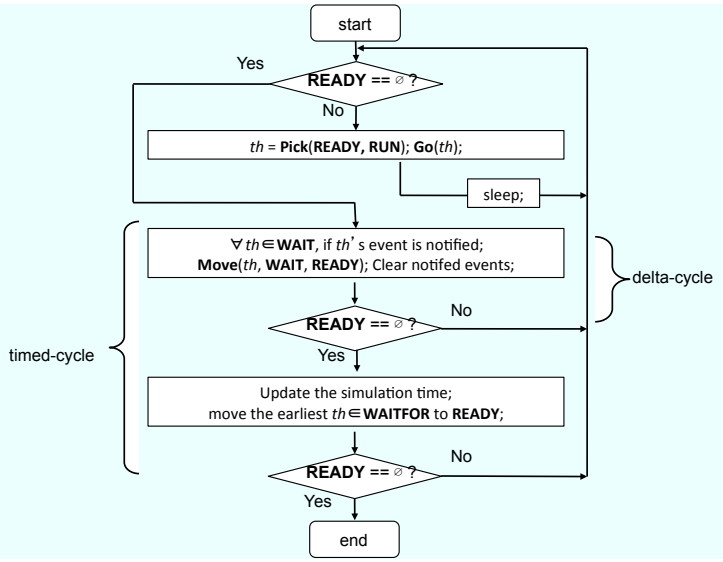


Figure 3.1: Traditional sequential Discrete Event simulation scheduler

If the **READY** queue is still empty after event delivery, the scheduler advances the simulation time, moves all threads with the earliest timestamp from the **WAITFOR** queue into the **READY** queue, and resumes execution. Traditional DE simulation uses cooperative multi-threading. Therefore, at any time, there is only one thread actively executing in the simulation.

3.2 SLDL Multi-threading Semantics

In order to allow multiple threads run in parallel, the SLDL simulation kernel needs to be extended. We need to check the SLDL execution semantics first to make sure the extension respects the definition.

Both SystemC and SpecC SLDLs define their execution semantics by use of DE-based scheduling of multiple concurrent threads, which are managed and coordinated by a central simulation kernel. More specifically, both the SystemC and SpecC reference simulators that are freely available from the corresponding consortia use cooperative multi-threading in their schedulers. That is, both reference schedulers select only a single thread to run at all times. However, the reference simulator implementations do not define the actual language semantics. In fact, the execution semantics of concurrent threads defined by the SystemC Language Reference Manual (LRM) differ significantly from the semantics defined in the SpecC LRM.

3.2.1 Cooperative multi-threading in SystemC

The SystemC LRM [40] clearly states (in Section 4.2.1.2) that *"process instances execute without interruption"*, which is known as *cooperative (or co-routine)* multitasking. In other words, preemptive scheduling is explicitly forbidden.

As a consequence, when writing a SystemC model, the system designer *"can assume that a method process will execute in its entirety without interruption"* [40]. This is convenient when

sharing variables among different threads (because mutual exclusive access to such variables in a critical region is implicitly guaranteed by the non-preemptive scheduling semantics). While this can aid significantly in avoiding bugs and race conditions if one is only concerned with modeling and simulation, such semantics are hard to verify and synthesize if the goal is to eventually reach an efficient implementation. Furthermore, sharing of plain variables also violates the overall system-level principle of separation of concerns where computation and communication are supposed to be clearly separated in modules and channels, respectively. The uninterrupted execution guaranteed by the SystemC LRM makes it hard to synthesize concurrent threads into a truly parallel implementation. This same problem also prevents an efficient implementation of a fully standards-compliant parallel multi-core simulator, which we are aiming for in this work. This particular problem of parallel simulation is actually addressed specifically in the SystemC LRM [40] , as follows:

> *"An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined [...]. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics."*

In short, complex inter-dependency analysis over all variables in the system model is a prerequisite to parallel multi-core simulation in SystemC.

3.2.2 Pre-emptive multi-threading in SpecC

In contrast to the cooperative scheduling mandated for SystemC models, multi-threading in the SpecC LRM [37] is (in Section 3.3) explicitly stated as *"preemptive execution"*. Thus, *"No atomicity is guaranteed for the execution of any portion of concurrent code"*. Consequently, shared variables need to be carefully protected for mutually exclusive access in critical regions.

Allowing preemptive execution of concurrent threads requires the system designer to plan and design the system model carefully with respect to all communication (in particular through shared variables!) and synchronization. However, that is exactly the main premise of clear separation of computation and communication in system-level models (which enables the reuse of existing components, such as IP). In other words, if communication and computation are already separated in a well-written system model, this also provides the framework for solving the critical section problem in the a preemptive multi-threading environment.

The SpecC Language Working Group realized this opportunity when defining the version 2.0 of the language in 2001 [98]. In fact, a new *"time interval formalism"* was developed that precisely defines the parallel execution semantics of concurrent threads in SpecC. Specifically, truly parallel execution (with preemption) is generally assumed in SpecC.

To then allow safe communication and synchronization, a single exception was defined for channels. The SpecC LRM [37] states (in Section 2.3.2(j)) that

> *"For each instance of a channel, the channel methods are mutually exclusive in their execution. Implicitly, each channel instance has a mutex associated with it that the calling thread acquires before and releases after the execution of any method of the channel instance."*

In other words, each SpecC channel instance implicitly acts as a monitor that automatically protects the shared variables for mutually exclusive access in the critical region of communication. Note that the SpecC semantics based on separation of computation and communication elegantly solve the problem of allowing truly parallel execution of computational parts in the design, as well as provide built-in protection of critical regions in channels for safe communication. This builds the basis for both synthesis of efficient concurrent hardware/software realizations as well as implementation of efficient parallel simulators. While the freely available SpecC reference simulator does not utilize this possibility, we exploit these semantics in our parallel multi-core simulator.

3.3 Synchronous Parallel Discrete Event Simulation

Synchronous parallel discrete event simulation (SPDES) allows multiple threads in the same cycle (delta- and time-cycle) to run concurrently during simulation.

The scheduler for multi-core parallel simulation works the same way as the traditional scheduler, with one exception: in each cycle, it picks multiple OS kernel threads from the **READY** queue and runs them in parallel on the available cores. In particular, it fills the **RUN** set with multiple threads up to the number of CPU cores available. In other words, it keeps as many cores as busy as possible.

More specifically, when the **READY** queue is empty at the beginning of a scheduling step, the notified event list cannot be processed immediately since there could still be some other threads running which will trigger events sometime later. Thus, we introduce the **RUN** queue and check it first to see if any other threads are still busy doing their jobs. If so, the scheduler just sleep itself and wait for the running ones to wake it up in the future for further processing; if not, the scheduler handle those events and timing advancements in the same way as what the single-thread scheduler does. After the delat-cycle or timed-cycle advances, it goes to the multiple threads issuing branch if the **READY** queue is not empty at that time.

Fig. 3.2 shows the extended control flow of the multi-core scheduler. Note the extra loop at the left which issues threads as long as CPU cores are available and the **READY** queue is not empty. Moreover, instead of using user-level threads, OS kernel threads need to be used so that they can be scheduled and executed on different CPU cores.

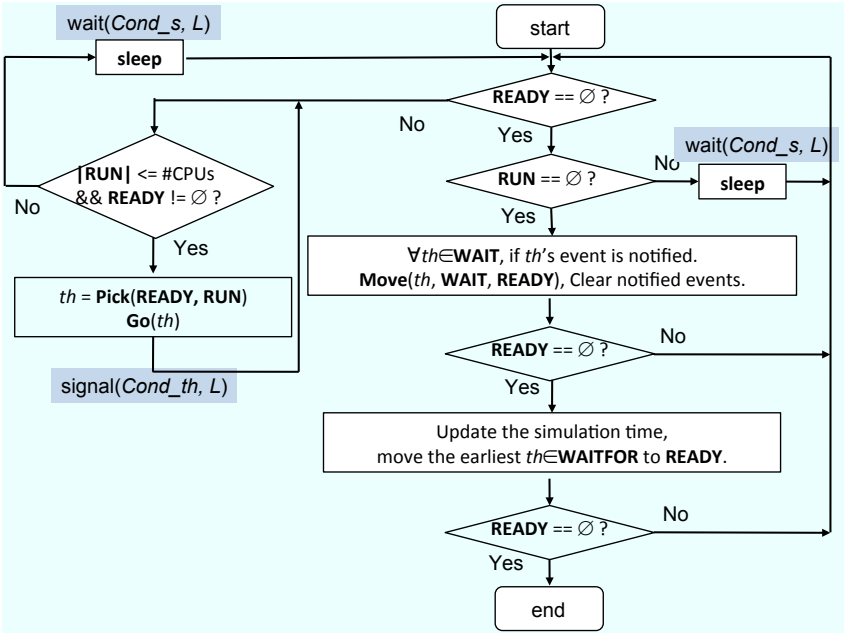


Figure 3.2: Synchronous parallel Discrete Event simulation scheduler

3.4 Synchronization for Multi-Core Parallel Simulation

The benefit of running more than a single thread at the same time comes at a price. Explicit synchronization becomes necessary. In particular, shared data structures in the simulation engine, including the thread queues and event lists in the scheduler, and shared variables in communication channels of the application need to be properly protected by locks for mutual exclusive accesses by the concurrent threads.

3.4.1 Protecting Scheduling Resources

To protect all central scheduling resources, we run the scheduler in its own thread and introduce locks and condition variables for proper synchronization. More specifically, we use

- one central lock L to protect the scheduling resources,
- a condition variable *Cond_s* for the scheduler, and

- a condition variable *Cond_th* for each working thread.

When a working thread executes a *wait* or *waitfor* instruction, we switch execution to the scheduling thread by waking the scheduler (signal(*Cond_s*)) and putting the working thread to sleep (wait(*Cond_th, L*)). The scheduler then uses the same mechanism to resume the next working thread.

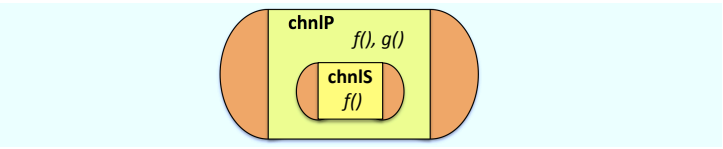
3.4.2 Protecting Communication

Communication between threads also needs to be explicitly protected as SLDL channels are defined to act as monitors [2]. That is, only one thread at a time may execute code wrapped in a specific channel instance.

[2]The SpecC Language Reference Manual V2.0 explicitly states in Section 2.3.2.j that channel methods are protected for mutual exclusive execution.

```
1 send(d)                          receive(d)
2 {                                {
3   Lock(this->chLock);              Lock(this->chLock);
4   while(n >= size){                while(!n){
5       ws ++;                           wr ++;
6       wait(eSend);                     wait(eRecv);
7       ws --;                           wr --;
8   }                                }
9   buffer.store(d);                 buffer.load(d);
10  if(wr){                          if(ws){
11      notify(eRecv);                   notify(eSend);
12  }                                }
13  unLock(this->chLock);            unLock(this->chLock);
14 }                               }
```

(a) Queue channel implementation for multi-core simulation.



(b) Example of user-defined hierarchical channels.

```
1 channel chnlS;
2 channel chnlP;
3
4 interface itfs
5 {
6   void f(chnlP* chp);
7   void g();
8 };
9
10 channel chnlP() implements itfs
11 {
12   chnlS chs;
13   void f(chnlP* chp){
14     chs.f(this);  //(a)inner channel function call
15     g();          //(b)call another member function
16   }
17   void g(){}
18 };
19
20 channel chnlS() implements itfs
21 {
22   void f(chnlP* chp){ chp->g();}
23   //(c)outer channel function call back
24 };
```

(c) SpecC description of the channels in (b).

Figure 3.3: Protecting synchronization in channels for multi-core parallel simulation

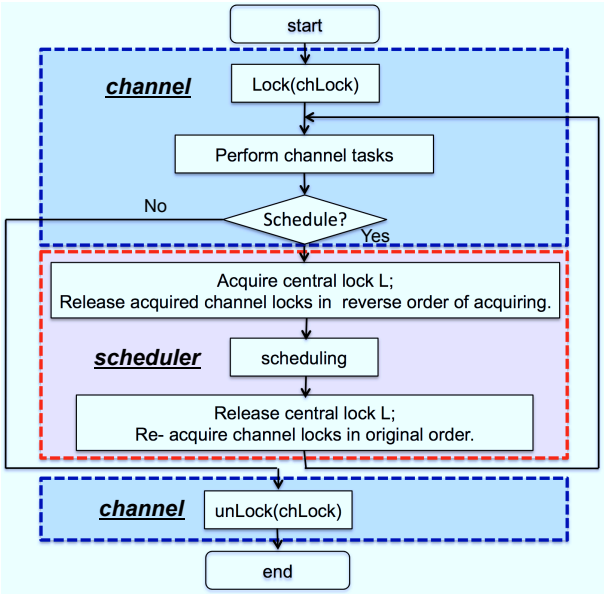
Since the sender and the receiver of a channel can be two separate threads, both can be running at the same time. Both of them can access the channel for data storing or loading. If the sender (receiver) should wait due to the fullness (emptiness) of the channel buffer, the thread can call the scheduler to pick up another one which is ready to go to avoid idling of the CPU. Thus, synchronization also needs to be considered for channels in the multi-core simulator.

To ensure this, we introduce a lock *ch→chLock* for each channel instance which is acquired at entry and released upon leaving any method of the channel. Fig. 3.3a shows this for the example of a simple circular buffer with fixed size.

3.4.3 Channel Locking Scheme

In general, however, channels can be more complex with multiple nested channels and methods calling each other. For such user-defined channels, we automatically generate the proper locking mechanism as follows. Fig. 3.3b shows an example of a channel *ChnlS* wrapped in another channel *ChnlP*. *ChnlP* has a method *ChnlP::f()* that calls another channel method *ChnlP::g()*. The inner channel *ChnlS* has a method *ChnlS::f()* that calls back the outer method *ChnlP::g()*.

In order to avoid duplicate lock acquiring and early releasing in channel methods, we introduce a counter *th→ch→lockCount* for each channel instance in each thread *th*, and a list *th→list* of acquired channel locks for each working thread. Instead of manipulating *ch→chLock* in every channel method, *th→ch→lockCount* is incremented at the entry and decremented before leaving the method, and *ch→chLock* is only acquired or released when *th→ch→lockCount* is *zero*. *ch→chLock* is added into *curTh→list* of the current thread when it is first acquired, and removed from the list when the last method of *ch* in the call stack exits. Note that, in order to avoid the possibility of dead locking, channel locks are stored in the order of acquiring. The working thread always releases them in the reverse order and keeps the original order when re-acquiring them. Fig. 3.4 illustrates the refined control flow in a channel method with proper locking scheme.




```
ch->lockLocal()
{
  if(curTh->ch->lockCount == 0){
    Lock(ch->chLock);
    curTh->list.Append(ch->chLock);
  }
  curTh->ch->lockCount ++;
}

ch->unlockLocal()
{
  curTh->ch->lockCount --;
  if(curTh->ch->lockCount == 0){
    curTh->list.Remove(ch->chLock);
    unLock(ch->chLock);
  }
}
```

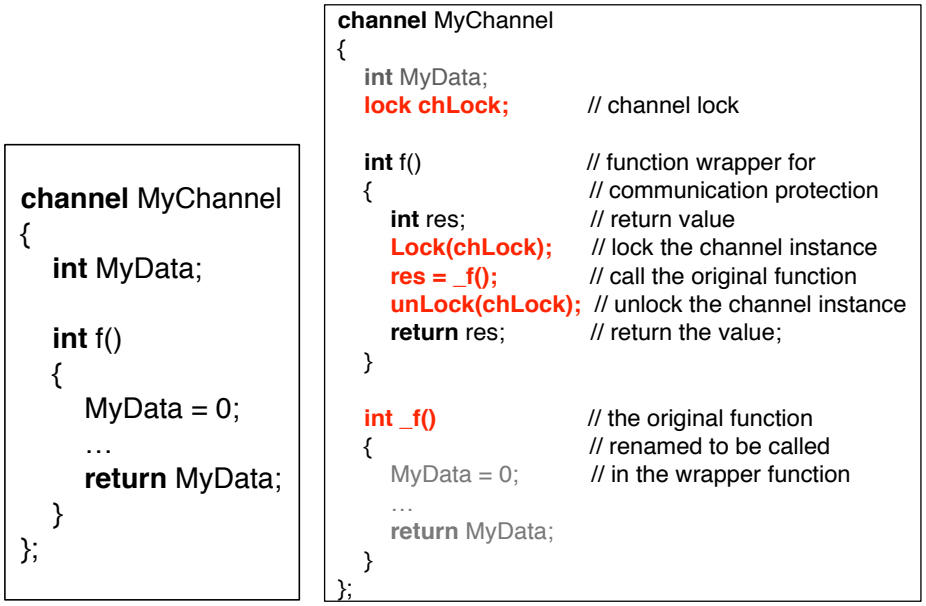
Figure 3.4: Synchronization protection for the member functions of communication channels.

3.4.4 Automatic Code Instrumentation for Communication Protection

SpecC channels serve as monitors for their member variables. Although the SpecC LRM defines this semantics for channels, channel locks are not added either in the original SpecC design nor in the SpecC compiler generated code (*.cc* and *.h* files in Fig. 1.8). The reason is that the reference simulator only allows one thread to be active at one time. Channel locks are not necessary in the reference sequential simulator.

The parallel simulator is highly desirable to be transparent for the designer to use. This means

the designer should be able to use the parallel simulator for fast simulation without modifying their models. Therefore, we extend the SpecC compiler to automatically instrument the SpecC code with channel locks. The instrumentation can be done by adding *lock* variable and wrapper functions in the *SIR* of the SpecC *channel*s. The SpecC code generator (translator) (Fig. 1.8) will then generate the proper C++ code which can be compiled into an executable image.



(a) Definition of a SpecC channel

(b) Definition of a instrumented SpecC channel

Figure 3.5: Channel definition instrumentation for communication protection

Fig. 3.5b shows an example for the channel instrumentation. First, a *lock* variable is added to the channel definition. Second, each function was renamed and be wrapped in an assist function between locking and unlocking the channel locks. The assist function has the original function name as well as the same parameter list and return type so that the design code can remain the same to call the protected channel functions.

The combination of a central scheduling lock and individual locks for channel instances ensures safe synchronization among many parallel working threads. Fig. 3.6 summarizes the detailed use of these locks and the thread switching mechanism for the life-cycle of a working thread.

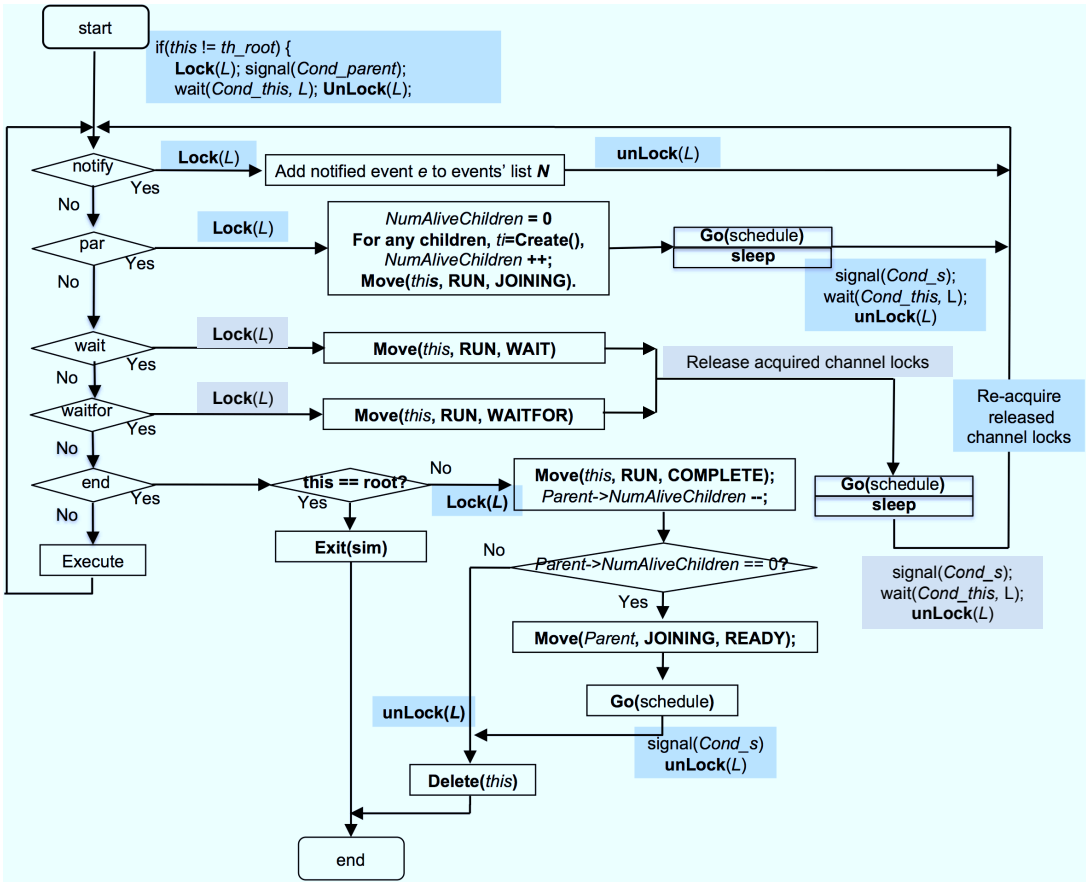


Figure 3.6: Life-cycle of a thread in the multi-core simulator

In Fig. 3.6, a working thread starts to execute its instructions and switch to the scheduler when there are instructions for simulation API calls, such as *notify*, *par*, *wait*, and *waitfor*. The thread will first acquire the central lock L for scheduling resources. If the lock acquiring is successful, then switch to the scheduler to perform scheduling tasks accordingly; otherwise, the thread will wait until the scheduling resource is available. The scheduling includes:

- creating child threads (*par*),
- setting notified event list (*notify*),
- suspending current working thread (*wait* or *waitfor*),
- updating cycles (after each scheduling loops as show in Fig. 3.2),

- clearing notified event list (at each scheduling cycles), etc.

If the current thread is suspended, it will release all the acquired channel locks before suspension so that the other threads can enter into the channel instances while the current thread is waiting for its trigger. When the thread is issued to run (be triggered by events or time advances), it will re-acquire all the channel locks to continue its execution (most likely in a channel instance).

3.5 Implementation Optimization for Multi-Core Simulation

As shown in Fig. 3.2 and Fig. 3.6, the threads in the simulator switch due to event handling and time advancement. Context switches occur when the owner thread of a CPU core is changed.

A dedicated scheduling thread (as discussed in Section 3.3 and used in Section 3.4) introduces context switch overhead when a working thread needs scheduling. This overhead can be eliminated by letting the current working thread perform the scheduling task itself. In other words, instead of using a dedicated scheduler thread, we define a function **schedule()** and call it in each working thread when scheduling is needed.

Then, the condition variable *Cond_s* is no longer needed. The "scheduler thread" (which is now the current working thread *curTh* itself) sleeps by waiting on *Cond_curTh*, i.e. the condition variable for the current working thread. In the left loop of Fig. 3.2, if the thread picked from the **READY** queue is the same as *curTh*, the **sleep** step is skipped and *curTh* continues. After the **sleep** step, the current working thread will continue its own work rather than entering into another scheduling iteration. In Fig. 3.6, **Go**(schedule) is replaced by the function call **schedule()** and the **sleep** step is no longer needed. Fig. 3.7 shows the refined multi-core scheduler with this optimization applied.

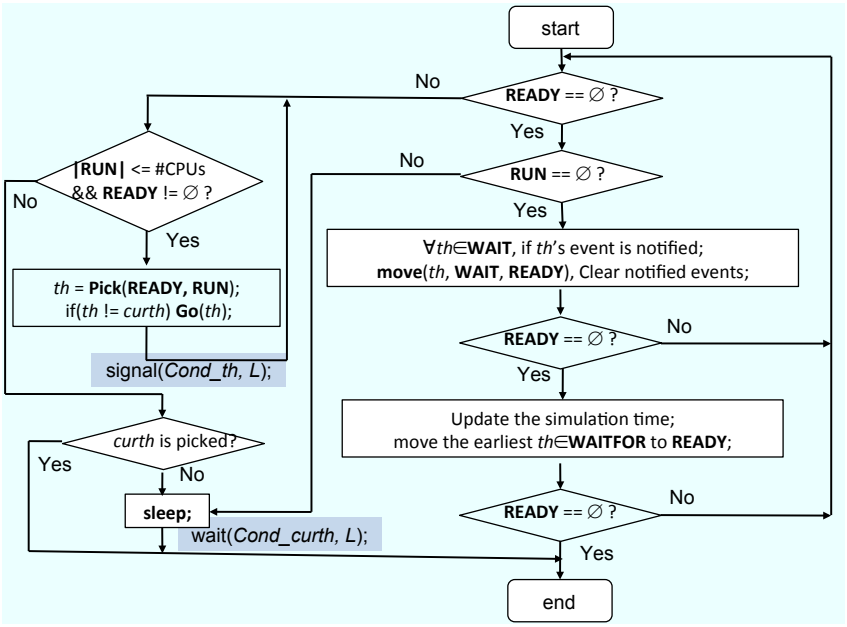


Figure 3.7: Optimized synchronous parallel discrete event simulation scheduler

Our experiments in next section show that this optimization achieves about 7.7% reduction in simulation time.

3.6 Experiments and Results

To demonstrate the improved simulation run time of our multi-core simulator, we use a H.264 video decoder and a JPEG encoder application.

3.6.1 Case Study on a H.264 Video Decoder

The H.264 AVC standard [99] is widely used in video applications, such as internet streaming, disc storage, and television services. H.264 AVC provides high-quality video at less than half the bit rate compared to its predecessors H.263 and H.262. At the same time, it requires more computing resources for both video encoding and decoding. In order to implement the standard

on resource-limited embedded systems, it is highly desirable to exploit available parallelism in its algorithm.

The H.264 decoder takes as input a video stream consisting of a sequence of encoded video frames. A frame can be further split into one or more slices during H.264 encoding, as illustrated in the upper right part of Fig. 3.8. Notably, slices are *independent* of each other in the sense that decoding one slice will not require any data from the other slices (though it may need data from previously decoded reference frames). For this reason, parallelism exists at the slice-level and parallel slice decoders can be used to decode multiple slices in a frame simultaneously.

H.264 Decoder Model with Parallel Slice Decoding

We have specified a H.264 decoder model based on the H.264/AVC JM reference software [100]. In the reference code, a global data structure (*img*) is used to store the input stream and all intermediate data during decoding. In order to parallelize the slice decoding, we have duplicated this data structure and other global variables so that each slice decoder has its own copy of input stream data and can decode its own slice locally. As an exception, the output of each slice decoder is still written to a global data structure (*dec_picture*). This is valid because the macro-blocks produced by different slice decoders do not overlap.

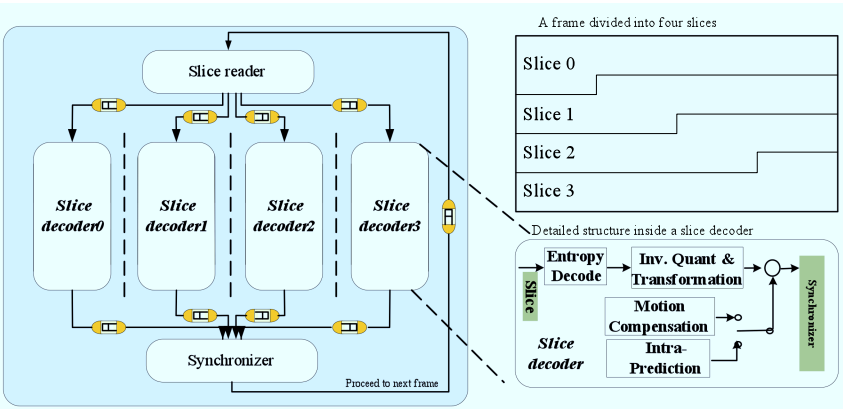


Figure 3.8: A parallelized H.264 decoder model.

Fig. 3.8 shows the block diagram of our model. The decoding of a frame begins with reading new slices from the input stream. These are then dispatched into four parallel slice decoders. Finally, a synchronizer block completes the decoding by applying a deblocking filter to the decoded frame. All the blocks communicate via FIFO channels. Internally, each slice decoder consists of the regular H.264 decoder functions, such as entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction.

Using SCE, we partition the above H.264 decoder model as follows: the four slice decoders are mapped onto four custom hardware units; the synchronizer is mapped onto an ARM7TDMI processor at 100MHz which also implements the overall control tasks and cooperation with the surrounding testbench. We choose Round-Robin scheduling for tasks in the processor and allocate an AMBA AHB for communication between the processor and the hardware units.

Experimental Results

For our experiment, we have prepared a test stream ("Harbour") of 299 video frames, each with 4 slices of equal size. Profiling the JM reference code with this stream showed that 68.4% of the total computation time is spent in the slice decoding, which we have parallelized in our decoder model.

As a reference point, we calculate the maximum possible performance gain as follows:

$$MaxSpeedup = \frac{1}{\frac{ParallelPart}{NumOfCores} + SerialPart}$$

For 4 parallel cores, the maximum speedup is

$$MaxSpeedup_4 = \frac{1}{\frac{0.684}{4} + (1 - 0.684)} = 2.05$$

The maximum speedup for 2 cores is accordingly $MaxSpeedup_2 = 1.52$.

Note that this H.264 decoder model has well balanced computation workload among the parallel slice decoders. Also the timing for the parallel sliced decoders are always the same, i.e. well

balanced timing.

Table 3.1: Simulation results of a well balanced H.264 Decoder ("Harbour", 299 frames 4 slices each, 30 fps).

Simulator Par. isd	Reference	Multi-Core						#delta cycles	#th
	n/a	1		2		4			
models	sim. time	sim. time	speed -up	sim. time	speed -up	sim. time	speed -up		
spec	20.80s (99%)	21.12s (99%)	**0.98**	14.57s (146%)	**1.43**	11.96s (193%)	**1.74**	76280	15
arch	21.27s (97%)	21.50s (97%)	**0.99**	14.90s (142%)	**1.43**	12.05s (188%)	**1.77**	76280	15
sched	21.43s (97%)	21.72s (97%)	**0.99**	15.26s (141%)	**1.40**	12.98s (182%)	**1.65**	82431	16
net	21.37s (97%)	21.49s (99%)	**0.99**	15.58s (138%)	**1.37**	13.04s (181%)	**1.64**	82713	16
tlm	21.64s (98%)	22.12s (98%)	**0.99**	16.06s (137%)	**1.35**	13.99s (175%)	**1.55**	115564	63
comm	26.32s (96%)	26.25s (97%)	**1.00**	19.50s (133%)	**1.35**	25.57s (138%)	**1.03**	205010	75
max speedup	1.00	1.00		1.52		2.05		n/a	n/a

Table 3.1 lists the simulation results for several TLMs generated with SCE when using our multi-core simulator on a Fedora Core 12 host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz, compiled with optimization (-O2) enabled. We compare the elapsed simulation run time against the single-core reference simulator (the table also includes the CPU load reported by the Linux OS). Although simulation performances decrease when issuing only one parallel thread due to additional mutexes for safe synchronization in each channel and the scheduler, our multi-core parallel simulation is very effective in reducing the simulation run time for all the models when multiple cores in the simulation host are used.

Table 3.1 also lists the measured speedup and the maximum theoretical speedup for the models that we have created following the SCE design flow. The more threads are issued in each scheduling step, the more speedup we gain. The *#delta cycles* column shows the total number of delta cycles executed when simulating each model. This number increases when the design is refined and is the reason why we gain less speedup at lower abstraction levels. More communication overhead is introduced and the increasing need for scheduling reduce the parallelism. However, the measured speedups are somewhat lower than the maximum, which is reasonable given the overhead introduced due to parallelizing and synchronizing the slice decoders. The comparatively lower performance gain for the *comm* model in simulation with 4 threads can be explained

due to the inefficient cache utilization in our Intel(R) Core(TM)2 Quad machine[3].

Table 3.2 compares the simulation run times of the non-optimized and the optimized implementation of the scheduler (Section 3.5). The number of the context switches in the optimized simulation reduces to about half and results in an average performance gain of about 7.8%.

Table 3.2: Comparison with optimized implementation.

Simulator		Regular		Optimized		
Parallel issued threads: 4		sim. time	#context switches	sim. time	#context switches	gain
models	spec	13.18s	161956	11.96s	81999	10%
	arch	13.62s	161943	12.05s	82000	13%
	sched	13.44s	175065	12.98s	85747	4%
	net	13.52s	178742	13.04s	88263	4%
	tlm	15.26s	292316	13.99s	140544	9%
	comm	27.41s	1222183	25.57s	777114	7%

Table 3.3: Simulation speedup with different h264 streams (spec model).

Simulator		Reference	Multi-Core		
Parallel issued threads:		n/a	1	2	4
slices frame	1	1.00	0.98	0.98	0.95
	2	1.00	0.98	1.40	1.35
	3	1.00	0.99	1.26	1.72
	4	1.00	0.98	1.43	1.74
	5	1.00	0.99	1.27	1.53
	6	1.00	0.99	1.41	1.68
	7	1.00	0.98	1.30	1.55
	8	1.00	0.98	1.39	1.59

Using a video stream with 4 slices in each frame is ideal for our model with 4 hardware decoders. However, we even achieve simulation speedup for less ideal cases. Table 3.3 shows the results when the test stream contains different number of slices. We also create a test stream file with 4 slices per frame but the size of the slices are imbalanced (percentage of MBs in each slice is 31%, 31%, 31%, 7%). Here, the speedup of our multi-core simulator versus the reference one is

[3]The Intel(R) Core(TM)2 Quad implements a two-pairs-of-two-cores architecture and Intel Advanced Smart Cache technology for each core pair (http://www.intel.com/products/processor/core2quad/prod_brief.pdf)

0.98 for issuing 1 thread, 1.28 for 2 threads, and 1.58 for 4 threads. As expected, the speedup decreases when available parallel work load is imbalanced.

3.6.2 Case Study on a JPEG Encoder

As an example of another application, Table 3.4 shows the simulation speedup for a JPEG Encoder example [101] which performs the DCT, $Quantization$ and $Zigzag$ modules for the 3 color components in parallel, followed by a sequential $Huffman$ encoder at the end (Fig. 3.9). Significant speedup is gained by our multi-core parallel simulator for the higher level models (*spec*, *arch*, *sched*). Simulation performance decreases for the models at the lower abstraction level (*net*) due to high number of bus transactions and arbitrations which are not parallelized and introduce large overhead due to the necessary synchronization protection.

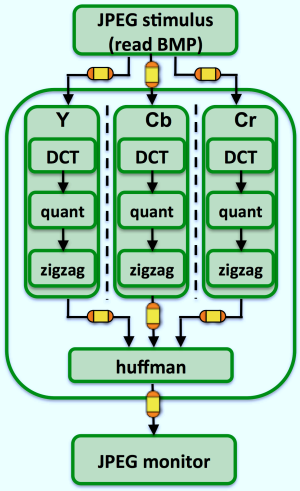


Figure 3.9: Parallelized JPEG encoder model.

Table 3.4: Simulation results of JPEG Encoder

Sim.	Ref.	Multi-Core					
Par. isd	n/a	1 thread		2 threads		4 threads	
models	sim time	sim time	speedup	sim time	speedup	sim time	speedup
spec	2.21s (99%)	2.15s (99%)	**1.03**	1.50s (143%)	**1.47**	1.40s (157%)	**1.58**
arch	2.22s (99%)	2.15s (99%)	**1.03**	1.50s (144%)	**1.48**	1.40s (158%)	**1.59**
sched	2.22s (99%)	2.17s (99%)	**1.02**	1.53s (143%)	**1.45**	1.41s (158%)	**1.57**
net	2.80s (99%)	2.86(99%)	**0.98**	2.38s (121%)	**1.18**	2.34s (123%)	**1.20**

4 Out-of-order Parallel Discrete Event Simulation

Both SystemC and SpecC SLDLs define DE-based execution semantics with "zero-delay" delta-cycles. Concurrent threads implement the parallelism in the design model, communicate via events and shared variables, and advance simulation time by use of *wait-for-time* primitives. Parallel execution of these threads is desirable to improve the simulation performance on multi-core hosts.

The PDES approach presented in Chapter 3 takes advantage of the fact that threads running at the same time and delta-cycle can execute in parallel. However, such *synchronous* PDES imposes a strict order on event delivery and time advance which makes delta- and time-cycles absolute barriers for thread scheduling.

In this chapter, we will propose an advanced PDES approach which aims at breaking the global simulation barrier to increase effectiveness of parallel simulation on multi-core computer hosts [102].

4.1 Motivation

We note that synchronous PDES issues threads strictly *in order*, with increasing time stamps after each cycle barrier. This can severely limit the desired parallel execution.

Specifically, when a thread finishes its execution cycle, it has to wait until all other active threads complete their execution for the same cycle. Only then the simulator advances to the next delta or time cycle. Additionally available CPU cores are idle until all threads have reached the cycle barrier.

As a motivating example, Fig. 4.1 shows a high-level model of a DVD player which decodes a stream of H.264 video and MP3 audio data using separate decoders. Since video and audio frames are data independent, the decoders run in parallel. Both decoders output the frames according to their rate, 30 FPS for video (delay 33.3ms) and 38.28 FPS for audio (delay 26.12ms).

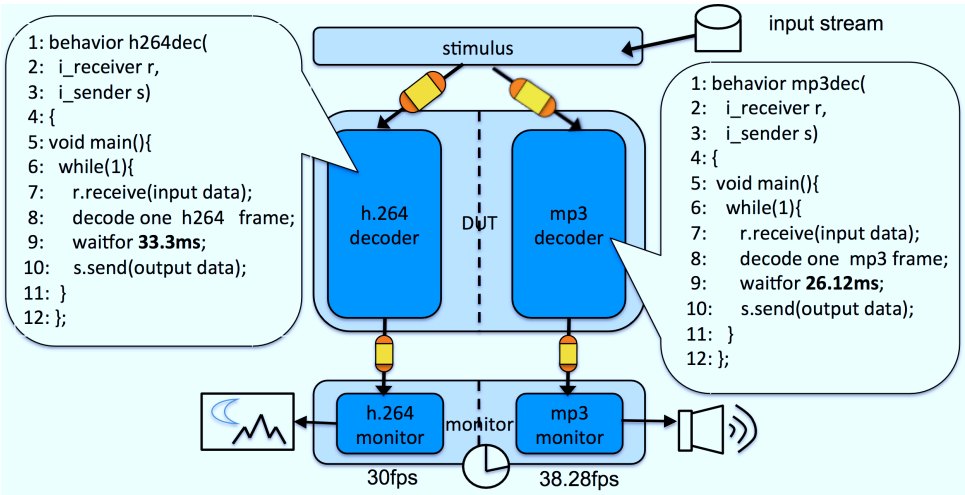
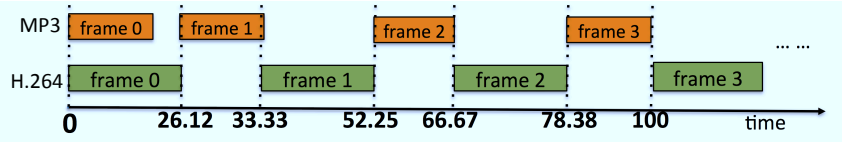


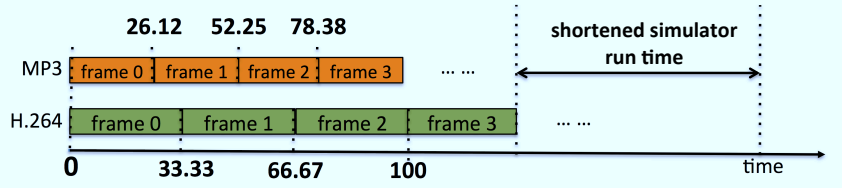
Figure 4.1: High-level DVD player model with video and audio streams.

Unfortunately, synchronous PDES cannot exploit the parallelism in this example. Fig. 4.2a shows the thread scheduling along the time line. Except for the first scheduling step, only one thread can run at a time. Note that it is not data dependency but the global timing that prevents parallel execution.

This observation motivates the idea of breaking simulation cycle barrier and let independent threads run *out-of-order* and concurrently so as to increase the simulation parallelism.



(a) Synchronous PDES schedule



(b) Out-of-order PDES schedule

Figure 4.2: Scheduling of the high-level DVD player model

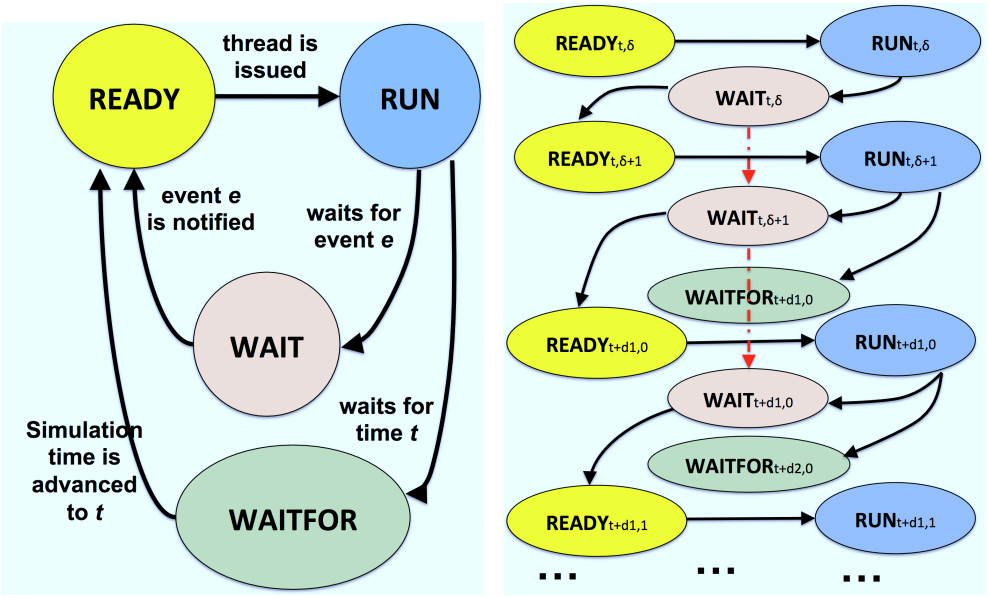
Fig. 4.2b shows the idea of out-of-order schedule for the DVD player example. The MP3 and H.264 decoders simulate in parallel on different cores and maintain their own time stamps. Since MP3 and H.264 are data independent, as a result, we can simulate the model correctly but significantly reduce the simulator run time.

4.2 Out-of-order Parallel Discrete Event Simulation

We now propose the new out-of-order simulation scheme where timing is only partially ordered in contrast to synchronous PDES which imposes a total order on event processing and time advances. We localize the global simulation time (*time*, *delta*) for each thread and allow threads without potential data or event conflicts to run ahead of time while other working threads are still running with earlier timestamps.

4.2.1 Notations

To formally describe the OoO PDES scheduling algorithm, we introduce the following notations:



(a) Static states in synchronous PDES (b) Dynamic states in out-of-order PDES

Figure 4.3: States and transitions of simulation threads (simplified).

1. We define simulation time as tuple (t, δ) where t = time, δ = delta-cycle. We order time stamps as follows:

   - **equal**: $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$
   - **before**: $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$
   - **after**: $(t_1, \delta_1) > (t_2, \delta_2)$, iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$

2. Each thread th has its own time (t_{th}, δ_{th}).

3. Since events can be notified multiple times and at different simulation times, we note an event e notified at (t, δ) as tuple (id_e, t_e, δ_e) and define: **EVENTS**$= \cup$**EVENTS**$_{t,\delta}$ where **EVENTS**$_{t,\delta} = \{(id_e, t_e, \delta_e) \mid t_e = t, \delta_e = \delta)\}$

4. For DE simulation, typically several sets of queued threads are defined, such as **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**}. These sets exist at all times and threads move from one to the other during simulation, as shown in Fig. 4.3a. For OoO PDES, we

define multiple sets with different time stamps, which we dynamically create and delete as needed, as illustrated in Fig. 4.3b.

Specifically, we define:

- **QUEUES** = {**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**}
- **READY** = ∪**READY**$_{t,\delta}$, **READY**$_{t,\delta}$={th | th is ready to run at (t, δ)}
- **RUN** = ∪**RUN**$_{t,\delta}$, **RUN**$_{t,\delta}$={th | th is running at (t, δ)}
- **WAIT** = ∪**WAIT**$_{t,\delta}$, **WAIT**$_{t,\delta}$={th | th is waiting since (t, δ) for events (id_e, t_e, δ_e), where $(t_e, \delta_e) \geq (t, \delta)$}
- **WAITFOR** = ∪**WAITFOR**$_{t,\delta}$, **WAITFOR**$_{t,\delta}$={th | th is waiting for simulation time advance to $(t, 0)$} Note that the delta cycle is always 0 for the **WAITFOR**$_{t,\delta}$ queues ($\delta = 0$)
- **JOINING** = ∪**JOINING**$_{t,\delta}$, **JOINING**$_{t,\delta}$={th | th created child threads at (t, δ), and waits for them to complete}
- **COMPLETE** = ∪**COMPLETE**$_{t,\delta}$, **COMPLETE**$_{t,\delta}$ = {th | th completed its execution at (t, δ)}

Note that for efficiency our implementation orders these sets by increasing time stamps.

5. Initial state at the beginning of simulation:
   - $t = 0$, $\delta = 0$
   - **RUN** = **RUN**$_{0,0}$ = {th_{root}}
   - **READY** = **WAIT** = **WAITFOR** = **COMPLETE** = **JOINING** = ∅
6. Simulation invariants:
   Let **THREADS** be the set of all existing threads. Then, at any time, the following conditions hold:
   - **THREADS** = **READY** ∪ **RUN** ∪ **WAIT** ∪ **WAITFOR** ∪ **JOINING** ∪ **COMPLETE**

- $\forall$ $\mathrm{A}_{t_1,\delta_1}$, $\mathrm{B}_{t_2,\delta_2} \in$ **QUEUES**:

  $A_{t_1,\delta_1} \neq B_{t_2,\delta_2} \Leftrightarrow A_{t_1,\delta_1} \cap B_{t_2,\delta_2} = \emptyset$

At any time, each thread belongs to exactly one set, and this set determines its state. Coordinated by the scheduler, threads change state by transitioning between the sets, as follows:

- $\mathbf{READY}_{t,\delta} \rightarrow \mathbf{RUN}_{t,\delta}$: thread becomes runable (is issued)
- $\mathbf{RUN}_{t,\delta} \rightarrow \mathbf{WAIT}_{t,\delta}$: thread calls **wait** for an event
- $\mathbf{RUN}_{t,\delta} \rightarrow \mathbf{WAITFOR}_{t',0}$, where $t < t' = t + delay$: thread calls **waitfor**($delay$)
- $\mathbf{WAIT}_{t,\delta} \rightarrow \mathbf{READY}_{t',\delta'}$, where $(t,\delta) < (t',\delta')$: the event that the thread is waiting for is notified; the thread becomes ready to run at (t',δ')
- $\mathbf{JOINING}_{t,\delta} \rightarrow \mathbf{READY}_{t',\delta'}$, where $(t,\delta) \leq (t',\delta')$: child threads completed and their parent becomes ready to run again
- $\mathbf{WAITFOR}_{t,\delta} \rightarrow \mathbf{READY}_{t,\delta}$, where $\delta = 0$: simulation time advances to $(t,0)$, making one or more threads ready to run

The thread and event sets evolve during simulation as illustrated in Fig. 4.3b. Whenever the sets $\mathbf{READY}_{t,\delta}$ and $\mathbf{RUN}_{t,\delta}$ are empty and there are no **WAIT** or **WAITFOR** queues with earlier timestamps, the scheduler deletes $\mathbf{READY}_{t,\delta}$ and $\mathbf{RUN}_{t,\delta}$, as well as any expired events with the same timestamp $\mathbf{EVENTS}_{t,\delta}$ (lines 8-14 in Algorithm 1).

4.2.2 Out-of-order PDES Scheduling Algorithm

Algorithm 1 defines the scheduling algorithm of our OoO PDES. At each scheduling step, the scheduler first evaluates notified events and wakes up corresponding threads in **WAIT**. If a thread becomes ready to run, its local time advances to $(t_e, \delta_e{+}1)$ where (t_e, δ_e) is the timestamp of the notified event (line 5 in Algorithm 1). After event handling, the scheduler cleans up any empty queues and expired events and issues qualified threads for the next delta-cycle (line 18).

Algorithm 1 Out-of-order PDES Algorithm

```
 1: /* trigger events */
 2: for all th ∈ WAIT do
 3:     if ∃ event (id_e, t_e, δ_e), th awaits e, and (t_e, δ_e) ≥ (t_th, δ_th) then
 4:         move th from WAIT_{t_th,δ_th} to READY_{t_e,δ_e+1}
 5:         t_th = t_e; δ_th = δ_e + 1
 6:     end if
 7: end for
 8: /* update simulation subsets */
 9: for all READY_{t,δ} and RUN_{t,δ} do
10:     if READY_{t,δ} = ∅ and RUN_{t,δ} = ∅ and WAITFOR_{t,δ} = ∅ then
11:         delete READY_{t,δ}, RUN_{t,δ}, WAITFOR_{t,δ}, EVENTS_{t,δ}
12:         merge WAIT_{t,δ} into WAIT_{next(t,δ)}; delete WAIT_{t,δ}
13:     end if
14: end for
15: /* issue qualified threads (delta cycle) */
16: for all th ∈ READY do
17:     if RUN.size < numCPUs and HasNoConflicts(th) then
18:         issue th
19:     end if
20: end for
21: /* handle wait-for-time threads */
22: for all th ∈ WAITFOR do
23:     move th from WAITFOR_{t_th,δ_th} to READY_{t_th,0}
24: end for
25: /* issue qualified threads (time advance cycle) */
26: for all th ∈ READY do
27:     if RUN.size < numCPUs and HasNoConflicts(th) then
28:         issue th
29:     end if
30: end for
31: /* if the scheduler hits this case, we have a deadlock */
32: if RUN = ∅ then
33:     report deadlock and exit
34: end if
```

Next, any threads in **WAITFOR** are moved to the **READY** queue corresponding to their wait time and issued for execution if qualified (line 28). Finally, if no threads can run (**RUN** $= \emptyset$), the simulator reports a deadlock and quits[1].

Note that the scheduling is aggressive. The scheduler issues threads for execution as long as idle CPU cores and threads without conflicts (**HasNoConflicts**(th)) are available.

Note also that we can easily turn on/off the parallel out-of-order execution at any time by setting

[1]The condition for a deadlock is the same as for a regular DE simulator.

the **numCPUs** variable. For example, when in-order execution is needed during debugging, we set **numCPUs** = 1 and the algorithm will behave the same as the traditional DE simulator where only one thread is running at all times.

4.3 Out-of-order PDES Conflict Analysis

In order to fully maintain the accuracy in simulation semantics and time, careful analysis on potential data and event dependencies and coordinating local time stamps for each thread are critical.

The OoO scheduler depends on conservative analysis of potential conflicts among the active threads. We now describe the threads and their position in their execution, and then present the conflict analysis and the which we separate into static compile-time and dynamic run-time checking.

4.3.1 Thread Segments and Segment Graph

At run time, threads switch back and forth between the states of **RUNNING** and **WAITING**. When **RUNNING**, they execute specific *segments* of their code. To formally describe our OoO PDES conflict analysis, we introduce the following definitions:

- **Segment** seg_i: source code statements executed by a thread between two scheduling steps

- **Segment Boundary** v_i: SLDL statements which call the scheduler, i.e. *wait*, *waitfor*, *par*, etc.

Note that segments seg_i and segment boundaries v_i form a directed graph where seg_i is the segment following the boundary v_i. Every node v_i starts its segment seg_i and is followed by other nodes starting their corresponding segments. We formally define:

- **Segment Graph (SG)**: **SG**=(**V**, **E**), where **V** = $\{v \mid v$ is a segment boundary$\}$, **E**=$\{e_{ij} \mid e_{ij}$ is the set of statements between v_i and v_j, where v_j could be reached from v_i, and $seg_i = \cup e_{ij}\}$.

```
1 #include <stdio.h>
2 int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3 behavior B(in int begin,// port variable
4            in int end,  // port variable
5            out int sum) // port variable
6 {
7   int i; event e;       // member variables
8   void main(){
9     int tmp;            // stack variable
10    tmp = 0;            // tmp(W)
11    i = begin;          // i(W), begin(R)
12    waitfor 2;          // segment boundary (waitfor)
13    while(i <= end){    // i(R), end(R)
14      notify e;         // notify event e
15      wait e;           // segment boundary (wait)
16      tmp += array[i];  // array(R), tmp(RW)
17      i ++;             // i(RW)
18    }
19    sum = tmp; }        // sum(W)
20 };
21
22 behavior Main()
23 {
24   int sum1, sum2;      // member variables
25   B b1(0, 4, sum1);    // behavior instantiation
26   B b2(5, 9, sum2);    // behavior instantiation
27   int main(){
28     par{
29          b1.main();    // segment boundary (par)
30          b2.main();
31     }                  // segment boundary (par_end)
32     printf("sum 1 is :%d \n", sum1); // sum1(R)
33     printf("sum 2 is :%d \n", sum2); // sum2(R) }
34 };
```

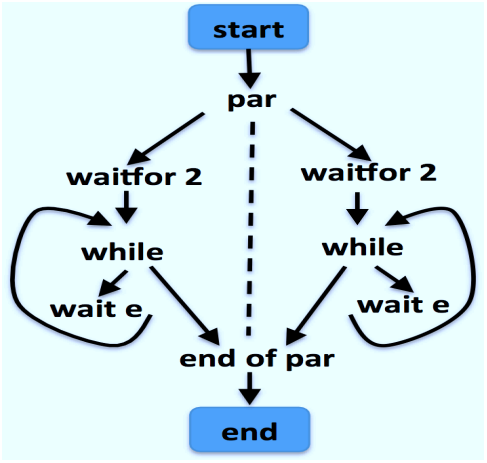
Figure 4.4: Example source code in SpecC

Fig. 4.4 shows a simple example written in SpecC SLDL. The design contains two parallel instances `b1` and `b2` of type `B`. Both `b1` and `b2` compute the sum of several elements stored in a global array `array`. The element indces' range is provided at the input ports `begin` and `end`, and the result is passed back to the parent via the output port `sum`. Finally, the `Main` behavior prints `sum1` of `b1` and `sum2` of `b2` to the screen.

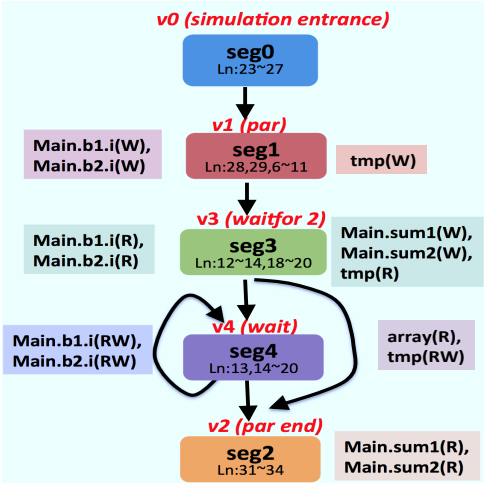
From the corresponding control flow graph in Fig. 4.5a, we derive the segment graph in Fig. 4.5b. The graph shows five segment nodes connected by edges indicating the possible flow between

the nodes. From the starting node *v0*, the control flow reaches the `par` statement (line 28) which is represented by the nodes *v1* (start) and *v2* (end) when the scheduler will be called. At *v1*, the simulator will create two threads for `b1` and `b2`. Since both are of the same type `B`, both will reach `waitfor 2` (line 12) via the same edge *v1*→*v3*. From there, the control flow reaches either `wait` (line 15) via *v3*→*v4*, or will skip the `while` loop (line 13) and complete the thread execution at the end of the `par` statement via *v3*→*v2*. From within the `while` loop, control either loops around *v4*→*v4*, or ends the loop and the thread via *v4*→*v2*. Finally, after *v2* the execution terminates.

Note that a code statement can be part of multiple segments. For example, line 19 belongs to both seg_3 and seg_4.



(a) Control Flow Graph (CFG)



(b) Segment Graph (SG)

seg	0	1	2	3	4
0	F	F	F	F	F
1	F	T	F	T	T
2	F	F	F	T	T
3	F	T	T	T	T
4	F	T	T	T	T

(c) Data conflict table (CTab)

seg	0	1	2	3	4
0	F	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F
3	F	F	F	F	T
4	F	F	F	F	F

(d) Event notification table (NTab)

segment	0	1	2	3	4
Current Time Advance	N/A	(0:0)	(0:0)	(2:0)	(0:1)

(e) Time advance table for the current segment (CTime)

segment	0	1	2	3	4
Next Time Advance	(0:0)	(2:0)	(∞:0)	(0:0)	(0:0)

(f) Time advance table for the next segment (NTime)

Figure 4.5: The control flow graph, segment graph, conflict tables, and time advance tables for the simple design example in Fig. 4.4

Note also that threads execute one segment in each scheduling step, and multiple threads may execute the same segments. [2]

[2]This is different from the segments in Section 5.1.2 where the "instance isolation" technique prevents multiple instances from sharing the same segment.

Algorithm 2 Build the Segment Graph

```
1: newSegList = BuildSegmentGraph(currSegList, stmnt)
2: {
3: switch (stmnt.type) do
4: case STMNT_COMPOUND:
5:   newL = currSegList
6:   for all subStmnt ∈ Stmnt do
7:     newL = BuildSegmentGraph(newL, subStmnt)
8:   end for
9: case STMNT_IF_ELSE:
10:  ExtendAccess(stmnt.conditionVar, currSegList) // collect segment variable access information
11:  tmp1 = BuildSegmentGraph(currSegList, subIfStmnt)
12:  tmp2 = BuildSegmentGraph(currSegList, subElseStmnt)
13:  newL = tmp1 ⋃ tmp2
14: case STMNT_WHILE:
15:  ExtendAccess(stmnt.conditionVar, currSegList) // collect segment variable access information
16:  helperSeg = new Segment
17:  tmpL = new SegmentList; tmpL.add(helperSeg)
18:  tmp1 = BuildSegmentGraph(tmpL, subWhileStmnt)
19:  if helperSeg ∈ tmp1
20:    then remove helperSeg from tmp1 end if
21:  for all Segment s ∈ tmp1⋃ currSegList do
22:    s.nextSegments ⋃= helperSeg.nextSegments
23:  end for
24:  newL = currSegList ⋃ tmp1; delete helperSeg
25: case STMNT_PAR:
26:  newSeg = new Segment; totalSegments ++
27:  newEndSeg = new Segment; totalSegments ++
28:  for all Segment s ∈ currSegList do
29:    s.nextSegments.add(newSeg) end for
30:  tmpL = new SegmentList; tmpL.add(newSeg)
31:  for all subStmnt ∈ stmnt do
32:    BuildSegmentGraph(tmpL, subStmnt)
33:    for all Segment s ∈ tmpL do
34:      s.nextSegments.add(newEndSeg) end for
35:  end for
36:  newL= new SegmentList; newL.add(newEndSeg)
37: case STMNT_WAIT:
38: case STMNT_WAITFOR:
39:  newSeg = new Segment; totalSegments++
40:  for all Segment s ∈ currSegList do
41:    s.nextSegments.add(newSeg); end for
42:  newL = new SegmentList; newL.add(newSeg)
43: case STMNT_EXPRESSION:
44:  if stmnt is a function call f() then
45:    newL = BuildFunctionSegmentGraph(currSegList, fct)
46:  else
47:    ExtendAccess(stmnt.expression, currSegList) //collect segment variable access information
48:    newL = currSegList
49:  end if
50: case ...: /* other statements omitted for brevity */
51: end switch
52: return newL;
53: }
```

The Segment Graph can be derived from the Control Flow Graph (CFG) of the SLDL model. Algorithm 2 shows the algorithm for Segment Graph building. Overall, the compiler traverses the application's control flow graph following all branches, function calls, and thread creation points, and recursively builds the corresponding segment graph.

Algorithm 2 shows the segment building function, *BuildSegmentGraph*, which we perform recursively on each of the statements in the design. The input of the function is the statement and a list of input segments ($currSegList$). For instance, when analyzing first statement of the design, we have the initial segment (seg_0) in $currSegList$. The output of the function is also a list of segments which we will use as the input to build the next statement.

In the *BuildSegmentGraph* function, we handle the statements of different types accordingly. For segment boundary statements, such as STMNT_PAR, STMNT_WAIT, STMNT_WAITFOR, ect., we create a new segment node and add it as the next segment to all the input segments. This helps to connect the segments in the segment graph.

For control flow statements, such as STMNT_IF_ELSE, STMNT_WHILE, etc., we conservatively follow all the branches so as to have a complete coverage. For example, we analyze both the *if* and *else* branches of a STMNT_IF_ELSE statement, and merge both of the branches' output segment lists into one as the output of the STMNT_IF_ELSE statement. Note that a *helper* segment is used for STMNT_WHILE statements to connect the segments properly due to the loops (i.e. $v4 \rightarrow v4$, Fig. 4.5b).

The *ExtendAccess* function in Algorithm 2 is used to collect the variable access information from the statement.

4.3.2 Static Conflict Analysis

To comply with SLDL execution semantics, threads must not execute in parallel or out-of-order if the segments they are in pose any hazard towards validity of data, event delivery, or

timing.

Data Hazards

Data hazards are caused by parallel or out-of-order accesses to shared variables. Three cases exist, namely read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW).

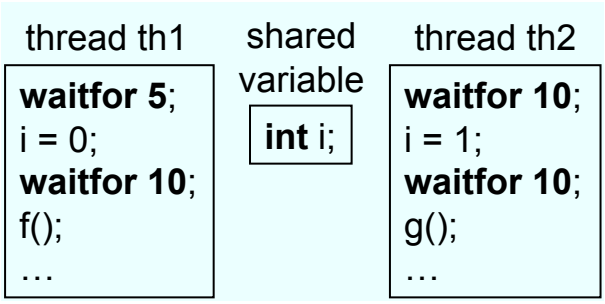


Figure 4.6: Write-after-write (WAW) conflict between two parallel threads.

Fig. 4.6 shows a simple example of a WAW conflict where two parallel threads th_1 and th_2 write to the same variable i at different times. Simulation semantics require that th_1 executes first and sets i to 0 at time $(5, 0)$, followed by th_2 setting i to its final value 1 at time $(10, 0)$. If the simulator would issue the threads th_1 and th_2 in parallel, this would create a race condition, making the final value of i non-deterministic. Thus, we must not schedule th_1 and th_2 out-of-order. Note, however, that th_1 and th_2 can run in parallel after their second wait-for-time statement if the functions $f()$ and $g()$ are independent.

Since data hazards stem from the code in specific segments, we analyze data conflicts statically at compile-time and create a table where the scheduler can then at run-time quickly lookup any potential conflicts between active segments.

We define a **data conflict table** $CTab[N, N]$ where N is the total number of segments in the application code: $CTab[i, j] = \mathit{true}$, iff there is a potential data conflict between the segments s_i and s_j; otherwise, $CTab[i, j] = \mathit{false}$.

To build the conflict table, we compile for each segment a **variable access list** which contains all variables accessed in the segment. Each entry is a tuple *(Symbol, AccessType)* where *Symbol* is the variable and *AccessType* specifies read-only (R), write-only (W), read-write (RW), or pointer access (Ptr).

Finally, we create the conflict table $CTab[N, N]$ by comparing the access lists for each segment pair. If two segments s_i and s_j share any variable with access type (W) or (RW), or there is any shared pointer access by s_i or s_j, then we mark this as a potential conflict.

Fig. 4.5c shows the data conflict table for the example in Fig. 4.4. Here, for instance, seg_4 has a data conflict (RAW) with seg_1 since seg_4 has `i`(RW) (line 17) and seg_1 has `i`(W) (line 11) in the variable access list.

Not all variables are straightforward to analyze, however. The SLDL actually supports variables at different scopes as well as ports which are connected by port maps in the structural hierarchy of the design model.

We distinguish and handle the following cases:

- **Global variables**, e.g. `array` in line 2: This case is discussed above and can be handled directly as tuple *(Symbol, AccessType)*.
- **Local variables**, e.g. `tmp` in line 9: Local variables are stored on the function call stack and cannot be shared between different threads. Thus, they can be ignored in the access lists.
- **Instance member variables**, e.g. `i` in line 7: Since instances may be instantiated multiple times and then their variables are different, we need to distinguish them by their complete *instance path* prepended to the variable name. For example, the actual *Symbol* used for the two instances of variable `i` is `Main.b1.i` or `Main.b2.i`, respectively.
- **Port variables**, e.g. `sum` in line 5: For ports, we need to find the actual variable mapped

to the port and use that as *Symbol* together with its actual instance path. For example, `Main.sum1` is the actual variable mapped to port `Main.b1.sum`. Note that tracing ports to their connected variables requires the compiler to follow port mappings through the structural hierarchy of the design model which is possible when all used components are part of the current translation unit.

- **Pointers**: We currently do not perform pointer analysis (future work). For now, we conservatively mark all segments with pointer accesses as potential conflicts.

Event Hazards

Thread synchronization through event notification also poses hazards to out-of-order execution. Specifically, thread segments are dependent when one is waiting for an event notified by another.

We define an **event notification table** $NTab[N, N]$ where N is the total number of segments: $NTab[i, j] = true$, iff segment s_i notifies an event that s_j is waiting for; otherwise, $NTab[i, j] = false$. Note that in contrast to the data conflict table above, the event notification table is not symmetric.

Fig. 4.5d shows the event notification table for the simple example. For instance, $NTab[3, 4] = true$ since seg_3 notifies the event `e` (line 14) which seg_4 waits for (line 15).

Note that, in order to identify event instances, we use the same scope and port map handling for events as described above for data variables.

Timing Hazards

The local time for an individual thread in OoO PDES can pose a timing hazard when the thread runs too far ahead of others. To prevent this, we analyze the time advances of threads at segment

boundaries. There are three cases with different increments, as listed in Table 4.1.

Table 4.1: Time advances at segment boundaries.

Segment boundary	Time Increment	
wait event	increment by one delta cycle	(0:1)
waitfor t	increment by time t	(t:0)
par/par_end	no time increment	(0:0)

We define two time advance tables, one for the segment a thread is currently in, and one for the next segment(s) that a thread can reach in the following scheduling step.

The **current time advance table** $CTime[N]$ lists the time increment that a thread will experience when it enters the given segment. For the example in Fig. 4.5e for instance, the *waitfor* 2 (line 12) at the beginning of seg_3 sets $CTime[3] = (2:0)$.

The **next time advance table** $NTime[N]$ lists the time increment that a thread will incur when it leaves the given and enters the next segment. Since there may be more than one next segment, we list in the table the minimum of the time advances which is the earliest time the thread can become active again. Formally:
$NTime[i] = min\{CTime[j], \forall seg_j$ which follow $seg_i\}$.

For example, Fig. 4.5f lists $NTime[3]$ = (0:0) since seg_3 is followed by seg_4 (increment (0:1)) and seg_2 (increment (0:0).

Table 4.2: Examples for direct and indirect timing hazards

Situation	th_1	th_2	Hazard?
Direct Timing Hazard	(10:2)	(10:0), next segment at (10:1)	yes
	(10:2)	(10:0), next segment at (12:0)	no
Indirect Timing Hazard	(10:2)	(10:0), wakes th_3 at (10:1)	yes
	(10:2)	(10:1), wakes th_3 at (10:2)	no

There is two types of timing hazards, namely direct and indirect ones. For a candidate thread th_1 to be issued, a *direct timing hazard* exists when another thread th_2, that is safe to run, resumes its execution at a time earlier than the local time of th_1. In this case, the future of th_2

is unknown and could potentially affect th_1. Thus, it is not safe to issue th_1.

Table 4.2 shows an example where th_1 is considered for execution at time (10:2). If there is a thread th_2 with local time (10:0) whose next segment runs at time (10:1), i.e. before th_1, then the execution of th_1 is not safe. However, if we know from the time advance tables that th_2 will resume its execution later at (12:0), no timing hazard exists with respect to th_2.

An *indirect timing hazard* exists if a third thread th_3 can wake up earlier than th_1 due to an event notified by th_2. Again, Table 4.2 shows an example. If th_2 at time (10:0) potentially wakes a thread th_3 so that th_3 runs in the next delta cycle (10:1), i.e. earlier than th_1, then it is not safe to issue th_1.

4.3.3 Dynamic Conflict Detection

With the above analysis performed at compile time, the generated tables are passed to the simulator so that it can make quick and safe scheduling decisions at run time by using table lookups. Our compiler also instruments the design model such that the current segment ID is passed to the scheduler as an additional argument whenever a thread executes scheduling statements, such as *wait* and *wait-for-time*.

At run-time, the scheduler calls a function **HasNoConflicts**(th) to determine whether or not it can issue a thread th early. As shown in Algorithm 3, **HasNoConflicts**(th) checks for potential conflicts with all concurrent threads in the **RUN** and **READY** queues with an earlier time than th. For each concurrent thread, function **Conflict**(th, th_2) checks for any data, timing, and event hazards. Note that these checks can be performed in constant time (O(1)) due to the table lookups.

Algorithm 3 Conflict Detection in Scheduler

```
 1: bool HasNoConflicts(Thread th)
 2: {
 3:   for all th_2 ∈ RUN ∪ READY,
 4:   where (th_2.t, th_2.δ) < (th.t, th.δ) do
 5:     if (Conflict(th, th_2))
 6:       then return false end if
 7:   end for
 8:   return true
 9: }
10:
11: bool Conflict(Thread th, Thread th_2)
12: {
13:   if (th has data conflict with th_2) then /* data hazard */
14:     return true end if
15:   if (th_2 may enter another segment before th) then /* time hazard */
16:     return true end if
17:   if (th_2 may wake up another thread to run before th) then /* event hazard */
18:     return true end if
19:   return false
20: }
```

4.4 Experimental Results

We have implemented the proposed out-of-order parallel simulator in the SCE system design environment and conducted experiments on three multi-media applications shown.

To demonstrate the benefits of our out-of-order PDES, we compare the compiler and simulator run times with the traditional single-threaded reference and the synchronous parallel implementation Section 3.

All experiments have been performed on the same host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz.

4.4.1 An Abstract Model of a DVD Player

Our first experiment uses the DVD player model shown in Fig. 4.7. Similar to the model discussed in Section 4.1, a H.264 video and a MP3 audio stream are decoded in parallel. However, this model features four parallel slice decoders which decode separate slices in a H.264 frame simultaneously. Specifically, the H.264 stimulus reads new frames from the input stream and dispatches its slices to the four slice decoders. A synchronizer block completes the decoding of each frame and triggers the stimulus to send the next one. The blocks in the model communicate via double-handshake channels.

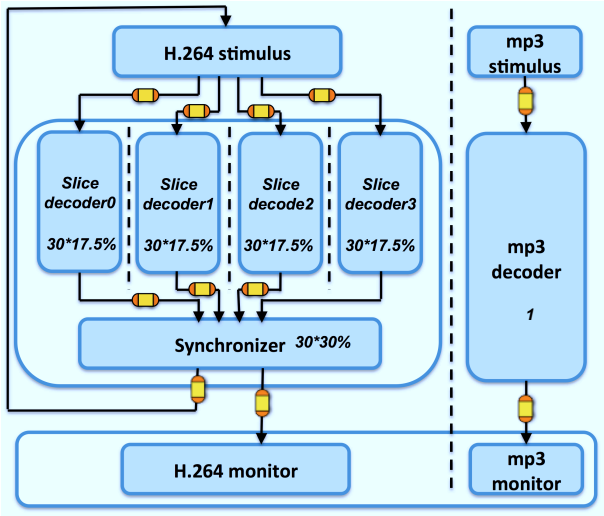


Figure 4.7: The abstract DVD player example

According to profiling results, the computational workload ratio between decoding one H.264 frame with 704x566 pixels and one 44.1kHz MP3 frame is about 30:1. Further, about 70% of the decoding time is spent in the slice decoders. The resulting workload of the major blocks is shown in the diagram.

Table 4.3 shows the statistics and measurements for this model. Note that the conflict table is very sparse, allowing 78.47% of the threads to be issued out-of-order. While the synchronous PDES loses performance due to in-order time barriers and synchronization overheads, our out-

of-order simulator shows twice the simulation speed.

Table 4.3: Experimental results for the abstract DVD player example

Simulator:	Single-thread reference	Multi-core	
		Synchronous PDES	Out-of-order PDES
compile time	0.71s	0.88s / 0.81	0.96s / **0.74**
simulator time	7.82s	7.98s / 0.98	3.89s / **2.01**
#segments N		50	
total conflicts in $CTab[N, N]$		160/2500 (6.4%)	
#threads issued		1008	
#threads issued out-of-order		**791 (78.47%)**	

4.4.2 A JPEG Encoder Model

Our second experiment uses the JPEG image encoder model described in Section 3.6.2 with timing annotations. Basically, the stimulus reads a BMP color image with 3216x2136 pixels and performs color-space conversion from RGB to YCbCr. The encoder performs the DCT, quantization and zigzag modules for the three independent color components (Y, Cb, Cr) in parallel, followed by a sequential Huffman encoder at the end. The JPEG monitor collects the encoded data and stores it in the output file.

To show the increased simulation speed also for models at different abstraction levels, we have created four models (*spec*, *arch*, *sched*, *net*) with increasing amount of implementation detail, down to a network model with detailed bus transactions. Table 4.4 lists the PDES statistics and shows that, for the JPEG encoder, about half or more of all threads can be issued out-of-order. Table 4.5 shows the corresponding compiler and simulator run times.

While the compile time increases similar to the synchronous parallel compiler, the simulation speed improves by about 138%, more than 5 times the gain of the synchronous parallel simulator.

Table 4.4: Out-of-order PDES statistics for JPEG and H.264 examples

JPEG Image Encoder				
TLM abstraction:	spec	arch	sched	net
#segments N	42	40	42	43
total conflicts in $CTab[N,N]$	199/1764 (11.3%)	184/1600 (11.5%)	199/1764 (11.3%)	212/1849 (11.5%)
#threads issued	271	268	268	4861
#threads issued out-of-order	**176 (64.9%)**	**173 (64.5%)**	**176 (65.7%)**	**2310 (47.5%)**
H.264 Video Decoder				
TLM abstraction:	spec	arch	sched	net
#segments N	67	67	69	70
total conflicts in $CTab[N,N]$	512/4489 (11.4%)	518/4489 (11.5%)	518/4761 (10.88%)	518/4900 (10.57%)
#threads issued	923017	921732	937329	1151318
#threads issued out-of-order	**179581 (19.46%)**	**176500 (19.15%)**	**177276 (18.91%)**	**317591 (27.58%)**

4.4.3 A Detailed H.264 Decoder Model

Our third experiment simulates the detailed H.264 decoder model described in Section 3.6.1. While this model is similar at the highest level to the video part of the abstract DVD player in Fig. 4.1 , it contains many more blocks at lower levels which implement the complete H.264 reference application. Internally, each slice decoder consists of complex H.264 decoder functions entropy decoding, inverse quantization and transformation, motion compensation, and intra-prediction. For our simulation, we use a video stream of 1079 frames and 1280x720 pixels per frame, each with 4 slices of equal size.

Note that although this H.264 decoder has the same structure as the one in Section 3.6.1, the double handshake channels are used in this model to reflect the options in the real design. Thus, the logic timing for each parallel slice decoder is actually imbalanced in the model.

Our simulation results for this industrial-size design are listed in the lower half of Table 4.4 and Table 4.5, again for four models at different abstraction levels, including a network model with detailed bus transactions. Due to the large complexity of the models, the compile time increases

by up to 35.2% This, however, is insignificant when compared to the much longer simulator run times.

While the synchronous parallel simulator shows almost no improvement in simulation speed, our proposed simulator shows more than 60% gain since many of the threads can be issued out-of-order (see Table 4.4).

Table 4.5: Experimental results for the JPEG Image Encoder and the H.264 Video Decoder examples

Simulator:		Single-thread reference		Multi-core			
				Synchronous parallel		Out-of-Order parallel	
		compile time [sec]	simulator time [sec]	compile time [sec] / speedup	simulator time [sec] / speedup	compile time [sec] / speedup	simulator time [sec] / speedup
JPEG Encoder	spec	0.80	2.23	1.10 / 0.73	1.84 / 1.21	1.13 / 0.71	0.93 / **2.40**
	arch	1.09	2.23	1.35 / 0.81	1.80 / 1.24	1.37 / 0.80	0.93 / **2.40**
	sched	1.14	2.24	1.41 / 0.81	1.83 / 1.22	1.43 / 0.80	0.92 / **2.43**
	net	1.34	2.90	1.59 / 0.84	2.33 / 1.24	1.63 / 0.82	1.26 / **2.30**
H.264 Decoder	spec	12.35	97.16	13.91 / 0.89	97.33 / 1.00	18.13 / 0.68	60.33 / **1.61**
	arch	11.97	97.81	12.72 / 0.94	99.93 / 0.98	18.46 / 0.65	60.77 / **1.61**
	sched	18.18	100.20	18.84 / 0.96	100.18 / 1.00	24.80 / 0.73	60.96 / **1.64**
	net	18.57	111.07	19.52 / 0.95	106.14 / 1.05	26.06 / 0.71	66.25 / **1.68**

5 Optimized Out-of-order Parallel Discrete Event Simulation

The out-of-order parallel discrete event simulation in Chapter 4 relies on static code analysis and dynamic out-of-order scheduling to break the global cycle barrier in simulation so as to achieve high multi-core CPU utilization. It is a conservative approach which only issues threads out of the order when it is safe based on the conflict analysis information, and thus avoids the expensive overhead for simulation roll-back. However, static code analysis can generate false conflicts which will in turn prevent the dynamic scheduler from making aggressive thread scheduling decisions.

In this chapter, we will present two optimizations on the out-of-order parallel discrete event simulation framework to reduce false conflicts. The first one is an optimized compiler which uses the idea of instance isolation to reduce false conflicts generated by the static code analysis [95]. It is also more efficient than the straight forward analysis approach in Section 4.3. The second one is an optimized scheduling approach which reduces false conflict detection at run time with the help of prediction information [103].

5.1 Optimized Compiler Using Instance Isolation

Out-of-order parallel simulation relies heavily on the conflict results generated by the static code analyzer. The quality of the static analysis has a significant impact on the correctness and efficiency of the out-of-order parallel simulation.

5.1.1 Motivation

To be safe, the static analysis sometimes is overly conservative and generates *false conflicts* which prevent the out-of-order scheduler from issuing threads in parallel that do not pose any real hazard. For example, in Fig. 5.1 both *seg3* and *seg4* contain write (W) accesses to variables `Main.sum1` and `Main.sum2`. Consequently, the analysis reports a conflict between *seg3* and *seg4*. In turn, the thread th_{b2} cannot execute *seg4* in parallel with thread th_{b1} in *seg3* (or vice versa). In reality, however, the execution is safe when th_{b1} and th_{b2} are in *seg3* and *seg4* at the same time since instance `b1` will only access `Main.sum1` and `b2` only modifies `Main.sum2`. Thus, the conflict does not really exist.

Looking closer at this *false conflict*, we observe that the compiler cannot tell that the access of `Main.sum1` in *seg3* and *seg4* will only happen in thread th_{b1} but not in th_{b2}, because both `b1` and `b2` share the same definition (and same segments/code). Following this, we can resolve the false conflict if `b1` and `b2` have separate definitions, such as `B_iso0` and `B_iso1` in Fig. 5.2a. Here, two different segments, *seg3* and *seg5*, start from `waitfor 1` in instances `B_iso0` and `B_iso1`, respectively. Now *seg3* only writes `Main.sum1` and *seg5* only writes `Main.sum2`. Consequently, the scheduler detects that it is safe for th_{b1} to execute *seg3* in parallel to th_{b2} in *seg5*. The same argument holds for *seg4* and *seg6*, as indicated in the extended *CTable* in Fig. 5.2c.

In general, the fewer conflicts are detected by the analysis, the more threads can be issued in parallel by the out-of-order scheduler, and the higher the simulation speed will be. On the other

```
1 #include <stdio.h>
2 int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3 behavior B(in int begin,// port variable
4            in int end,  // port variable
5            out int sum) // port variable
6 {
7   int i; event e;       // member variables
8   void main(){
9     int tmp;            // stack variable
10    tmp = 0;            // tmp(W)
11    i = begin;          // i(W), begin(R)
12    waitfor 2;          // segment boundary (waitfor)
13    while(i <= end){    // i(R), end(R)
14      notify e;         // notify event e
15      wait e;           // segment boundary (wait)
16      tmp += array[i];  // array(R), tmp(RW)
17      i ++;             // i(RW)
18    }
19    sum = tmp; }        // sum(W)
20 };
21
22 behavior Main()
23 {
24   int sum1, sum2;      // member variables
25   B b1(0, 4, sum1);    // behavior instantiation
26   B b2(5, 9, sum2);    // behavior instantiation
27   int main(){
28     par{
29          b1.main();    // segment boundary (par)
30          b2.main();
31     }                  // segment boundary (par_end)
32     printf("sum 1 is :%d \n", sum1); // sum1(R)
33     printf("sum 2 is :%d \n", sum2); // sum2(R) }
34 };
```

Figure 5.1: Example source code in SpecC

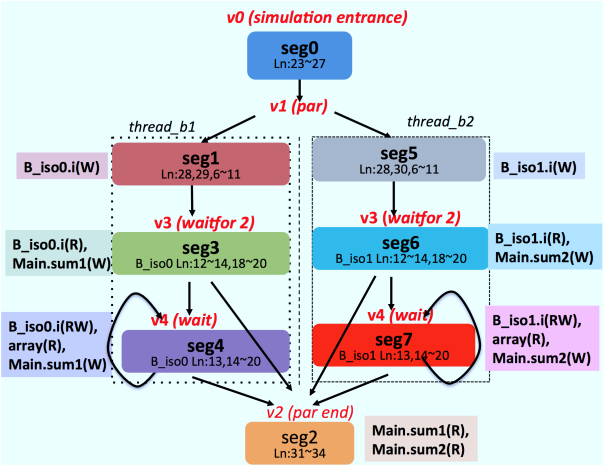
hand, if the code analyzer reports conflict between all the segments, the out-of-order simulator will downgrade to a regular in-order simulator.

```
 3 behavior B_iso0(...)      behavior B_iso1(...)
 4 {                         {
   // same body as B          // same body as B
20 };                        };
21
22 behavior Main()
23 {
24   int sum1, sum2;
25   B_iso0 b1(0, 4, sum1);
26   B_iso1 b2(5, 9, sum2);
27   int main(){
     ...
34 };
```

(a) Source code with isolated instances

seg	0	1	2	3	4	5	6	7
0	F	F	F	F	F	F	F	F
1	F	T	F	T	T	F	F	F
2	F	F	F	T	T	F	T	T
3	F	T	T	T	T	F	F	F
4	F	T	T	T	T	F	F	F
5	F	F	F	F	F	T	T	T
6	F	F	T	F	F	T	T	T
7	F	F	T	F	F	T	T	T

(b) Conflict table after isolation



(c) Segment graph after isolation

Figure 5.2: Simple design example with isolated instances.

Definition: We introduce the term **Instance Isolation** to describe the source code modification demonstrated above. *Isolating* an instance means to create a unique copy of the instance definition (behavior or channel) so that the instance has its own segments.

Table 5.1: Experimental Results for H.264 Decoder Models with Different Degree of Isolation.

Model	#bhvr	#chnl	formatted lines of code	Synchronous PDES		Out-of-Order PDES		#seg	#conflicts	#total issues	#OoO issues
				cmpl [sec]	sim [sec]	cmpl [sec] (speedup)	sim [sec] (speedup)				
iso0	54	11	55258	11.86	99.42	18.76 (0.63)	96.97 **(1.03)**	38	322 (22.3%)	927583	152816 (16.5%)
iso1	54	13	55330	11.93	101.01	17.57 (0.68)	96.49 **(1.05)**	41	336 (20.0%)	933222	146711 (15.7%)
iso2	58	16	55558	12.07	99.25	17.74 (0.68)	83.20 **(1.19)**	48	388 (16.8%)	913225	147541 (16.2%)
iso3	62	19	55786	12.23	99.68	17.87 (0.68)	72.72 **(1.37)**	55	440 (14.6%)	920001	166065 (18.1%)
iso4	66	24	56160	12.46	100.95	18.13 (0.69)	60.33 **(1.67)**	67	512 (11.4%)	923017	179581 (19.5%)

We will now show that instance isolation has a major impact on out-of-order PDES. Using

the example of a H.264 video decoder (Fig. 3.6.1[1]) with a test stream of 1079 video frames (1280x720 pixels per frame), Table 5.1 shows the impact of instance isolation with respect of model and code size, as well as compilation and simulation run time.

We have manually created 5 design models with an increasing number of isolated instances. *iso0* is the initial model without instance isolation, *iso1*, *iso2* and *iso3* are partially isolated, and *iso4* is a fully isolated model where each instance has its own definition. As expected and shown in Table 5.1, the more instances we isolate, the more behaviors (*#bhvr*) and channels (*#chnl*) exist in the model. The number of lines of code increases as well, as does the compile time for out-of-order PDES due to the larger segment graph.

On the other hand, the out-of-order simulation gains significant speedup (up to 67.5%) due to the decreasing ratio of detected conflicts (*#conflicts*) among the increasing number of segments (*#seg*). This tendency is also clearly visible in the number of threads issued in parallel by the out-of-order PDES (*#OoO issues*). We conclude that instance isolation can significantly improve the run-time efficiency of out-of-order PDES. Next, we will automate this technique and address the overhead of larger designs and longer compilation time at the same time.

5.1.2 Instance Isolation without Code Duplication

Instance isolation essentially creates additional scheduling statements (i.e. *par*, *wait*, and *wait-for-time*) for specific instances. Different segments are then generated for these statements and the variable access information is separated. [2]

As described above, isolation textually creates an additional class with a different name (e.g. *B_iso1* in Fig. 5.2a), but all statements remain the same as in the original definition (e.g. `B` in

[1] The simulation results in Section 3.6 are based on a set of isolated models to demonstrate the effectiveness of synchronous parallel discrete event simulation.

[2] Instance isolation helps to distinguish different instances of the same type, i.e. behavior or channel. Here, multiple instances will not share the same segment which is different from the segments in Section 4.3.1 without applying instance isolation.

Fig. 5.1). This code duplication eases the understanding of isolation, but is not desirable for an implementation due to its waste of resources.

In the following section, we propose a compiler optimization that uses the original statements without duplication, but still creates separate segments for different instances and attaches the variable access information accordingly. Thus, the size of the design model and the program segment in the binary executable will not increase.

To achieve this, we distinguish instances by unique identifiers (i.e. *instID*) which the compiler maintains and passes to the simulator. At run time, the simulator can then use the current instance identifier to distinguish the segments even though they execute the same program code. In Fig. 5.2a for example, seg_3 of `b1` and seg_5 of `b2` originate from the same statement `waitfor 2` in line 12 (see also Fig. 5.1).

Also, when processing a function call in a segment, we need to analyze the function body to obtain its variable access information since these variables affect the same segment. Moreover, we need to analyze function definitions also to connect the segment graph correctly if additional segments are started due to segment boundary statements inside the function body. This analysis could grow exponentially with the size of the design if there are frequent deep function calls. We avoid this by *caching* the information obtained during the function analysis, so that we can reuse this data the next time the same function is called. The time complexity of our compile-time analysis is therefore practically linear to the size of the code.

We present the detailed algorithm for the static code analysis, which automatically isolates the instances in the design "on-the-fly" without duplicated source code and with only minimal compile time increase, in the next section.

5.1.3 Definitions for the Optimized Static Conflict Analysis

At compile time, we use static analysis of the application source code to determine whether or not any conflicts exist between the segments. As shown in Algorithm 2, the compiler traverses the application's control flow graph following all branches, function calls, and thread creation points, and recursively builds the corresponding segment graph. The segment graph is then used to build the access lists and desired conflict tables.

To present the optimized algorithm in detail, we need to introduce a few definitions:

- **cacheMultiInfo**: a Boolean flag at each function for caching multiple sets of information for different instances; *true* if new segments are created or interface methods[3] are called in this function; *false* otherwise (default).

  Note that, if *cacheMultiInfo* is *false*, the cached information is the same for all instances that call the function.

- **cachedInfo**: cached information, as follows:
  - **instID**: instance identifier, e.g. `Main.b1`.
  - **dummyInSeg**: a dummy segment as the initial segment of the current set of cached information.
  - **carryThrough**: a Boolean flag; *true* when the input segment carries through the function and is part of the output segments; *false* otherwise. For example for `Main.b1`, `B.main.carrythrough` = *false* since seg_1 is connected to seg_3 and will not carry through `B.main`.
  - **outputSegments**: segments (without the input segment) that will be the output after analyzing this function. For example for `Main.b1`, `B.main.outputSegments`=$\{seg_3, seg_4\}$.

[3] Interface method definitions differ for different types of instances with different implementations.

– **segAccessLists**: list of segment access lists. The segments here are a subset of the global segments in the design. This list only contains the segments that are accessed by *instID.fct*.

During the analysis, when a statement is processed, there is always an input segment list and an output segment list. For example, for the `while` loop (Fig. 5.1, line 13), the input segment list is $\{seg_3\}$ and the output segment list is $\{seg_3, seg_4\}$. To start, we create an initial segment (i.e. seg_0) for the design as the input segment of the first statement in the program entrance function, i.e. `Main.main()`.

5.1.4 Algorithm for Static Conflict Analysis

Our optimized algorithm for the static code analysis consists of four phases, as follows:

- **Input:** Design model (e.g. from file `design.sc`).

  Output: Segment graph and segment conflict tables.

- **Phase 1:** Use Algorithm 2 to create the *global* segment graph, where Algorithm 4 lists the details of function *BuildFunctionSegmentGraph()* at line 45.

  If no function needs to be cached for multiple instances, the complexity of this phase is $O(n)$ where n is the number of statements in the design.

- **Phase 2:** Use Algorithm 5 to build a *local* segment graph with segment access lists for each function. Here, we only add variables accessed in this function to the segment access lists. We do not follow function calls in this phase.

  The complexity of this phase is $O(n_s)$ where n_s is the number of statements in the function definition.

  Fig. 5.3a illustrates this for the example in Fig. 5.2. We do not follow function calls to

`B.main` from `Main.main` but connect a *dummyInSeg* node instead. We also create two sets of `main` function cached information, shown in Fig. 5.3b and Fig. 5.3c, for the two instances of `B` since segment boundary nodes are created when calling `B.main`.

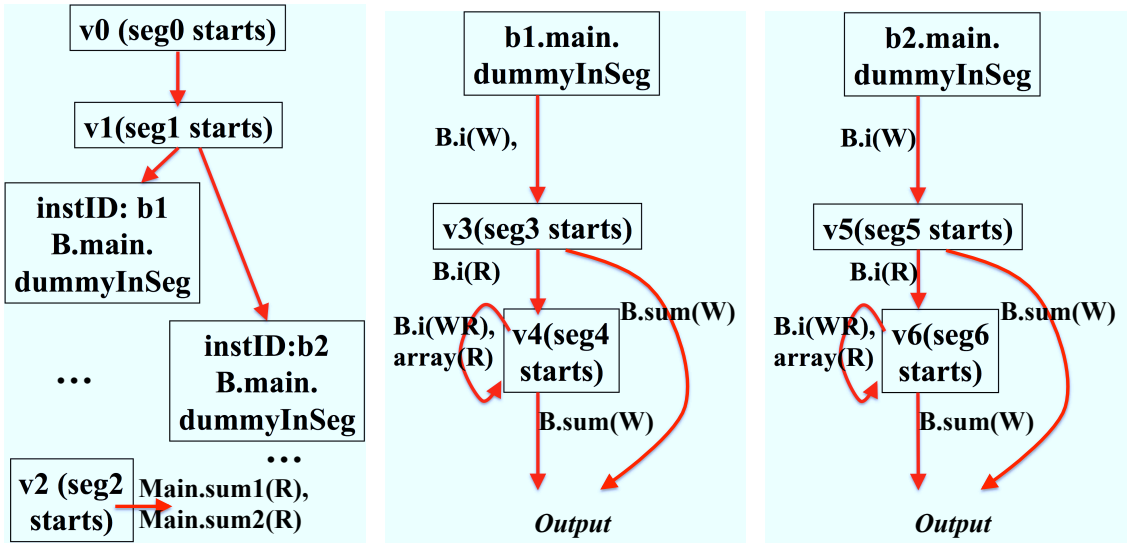
Note that we do not know yet the instance path of the member variables in this phase. Therefore, we use port variable `sum` instead of its real mapping `Main.sum1` and `Main.sum2` here.

Algorithm 4 Code analysis for out-of-order PDES, Phase 1: BuildFunctionSegmentGraph(currSegList, fct)

```
 1: if fct is first called then
 2:   BuildSegmentGraph(currSegList, fct.topstmnt);
 3:   if new segment nodes are created in fct then
 4:     set fct.cacheMultiInfo = true;
 5:     cache function information with current instID;
 6:   else
 7:     cache function information without instID; endif
 8: else /*fct has already been analyzed*/
 9:   if fct.cacheMultiInfo = false then
10:     /*no segments are created by calling this function*/
11:     use the cached information of fct;
12:   else /*new segments are created by calling this function*/
13:     if current instID is cached then
14:       use the cached information of fct.cacheinfo[instID];
15:     else
16:       BuildSegmentGraph(currSegList, fct.topstmnt);
17:       cache function information with current instID; endif
18:   endif
19: endif
```

Algorithm 5 Code analysis for out-of-order PDES, Phase 2

```
1: for all function fct do
2:     Traverse the control flow of fct.
3:     Create and maintain a local segment list localSegments.
4:     Use fct.dummyInSeg as the initial input segment.
5:     For each statement, add variables with their access type into proper segments.
6:     for all function calls inst.fct or fct do
7:         Do not follow the function calls. Just register inst.fct.dummyInSeg or fct.dummyInSeg as the
           input segments of the current statement and indicate that inst.fct or fct is called in the input
           segments.
8:         Add the output segments of the function call to localSegments and use the output segments as
           the input of the next statement in the current function.
9:     end for
10: end for
```



(a) Main.main (b) B.main (Main.b1) (c) B.main (Main.b2)

Figure 5.3: Function-local segment graphs and variable access lists for the example in Fig. 5.2.

- **Phase 3:** Build the *complete* segment access lists for each function. Here, we propagate function calls and add all accessed variables to the cached segment access lists cascaded with proper instance paths.

  For our example, `b1` is cascaded to the instance paths of the member variables accessed in

segment `B.main.dummyInSeg` for instance `b1` when analyzing `Main.main` (if necessary)[4]. We also use the function caching technique here to reduce the complexity of the analysis. The complexity of this phase is $O(n_g)$ where n_g is the size of the segment graph for each function.

- **Phase 4:** Collect the access lists for each segment in `Main.main` and add them to the global segment access lists. Since `Main.main` is the program entry (or root function), all the segments are in its local segments and the member variables have complete cascaded instance paths in the segment access lists. The actual mapping of port variables can now be found according to their instance path.

  The complexity of this phase is $O(n_l)$ where n_l is the size of the local segment access list of `Main.main`.

In summary, the algorithm generates precise segment conflict information using implicit instance isolation. The overall complexity is, for practical purposes, linear to the size of the analyzed design.

As a limitation, we do not support the analysis of recursive function calls (future work).

5.1.5 Experimental Results

To demonstrate the effects of our optimization, we have implemented the synchronous and out-of-order PDES algorithms for the SpecC SLDL[5]. We have run two sets of experiments and have measured the results on the same host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz.

The first experiment uses the same JPEG image encoder design in Section 4.4.2. It performs the *DCT*, *Quantization* and *Zigzag* modules for the three color components (Y, Cb, Cr) concurrently,

[4]Regular member variables accessed in different instances will not cause data hazards. Only port variables and interface member variables need the instance path for tracing the real mapping later.

[5]Our results should be equally applicable to SystemC SLDL.

and uses a sequential Huffman encoder at the end. The size of the input BMP image is 3216x2136 pixels.

The second experiment uses the same parallelized video decoder model in Section 4.4.3 with four parallel slice decoders to decode the separate slices in one frame simultaneously. We use the same test stream of 1079 video frames with 1280x720 pixels per frame.

Table 5.2 and Table 5.3 lists our experimental results for both design models at four different abstraction levels (*spec*, *arch*, *sched*, *net*). We measure and compare the out-of-order PDES algorithms in compile and simulation run time against the *synchronous PDES* implementation as reference.

The results shown in Table 5.2 and Table 5.3 clearly support the two main contributions of this optimization. First, instance isolation is very effective in improving the simulation speed, as shown in the columns for the unoptimized out-of-order PDES. Second and more importantly, our new analysis algorithm for optimized out-of-order PDES not only shows high speedup in simulation due to the automatic isolation "on-the-fly", it also shows only an insignificant increase in compile time of less than 6%.

Table 5.2: Compilation time for a Set of JPEG Image Encoder and H.264 Video Decoder Models.

Model		Synchronous PDES	Unoptimized Out-of-Order PDES		Optimized Out-of-Order PDES
			Original Design	Isolated Design	
		compile [sec]	compile [sec] (speedup)	compile [sec] (speedup)	compile [sec] (speedup)
JPEG Encoder	spec	0.95	0.96 (0.99)	1.13(0.84)	1.01 **(0.94)**
	arch	1.24	1.25 (0.99)	1.37 (0.91)	1.28 **(0.97)**
	sched	1.30	1.31 (0.99)	1.43 (0.91)	1.34 **(0.97)**
	net	1.50	1.52 (0.99)	1.63 (0.92)	1.55 **(0.97)**
H.264 Decoder	spec	13.12	18.76(0.70)	18.13 (0.72)	12.25 **(1.07)**
	arch	12.20	17.89 (0.68)	18.46 (0.66)	12.77 **(0.96)**
	sched	18.34	24.23 (0.76)	24.80 (0.74)	18.85 **(0.97)**
	net	19.07	25.71 (0.74)	26.06 (0.73)	19.54 **(0.98)**

Table 5.3: Simulation run time for a Set of JPEG Image Encoder and H.264 Video Decoder Models.

Model		Synchronous PDES	Unoptimized Out-of-Order PDES		Optimized Out-of-Order PDES
			Original Design	Isolated Design	
		simulation [sec]	simulation [sec] (speedup)	simulation [sec] (speedup)	simulation [sec] (speedup)
JPEG Encoder	spec	1.84	1.84 (1.00)	0.92 (2.00)	0.96 **(1.92)**
	arch	1.82	1.70 (1.07)	0.95 (1.92)	0.92 **(1.98)**
	sched	1.80	1.72 (1.05)	0.94 (1.91)	0.96 **(1.88)**
	net	2.33	2.01 (1.16)	1.25 (1.86)	1.24 **(1.88)**
H.264 Decoder	spec	96.91	96.97 (1.00)	60.33 (1.61)	60.25 **(1.61)**
	arch	99.86	100.30 (1.00)	60.77 (1.64)	59.78 **(1.67)**
	sched	99.80	99.44 (1.00)	60.96 (1.64)	60.29 **(1.66)**
	net	104.77	104.56(1.00)	66.25 (1.58)	66.00 **(1.59)**

5.2 Optimized Scheduling Using Predictions

Out-of-order PDES maintains simulation semantics and timing accuracy with the help of the compiler which performs complex static conflict analysis on the design source code. The scheduler utilizes the analysis results so as to make quick and safe decisions at run time and issue threads as early as possible. However, out-of-order scheduling is often prevented because of the unknown future behavior of the threads.

In this section, we extend the static code analysis in order to predict the future of candidate threads. Looking ahead of the current simulation state allows the scheduler to issue more threads in parallel, resulting in significantly reduced simulator run time.

5.2.1 State Prediction to Avoid False Conflicts

Out-of-order PDES issues threads in different simulation cycles to run in parallel if there are no potential hazards.

Fig. 5.5a shows the scheduling of thread execution for the example in Fig. 5.4. The threads th_1 and $th2$ are running in different segments with their own time. When one thread finishes its segment, shown as bold black bars as *scheduling point*, the scheduler is called for thread

synchronization and issuing.

The OoO PDES scheduling algorithm is very conservative. Sometimes it makes *false conflict* detections at run time. For instance, in Fig. 5.5a, when th_2 finishes its execution in seg_5 and hits the *scheduling point* $th_2.\textcircled{1}$, th_1 is running in seg_2. The current time is (1:0) for th_1 and (0:0) for th_2. As listed in Fig. 5.4c, the next time advance is (0:1) for seg_2 and (2:0) for seg_5. Therefore, the earliest time for th_1 to enter the next segment, i.e. seg_3, is (1:1), and for th_2 is (2:0). Since th_1 may run into its next segment (seg_3) with an earlier timestamp (1:1) than th_2 (2:0), the *Conflict()* in Algorithm 3 will return *true* at line 16. The scheduler therefore cannot issue th_2 out-of-order at scheduling point $th_2.\textcircled{1}$.

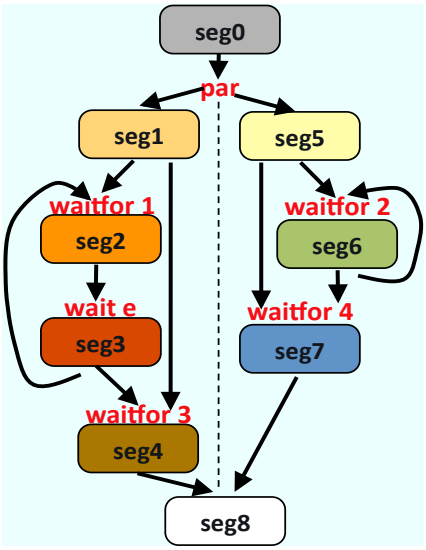
However, this is a *false conflict* for out-of-order thread issuing. Although th_1 may run into next segment (seg_3) earlier than th_2, there are no data conflicts between th_1's next segment seg_3 and th_2's current segment seg_6. Moreover, the next time advance of seg_3 is (1:0). So th_1 will start a new segment no earlier than (2:0) after finishing seg_3. It is actually safe to issue th_2 out-of-order at scheduling point $th_2.\textcircled{1}$ since th_2's time is not after (2:0).

If the scheduler knows what will happen with th_1 in more than one scheduling step ahead of scheduling point $th_2.\textcircled{1}$, it can issue th_2 to run in parallel with th_1 instead of holding it back for the next scheduling step.

This motivates our idea of optimizing out-of-order PDES scheduling. With prediction information, as shown in Fig. 5.5b, th_2 can be issued at both scheduling point $th_2.\textcircled{1}$ and $th_2.\textcircled{6}$. The simulator run time can thus be shortened.

```
1: #include <stdio.h>
2: int x = 0, y;
3: behavior B1(event e)        behavior B2(event e)
4: {                           {
5:   void main(){                 void main(){
6:     int i = 0;                    int i = 0;
7:     do {                          do {
8:       waitfor 1;                    waitfor 2;
9:       wait e;                       y = i;
10:      x = i;                        notify e;
11:    }while( i ++ < 2);            } while( i ++ < 2);
12:    waitfor 3;                    waitfor 4;
13:    x = 27;  }                    x = 42;   }
14: };                         };
15: behavior Main()
16: {
17:   event e; B1 b1(e); B2 b2(e);
18:   int main(){
19:     par{
20:       b1.main();
21:       b2.main();
22:   }
23:   printf("x = %d\n", x);}
24: };
```

(a) A simple design example



(b) Segment Graph

Data Conflict Table

seg	VAlist
0	
1	
2	
3	x(W)
4	x(W)
5	
6	y(W)
7	x(W)
8	x(R)

seg	0	1	2	3	4	5	6	7	8
0	F	F	F	F	F	F	F	F	F
1	F	F	F	F	F	F	F	F	F
2	F	F	F	F	F	F	F	F	F
3	F	F	F	T	T	F	F	T	T
4	F	F	F	T	T	F	F	T	T
5	F	F	F	F	F	F	F	F	F
6	F	F	F	F	F	F	T	F	F
7	F	F	F	T	T	F	F	T	T
8	F	F	F	T	T	F	F	T	F

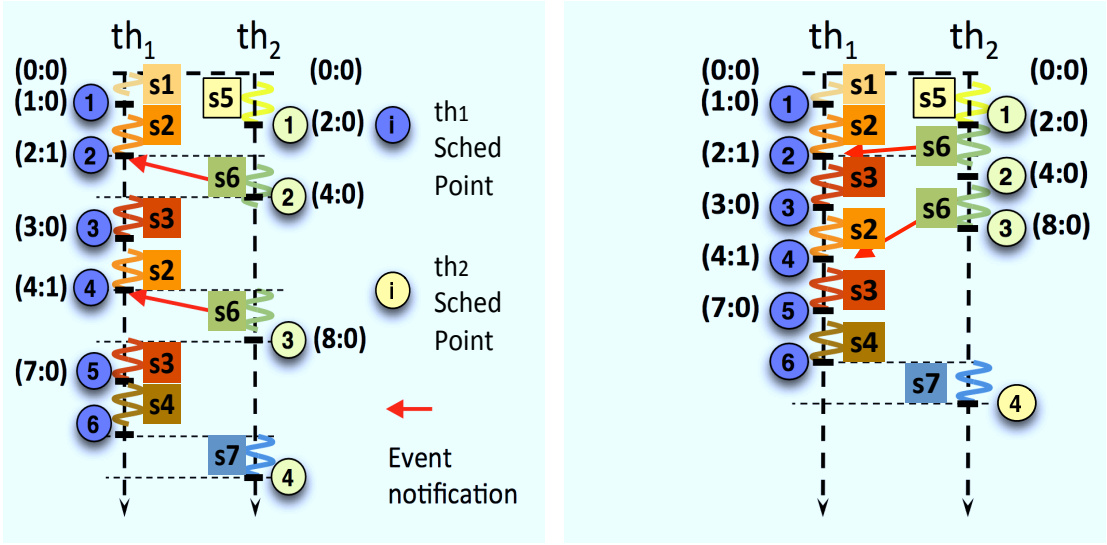
seg	NT
0	(0:0)
1	(1:0)
2	(0:1)
3	(1:0)
4	(0:0)
5	(2:0)
6	(2:0)
7	(0:0)
8	(∞,0)

Event Notification Table

seg	0	1	2	3	4	5	6	7	8
0	F	F	F	F	F	F	F	F	F
1	F	F	F	F	F	F	F	F	F
2	F	F	F	F	F	F	F	F	F
3	F	F	F	F	F	F	F	F	F
4	F	F	F	F	F	F	F	F	F
5	F	F	F	F	F	F	F	F	F
6	F	F	F	T	F	F	F	F	F
7	F	F	F	F	F	F	F	F	F
8	F	F	F	F	F	F	F	F	F

(c) Variable access list, data conflict, next time advance, and event notification table

Figure 5.4: Simple design example.



(a) Original Scheduling

(b) Optimized Scheduling

Figure 5.5: Out-of-order PDES scheduling.

5.2.2 Static Prediction Analysis

Out-of-order PDES relies on static code analysis for safe scheduling decisions. The knowledge of future thread status helps the scheduler to issue more threads out of the order for faster simulation.

At run time, threads switch back and forth between the states of RUNNING and WAITING. While RUNNING, the threads execute specific *segments* of their code. The out-of-order PDES scheduler checks the status of the threads by looking up the data structures for the *segments*.

The *Segment Graph* illustrates the execution order of the *segments* and their boundaries when the scheduler is called. The future segment information from any current segment can be derived from the *Segment Graph* at compile time.

We define the following data structures for static prediction analysis:

Data hazards prediction

- **Segment Adjacency Matrix (A[N,N])**:

$$A[i,j] = \begin{cases} 1 & \text{if } seg_i \text{ is followed by } seg_j; \\ 0 & \text{otherwise.} \end{cases}$$

- **Data Conflict Table with n prediction steps ($\mathbf{CT}_n$[N,N])** as follows:

$$CT_n[i,j] = \begin{cases} \textit{true} & \text{if } seg_i \text{ has a potential data conflict} \\ & \text{with } seg_j \text{ within n scheduling steps;} \\ \textit{false} & \text{otherwise.} \end{cases}$$

  Here, CT_0[N,N] is the same as segment data conflict table CT[N,N]. However, CT_n (n>0) is asymmetric.

Fig. 5.6(a) and (b) shows a partial segment graph and its *Adjacency Matrix*. The *Data Conflict Table* is shown in Fig. 5.6c where a data conflict exist between seg_3 and seg_4.

The *Data Conflict Tables with 0, 1 and 2 prediction steps* are shown in Fig. 5.7(a), (b) and (c), respectively. Since seg_2 is followed by seg_3 and seg_3 has a conflict with seg_4, a thread in seg_2 has a conflict with a thread who is in seg_4 after one scheduling step. Thus, $CT_1[2,4]$ is *true* in Fig. 5.7b. Similarly, seg_1 is followed by seg_2 and seg_2 is followed by seg_3, so $CT_2[1,4]$ is *true* in Fig. 5.7c.

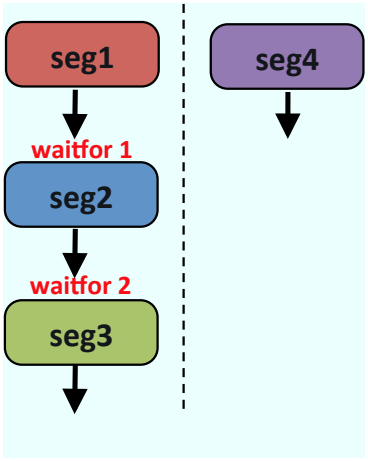
The *Data Conflict Table with n prediction steps* can be built recursively by using Boolean matrix multiplication. Basically, if seg_i is followed by seg_j, and seg_j has a data conflict with seg_k within the next $n-1$ prediction steps, then seg_i has a data conflict with seg_k within the

next n prediction steps. Formally,

$$CT_0[N,N] = CT[N,N] \tag{5.1}$$

$$CT_n[N,N] = A'[N,N] * CT_{n-1}[N,N], \text{ where n} > 0. \tag{5.2}$$

Here, A'[N,N] is the *modified Adjacency Matrix* (e.g. Fig. 5.7d) with 1s on the diagonal so as to preserve the conflicts from the previous data conflict prediction tables. Note that more conflicts will be added to the conflict prediction tables when the number of prediction steps increases.



(a) Segment Graph

seg	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	0
4	0	0	0	0

(b) Segment Adjacency Matrix

seg	1	2	3	4
1	F	F	F	F
2	F	F	F	F
3	F	F	F	T
4	F	F	T	F

(c) Data Conflict Table

Figure 5.6: A partial Segment Graph with Adjacency Matrix and Data Conflict Table.

Theorem 5.2.1. *$\exists M_{FP}$, $M_{FP} > 0$, so that $\forall n \geq M_{FP}$, no more conflicts will be added to CT_n.*

Proof. Eq. (5.1) and (5.2) $\Rightarrow CT_n = A'^n * CT$.

In A', $A'[i,j] = 1 \iff seg_j$ directly follows seg_i.

In A'^2, $A'^2[i,j] = 1 \iff \exists k$ that $A'[i,k] = A'[k,j] = 1$ or $A'[i,j] = 1$. In other words, $A'^2[i,j] = 1$ means that seg_j can be reached from seg_i via at most 1 other segment (1 segment

seg	1	2	3	4
1	F	F	F	F
2	F	F	F	F
3	F	F	F	T
4	F	F	T	F

(a) Data Conflict Table w 0 prediction step (CT_0)

seg	1	2	3	4
1	F	F	F	F
2	F	F	F	T
3	F	F	F	T
4	F	F	T	F

(b) Data Conflict Table w 1 prediction step (CT_1)

seg	1	2	3	4
1	F	F	F	T
2	F	F	F	T
3	F	F	F	T
4	F	F	T	F

(c) Data Conflict Table w 2 prediction steps (CT_2)

seg	1	2	3	4
1	1	1	0	0
2	0	1	1	0
3	0	0	1	0
4	0	0	0	1

(d) Modified Segment Adjacency Matrix

seg	0	1	2
1	(1:0)	(3:0)	(∞,0)
2	(2:0)	(∞,0)	(∞,0)
3	(∞,0)	(∞,0)	(∞,0)
4	(∞,0)	(∞,0)	(∞,0)

(e) Time Advance Table w 0, 1, 2 prediction steps

seg	1	2	3	4
1	0	0	0	3
2	0	0	0	2
3	0	0	0	1
4	0	0	1	0

(f) Combined Data Conflict Table

Figure 5.7: Data structures for optimized out-of-order PDES scheduling.

apart). Hence, $A'^n[i,j] = 1$ means seg_j can be reached from seg_i via at most n other segments (n segments apart).

Since there are a limited number of segments in the *Segment Graph*, $\exists L$ that $\forall i, j$, seg_i and seg_j are either at most L segments apart or they can never be reached from each other.
$\Rightarrow$ There exists a *fixpoint* $M_{FP} = L$, that $\forall n > M_{FP}$, $A'^n = A'^{M_{FP}}$, and $CT_n = A'^n * CT = A'^{M_{FP}} * CT = CT_{M_{FP}}$. □

Theorem 5.2.1 states that the number of prediction conflict tables for each design is limited. The maximum number of predictions is at most the length of the longest path in the Segment Graph.

Time Hazards Prediction

- **Time Advance Table with n prediction steps ($\mathbf{NT}_n$[N])**:

  NT_n[i] = min{thread time advance after n + 1 scheduling steps from seg_i}.

  Here, $NT_0 = NT$.

Fig. 5.7e shows the segment *Time Advance Table with Predictions* (NT_n) for the example in Fig. 5.6. If a thread is now running in seg_1, it will be in seg_2 after one scheduling step and in seg_3 after two scheduling steps. The thread time will advance by at least (3:0) after two scheduling steps since seg_2 starts from *waitfor 1* and seg_3 starts from *waitfor 2*. Therefore, $NT_1[1] = (3:0)$.

Event Hazards Prediction

We need prediction information for event notifications to handle event hazards.

- **Event Notification Table with predictions (ETP[N, N])**:

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

  Here, we have table entries of time advances.

Note that a thread can wake up another thread directly or indirectly via other threads. For instance, th_1 wakes up th_2, and th_2 then wakes up th_3 through event delivery. In this case, th_1 wakes up th_2 directly, and th_3 indirectly via th_2. We predict the minimum time advances between each thread segment pair in respect of both direct or indirect event notifications. The

scheduler needs the predicted event notification information to know when a new thread may be ready to run for conflict checking at run time.

5.2.3 Out-of-order PDES scheduling with Predictions

The out-of-order PDES scheduler issues threads out of the order at each scheduling step only when there are no potential hazards. With the help of static prediction analysis, we can optimize the scheduling conflict detection algorithm to allow more threads to run out-of-order.

Algorithm 6 shows the conflict checking function with M ($0 \leq M \leq M_{FP}$) prediction steps. Note that when M=0, it is the original out-of-order PDES conflict detection.

Algorithm 6 Conflict Detection with M Prediction Steps

```
1:  bool Conflict(Thread th, Thread th_2)
2:  {
3:    /*iterate the prediction tables for data and time hazards*/
4:    for (m = 0; m<M; m++) do
5:      if (CT_m[th_2.seg, th.seg] == true) then
6:        return true; end if /*data hazards*/
7:      if (th_2.timestamp + NT_m[th_2.seg]≥th.timestamp) then
8:        break; /*no data or time hazards between th_2 and th*/ end if
9:    end for
10:   if (m > M && M < M_FP ) then
11:     return true; end if /*time hazards*/
12:   /*check event hazards*/
13:   for all th_w ∈ WAIT do
14:     if(ETP[th_2.seg, th_w.seg] + th_2.timestamp<th.timestamp) then
15:       /*th_w may wake up before th*/
16:       check data and time hazards between th_w and th; endif
17:   end for
18:   return false;
19: }
```

Now, assume that th_1 and th_2 are two threads in the simulation of a model whose Segment Graph is Fig. 5.6a. $th_1$is ready to run in seg_4 with timestamp (3:0), and th_2 is still running in seg_1 with timestamp (1:0).

Conflict(th_1) in Algorithm 3 will return *true* because th_2 is possible to enter seg_2 with timestamp of (2:0) that is before th_1. Since the scheduler does not have information about the future status

of th_2, it cannot issue th_1 to run out-of-order at the current scheduling step.

Conflict(th_1) in Algorithm 6 will return *false* when M=1 or 2. With prediction information, the scheduler will figure out that th_1 (in seg_4) will not have data conflicts with th_2 after its next scheduling step (then in seg_2). Moreover, after th_2 finishes seg_2, the time for the next segment is at least (4:0), which is after th_1's current one, i.e. (3:0). It is safe to issue th_1 out-of-order at the current scheduling step. As shown, the prediction information helps the run-time conflict checking to eliminate a *false conflict*.

5.2.4 Optimized out-of-order PDES Scheduling Conflict Checking with a Combined Prediction Table

We observe that CT_m contains all the conflicts from CT_0 to CT_{m-1} (m>0). In Algorithm 6, the checking loop in line 4-9 stops when the first conflict is found from the CT_ns.

We propose an optimized conflict checking algorithm (Algorithm 7) by using the following data structure:

- **Combined Conflict Prediction Table (CCT[N,N])**:

$$CCT[i,j] = \begin{cases} k{+}1 & \min\{k \mid CT_k[i,j] = true\}; \\ 0 & \text{otherwise.} \end{cases}$$

  As shown in Fig. 5.7f, the number of prediction steps is stored in CCT instead of Boolean values.

There is no loop iteration for checking the conflict prediction table in Algorithm 7 since only one NxN combined table is used instead of M NxN data conflict prediction tables.

Note that, Theorem 5.2.1 proves that only a *fixed* number of data conflict tables with predictions are needed for a specific design. The compiler can generate the complete series of conflict

Algorithm 7 Optimized Conflict Detection with Combined Prediction Tables for M steps

```
1: bool Conflict(Thread th, Thread th2)
2: {
3:   /*check the combined prediction table for data and time hazards*/
4:   m = CT[th2.seg, th.seg] - 1;
5:   if(m ≥ 0) then /*There are data conflicts within M scheduling steps*/
6:   /*th2 may enter into a segment before th and cause data hazards*/
7:     if(th2.timestamp + NTm[th2.seg]<th.timestamp) then
8:       return true; end if
9:   else if (M < MFP)
10:    /*hazards may happen after M scheduling steps*/
11:    if(th2.timestamp + NTM[th2.seg]<th.timestamp) then
12:      return true; end if
13:  endif
14:  /*check event hazards*/
15:  for all thw ∈ WAIT do
16:    if(ETP[th2.seg, thw.seg] + th2.timestamp< th.timestamp) then
17:      /*thw may wake up before th*/
18:      check data and time hazards between thw and th; endif
19:  end for
20:  return false;
21: }
```

prediction tables and combine them into one table, i.e. CCT[N,N]. With this complete combined prediction table CCT, line 9-12 can be removed from Algorithm 7.

5.2.5 Experimental Results

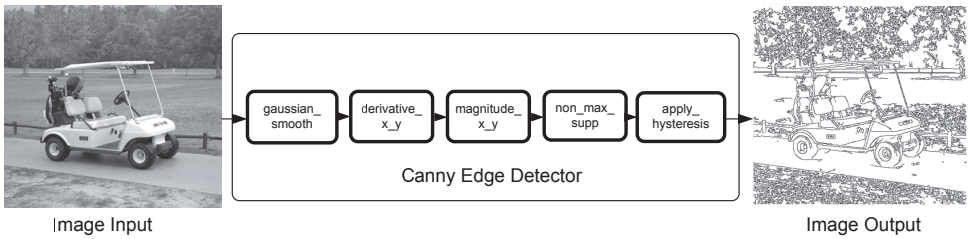
We have implemented the proposed static prediction analysis and the optimized out-of-order PDES scheduler in the SCE system design environment, and conducted experiments on three multi-media applications.

To demonstrate the benefits of out-of-order PDES scheduling using predictions, we show the compiler and simulator run times with different number of predictions in this section. Since one of our applications, i.e. the H.264 encoder, has 30 parallel units for motion estimation, we perform our experiment on a symmetric multi-processing (SMP) capable server consists of 2 Intel(R) Xeon(R) X5650 processors running at 2.67 GHz Each CPU contains 6 parallel cores, each of which supports 2 hyper-threads per core. The server specifically runs the 64-bit Fedora

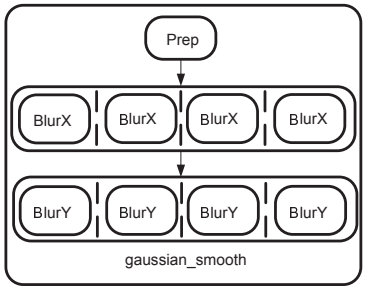
Core 12 Linux operating system.

Edge Detection with Parallel Gaussian Smoothing

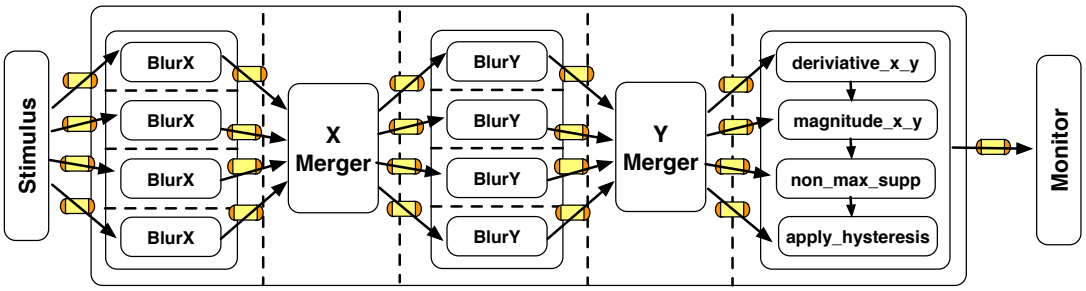
Our first application example is a **Video Edge Detector** which calculates edges in the images of a video stream. This model is an extension to the parallel Canny image edge detector [104] which takes multiple frames in the video stream one by one as the input.



(a) The Canny image edge detector flowchart (source [104])



(b) The parallel gaussian_smooth module (source [104])



(c) The video edge detector example

Figure 5.8: An video edge detector example

The application parallelizes the most computationally complex function *Gaussian Smooth* in

the design (approximately 45% of the total computation) on 4 cores. Fig. 5.9a shows the result with a test video stream of 100 frames with 1280x720 pixels. The simulation speed increases with more prediction steps. With the maximum prediction information, Table 5.4 shows a speedup of 1.15 with very small increase of compilation time.

H.264/AVC Video Decoder with Parallel Slice Decoding

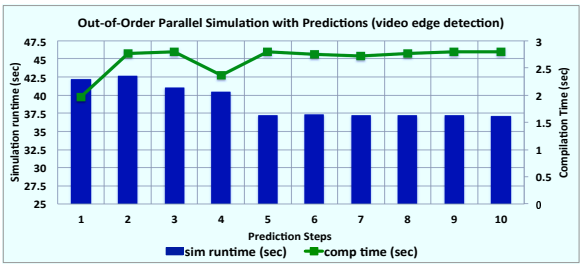
Our second application is the parallelized **H.264/AVC Video Decoder** described in Section 3.6.1. It uses four parallel slice decoders to decode the independent slices in a video frame simultaneously. We use a test stream of 1079 video frames with 1280x720 pixels per frame and simulate the model at four different abstraction levels, i.e. specification, architecture mapped, scheduling refined, and network linkage allocated.

Table 5.4 shows an average speedup of 1.89 for simulation with maximum prediction information compared to the baseline out-of-order PDES simulation without predictions. Note that even for such a large design, the increased compile time due to the static prediction analysis is negligible. Fig. 5.9b shows that more simulation speedup can be gained with more prediction steps.

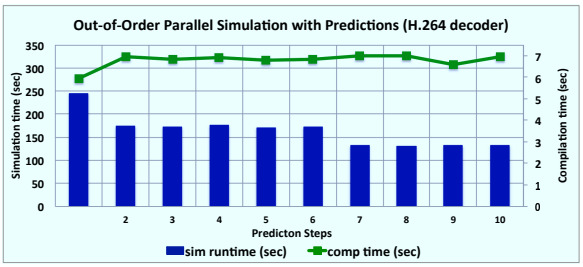
H.264/AVC Video Encoder with Parallel Motion Search

The third application is a parallelized **H.264/AVC Video Encoder** with parallel motion search [105]. Intra- and inter-frame prediction are applied to encode an image according to the type of the current frame. During inter-frame prediction, the current image is compared to the reference frames in the decoded picture buffer and the corresponding error for each reference image is obtained. In our model, multiple motion search units are processing in parallel so that the comparison between the current image and multiple reference frames can be performed simultaneously. The test stream is a video of 95 frames with 176x144 pixels per frame. Table 5.4 shows a speedup of 1.88 for simulation with complete prediction information.

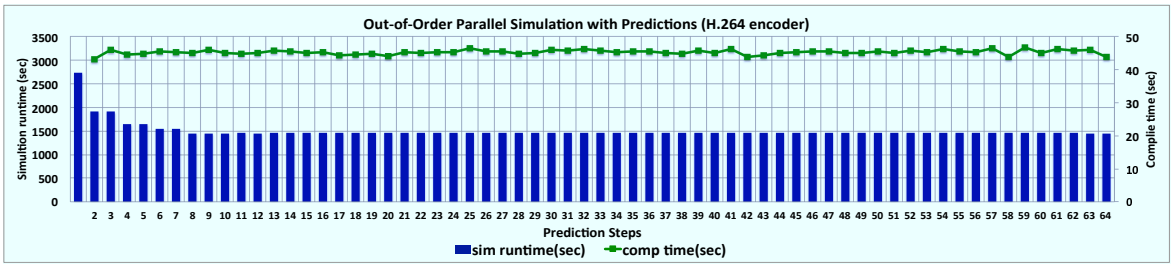
As a large industrial design, the prediction conflict tables get to the *fixpoint* after 62 prediction steps. Fig. 5.9c shows the same trend of simulation speedup vs. prediction steps.



(a) the video edge detection model



(b) the H.264 decoder model



(c) the H.264 encoder model

Figure 5.9: Simulation run time and compilation time for OoO PDES with predictions

Table 5.4: Experimental results for embedded applications

Simulator:		Out-of-order PDES without Predictions		Out-of-order PDES with Predictions		
		compile time [sec]	sim time [sec]	compile time [sec] / speedup	sim time [sec] / speedup	max pred steps
Edge Detection		1.97	42.19	2.80 / 0.70	37.05 / **1.14**	8
H.264 Decoder	spec	5.86	244.98	6.95 / 0.85	131.99 / **1.86**	8
	arch	6.53	242.95	6.95 / 0.94	132.78 / **1.83**	7
	sched	6.95	244.32	7.24 / 0.96	133.23 / **1.83**	8
	net	6.55	244.61	7.20 / 0.91	132.85 / **1.92**	9
H.264 Encoder		37.95	2719.44	43.82 / 0.87	1448.81 / **1.88**	62

6 Comparison and Outlook

In this chapter, we will compare the three discrete event simulation approaches, i.e. the traditional sequential DES, the synchronous PDES, and the out-of-order PDES, with a comprehensive set of experiments, and discuss their advantages and limitations [106] [1].

6.1 Experimental Setup

To demonstrate the potential of parallel simulation, we have designed three highly parallel benchmark models:

1. a parallel floating-point multiplication example,
2. a parallel recursive Fibonacci calculator,
3. and a parallel recursive Fibonacci calculator with timing information.

All these benchmarks are system-level models specified in SpecC SLDL.

6.1.1 Experimental Environment Setup

For our experiments, we use a symmetric multi-processing server running 64-bit Fedora Core 12 Linux. The multi-core hardware specifically consists of 2 Intel(R) Xeon(R) X5650 processors

[1] for the our-of-order PDES, both of the optimizations discussed in Chapter 5 have been applied.

running at 2.67 GHz. Each CPU contains 6 parallel cores, each of which supports 2 hyper-threads per core. Thus, in total the server hardware supports up to 24 threads running in parallel.

We use the Linux */usr/bin/time* command to measure the simulation and compilation time for the experiment applications. According to [107], the *time* command provides the statistical report for the *time*, *memory*, and *input/output* usages for the measured program. For the experiments in this dissertation, we focus on the following numbers:

- **%S**, total number of CPU-seconds that the process spent in kernel mode
- **%U**, total number of CPU-seconds that the process spent in user mode
- **%E**, elapsed real time (in [hours:]minutes:seconds)
- **%P**, Percentage of the CPU that this job got, computed as (%U + %S) / %E

We observe that there are several factors can affect the run time measurement results. In other words, we can get different measurement numbers for different runs of compiling and simulating the same application. Some factors are listed below:

1. The workload overhead from the underlying Linux system
2. Delays due to operating system and micro-architecture features, such as cache misses, page faults, and memory contentions
3. Dynamic features due to the Linux strategies for mapping and scheduling multiple threads on multiple CPU cores at run time
4. Other users, who remotely connect to the simulation host, can run some programs at the same time and use some of the available CPU cores
5. Network delays, such as reading and writing delays, if the simulation host is using the Network File System (NFS) [108]

6. CPU frequency scaling support by the underlying operating system on the simulation computer platform

The first three items (1, 2, and 3) cannot be controlled for regular usage of multi-core platforms with operating system support. Our work aims at providing a fast simulation approach that can be applied on general multi-core computer platforms. Thus, we intentionally do not control these factors in our experiment.

The last three item (4, 5, and 6) are controlled by setting up our specific experiment environment. We notify all the remote users who can access the simulation host about the period of time to run our experiments so that they did not work on the computer at the same time. We use local directories instead of NFS mapped directories. We also disable the CPU frequency scaling and turbo mode of the simulation processors.

6.1.2 The Parallel Benchmark Models

We use three parallel benchmark models in this section to show the potential for the parallel discrete event simulations.

Parallel floating-point multiplications

Our first parallel benchmark **fmul** is a simple stress-test example for parallel floating-point calculations. Specifically, **fmul** creates 256 parallel instances which perform 10 million double-precision floating-point multiplications each. As an extreme example, the parallel threads are completely independent, i. e. do not communicate or share any variables.

The chart in Fig. 6.1a shows the experimental results for our synchronous PDES simulator when executing this benchmark. To demonstrate the scalability of parallel execution on our server, we vary the number of parallel threads admitted by the parallel scheduler (the value `#CPUs` in Fig. 3.2) between 1 and 32.

We use the elapsed simulator run time for one core as the base (33.5 seconds). When plotting the relative speedup, one can see that, as expected, the simulation speed increases in nearly linear manner the more parallel cores are used and tops out when no more CPU cores are available. The maximal speedup is about 16x for this example on our 24-core server.

Parallel Fibonacci calculation

Our second parallel benchmark **fibo** calculates the Fibonacci series in parallel and recursive fashion. Recall that a Fibonacci number is defined as the sum of the previous two Fibonacci numbers, $fib(n) = fib(n-1) + fib(n-2)$, and the first two numbers are $fib(0) = 0$ and $fib(1) = 1$. Our **fibo** design parallelizes the Fibonacci calculation by letting two parallel units compute the two previous numbers in the series. This parallel decomposition continues up to a user-specified depth limit (in our case 5), from where on the classic recursive calculation method is used.

In contrast to the **fmul** example above, the **fibo** benchmark uses shared variables to communicate the input and calculated output values between the units, as well as a few counters to keep track of the actual number of parallel threads (for statistical purposes). Thus, the threads are not fully independent from each other. Also, the computational load is not evenly distributed among the instances due to the fact that the number of calculations increases by a factor of approximately 1.618 (the *golden ratio*) for every next number.

The **fibo** simulation results are plotted in Fig. 6.1b. Again we use the elapsed simulator run time for one core as base (29.7 seconds). The curve for the relative simulation speedup shows the same increasing shape as in Fig. 6.1a. Speed increases in nearly linear fashion until it reaches saturation at about a factor of 12x.

When comparing the **fmul** and **fibo** benchmark results, we notice a more regular behavior of the **fmul** example due to its even load and zero inter-thread communication.

Parallel Fibonacci calculation with timing information

Our third parallel benchmark **fibo_timed** is an extension of **fibo** with timing information. System models usually have timing information either back-annotated by estimation tools or added by the designers to evaluate the real-time behavior of the design. Compared to the untimed **fibo**, this timed benchmark is a more realistic embedded application example.

fibo_timed has the same structure as **fibo** with the same parallel decomposition depth (in our case 5). Timing information is annotated using *wait-for-time* statements at each leaf block where the classic recursive calculation method is used. The time delay is determined by the computational load of the unit, i.e. $T_{fib(n)} = 1.618 * T_{fib(n-1)}$.

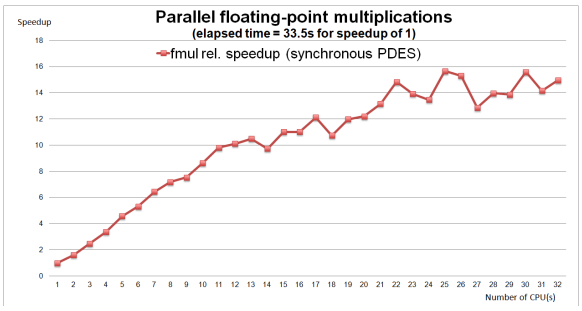
Fig. 6.1c plots the simulation results for both synchronous and out-of-order PDES. Using the 1-core elapsed simulator time as base (32.7 seconds for both simulators), the relative speedup shows that out-of-order PDES can exploit more parallelism during the simulation and is more efficient than synchronous PDES. This benchmark confirms the increased CPU utilization on a multi-core host by out-of-order PDES.

6.1.3 The Embedded Applications

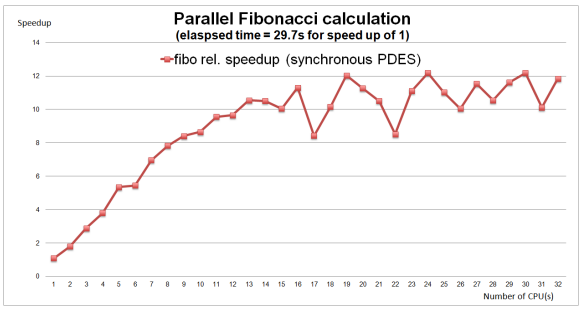
In this section, we use six embedded applications to demonstrate the effectiveness of the PDES approaches for realistic design examples. We modeled these embedded applications in-house based on reference source code for standard algorithms.

JPEG Image Encoder with Parallel Color Space Encoding

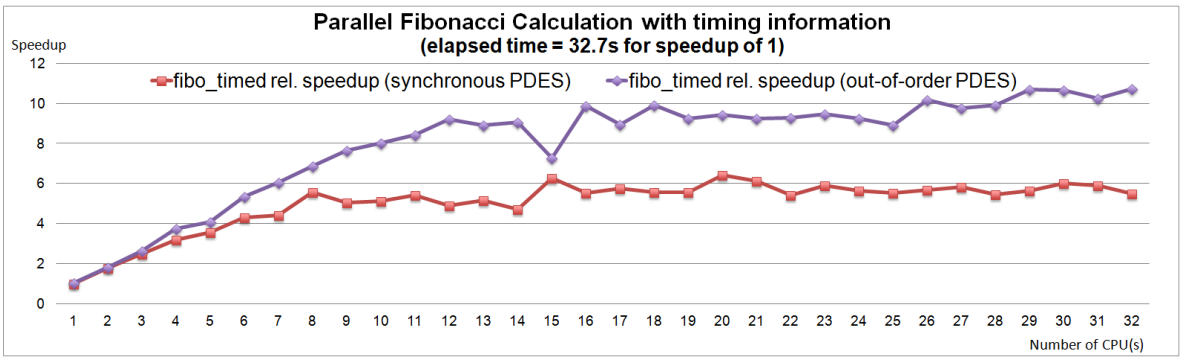
As described in Section 3.6.2, Section 4.4.2 and Section 5.2.5, the JPEG encoder performs its *DCT*, *Quantization* and *Zigzag* modules for the 3 color components in parallel, followed by a sequential *Huffman* encoder at the end. Table 6.1 shows the simulation speedup. The size of



(a) fmul (synchronous PDES)



(b) fibo (synchronous PDES)



(c) fibo_timed (synchronous PDES vs. out-of-order PDES)

Figure 6.1: Simulation results for highly parallel benchmark models.

our input BMP image is 3216x2136 pixels. Note that, the model has maximal 3 parallel threads, followed by a significant sequential part.

We simulate this application model at four abstraction levels (specification, architecture mapped, OS scheduled, network linked). As shown in Table 6.1, simulation speed increases for both parallel simulators but the out-of-order PDES gains more speedup than synchronous PDES.

H.264 Video Decoder with Parallel Slice Decoding

Our second application is the parallelized video decoder model based on the H.264/AVC standard as described in Section 3.6.1, sec:OoOPDES-experiment-h264 and Section 5.2.5. An H.264 video frame can be split into multiple independent slices during encoding. Our model uses

four parallel slice decoders to decode the separate slices in a frame simultaneously. The H.264 stimulus module reads the slices from the input stream and dispatches them to the four following slice decoders for parallel processing. A synchronizer block at the end completes the decoding of each frame and triggers the stimulus to send the next one. This design model is of industrial-size and consists of about $40k$ lines of code.

We use a test stream of 1079 video frames with 1280x720 pixels per frame (approximately 58.6% of the total computation is spent on the slide decoding which has been parallelized). Table 6.1 shows that synchronous PDES can hardly gain any speedup due to the simulation cycle barriers. Furthermore, protecting the shared resources and added synchronizations introduce simulation overhead for PDES. However, out-of-order PDES still gains significant speed up to a factor of 1.77x. Note that even for a large realistic design, such as this H.264 decoder model, the increased compilation time due to the static model analysis for out-of-order PDES is negligible.

Edge Detection with Parallel Gaussian Smoothing

Our third application example, a Canny edge detector application, calculates edges in images of a video stream (Section 5.2.5). In our model, we have parallelized the most computationally complex function *Gaussian Smooth* (approximately 45% of the total computation) on 4 cores. With a test stream of 100 frames of 1280x720 pixels, the simulation results in Table 6.1 show 1.38 speedup for synchronous PDES and 1.52 speedup for out-of-order PDES.

The fourth example uses the same edge detection algorithm but only detects the edges in a single image. Again we split the *Gaussian Smooth* function equally on 4 parallel modules, but use a larger image. For the test image with 3245x2500 pixels, PDES accelerates the simulation with an average speedup of 1.27. The workload is evenly distributed so it fully fills the simulation cycles of the mapped parallel threads. Thus, out-of-order PDES loses its advantage and performs slightly slower than synchronous PDES due to the out-of-order scheduling overhead.

H.264 Video Encoder with Parallel Motion Search

The fifth application is a parallelized video encoder based on the H.264/AVC standard (Section 5.2.5). Intra- and inter-frame prediction are applied to encode an image according to the type of the current frame. During inter-frame prediction, the current image is compared to the reference frames in the decoded picture buffer and the corresponding error for each reference image is obtained.

In our model, multiple motion search units are processing in parallel so that the comparison between the current image and multiple reference frames can be performed simultaneously. Out test stream is a video of 95 frames with 176x144 pixels per frame, and the number of B-slices between every I-slice or P-slice is 4. That is, among every 5 consecutive frames 4 frames need inter-frame prediction. Table 6.1 shows a similar simulation acceleration with a speedup of 1.87 for synchronous PDES, and 1.98 for out-of-order PDES.

MP3 Stereo Audio Decoder

The last application, a MP3 player, is another example for which the performance of PDES is marginal due to the limited parallelism in the model. Our MP3 audio decoder is modeled with parallel decoding for stereo channels. Our test stream is a 99.6 Kbps, 44.1 Hz joint stereo MP3 file with 2372 frames. It takes less than 5 seconds to simulate, but there are 7114 context switches in scheduling the two parallel threads. Here, both PDES approaches take longer time than the traditional DE simulation due to the low computation workload and the then significant overhead for synchronization.

Overall, we can see that the 24 available parallel cores on the server are under-utilized for all six applications, and by both parallel simulators. The reason is clearly the limited available parallelism in the models.

Table 6.1: Experimental results for embedded application examples using standard algorithms.

Simulator: Par. Issued Threads:		Single-thread reference n/a		Multi-core Synchronous parallel 24		Multi-core Out-of-Order parallel 24	
		compile time [sec]	simulator time [sec]	compile time [sec] / speedup	simulator time [sec] / speedup	compile time [sec] / speedup	simulator time [sec] / speedup
JPEG Encoder	spec	1.95	1.70	2.14 / 0.91	1.38 / **1.23**	2.24 / 0.87	0.63 / **2.70**
	arch	2.48	1.72	2.72 / 0.91	1.42 / **1.21**	3.04 / 0.81	0.66 / **2.60**
	sched	2.58	1.73	2.63 / 0.98	1.39 / **1.24**	2.73 / 0.95	0.64 / **2.70**
	net	2.77	2.79	3.20 / 0.86	2.12 / **1.32**	3.18 / 0.87	1.06 / **2.63**
H.264 Decoder	spec	4.94	230.53	5.28 / 0.94	231.49 / **1.00**	6.09 / 0.81	126.19 / **1.83**
	arch	5.17	229.70	5.53 / 0.93	232.17 / **0.99**	6.58 / 0.79	123.04 / **1.87**
	sched	5.2 2	231.55	5.44 / 0.96	232.08 / **1.00**	6.58 / 0.79	122.68 / **1.89**
	net	5.48	233.07	5.87 / 0.93	234.20 / **1.00**	6.45 / 0.85	124.20 / **1.88**
Video Edge Detection		1.30	60.93	1.73 / 0.75	44.07 / **1.38**	1.82/ 0.71	40.10 / **1.52**
Image Edge Detection		2.57	2.99	2.71 / 0.95	2.34 / **1.28**	2.65 / 0.97	2.37 / **1.26**
H.264 Encoder		18.66	2875.83	20.88 / 0.89	1534.93 / **1.87**	22.94/ 0.81	1452.72 / **1.98**
MP3 Decoder		2.13	4.06	2.12 / 1.00	4.42 / **0.92**	2.31 / 0.92	4.67/ **0.87**

6.2 Parallel Discrete Event Simulation Overlook

Table 6.2 provides the technical details for the three discrete event simulation approaches.

Parallel discrete event simulation holds the promise to leverage the CPU utilization of multi-core computer host for fast simulation. However, the PDES performance also highly relies on the available parallelism in the simulated model. We list the advantages and limitations for the three discrete event simulation approaches below as a guidance for choosing different approaches to simulate different ESL models.

- **Traditional Sequential Discrete Event Simulation (sequential DES)**
  - Advantages:
    - Can use light-weight user-level multithreading library which has minimal context switch and thread management overhead
    - No need to protect thread synchronization and communication by the compiler, i.e. fast compilation
    - Simple thread synchronization and communication, i.e. fast scheduling

Table 6.2: Comparison of traditional sequential, synchronous parallel, and out-of-order parallel discrete event simulation

	Traditional DE simulation	Synchronous PDES	Out-of-Order PDES
Simulation Time	One global time tuple (t, δ) shared by every thread and event		Local time for each thread th as tuple (t_{th}, δ_{th}). A total order of time is defined with the following relations: **equal**: $(t_1, \delta_1) = (t_2, \delta_2)$, iff $t_1 = t_2$, $\delta_1 = \delta_2$. **before**: $(t_1, \delta_1) < (t_2, \delta_2)$, iff $t_1 < t_2$, or $t_1 = t_2$, $\delta_1 < \delta_2$. **after**: $(t_1, \delta_1) > (t_2, \delta_2)$, iff $t_1 > t_2$, or $t_1 = t_2$, $\delta_1 > \delta_2$.
Event Description	Events are identified by their *id*s, i.e, event (id).		A timestamp is added to identify every event, i.e. event (id, t, δ).
Simulation Thread Sets	**READY**, **RUN**, **WAIT**, **WAITFOR**, **JOINING**, **COMPLETE**		Threads are organized as subsets with the same timestamp (t_{th}, δ_{th}). Thread sets are the union of these subsets, i.e, **READY** $= \cup$**READY**$_{t,\delta}$, **RUN** $= \cup$**RUN**$_{t,\delta}$, **WAIT** $= \cup$**WAIT**$_{t,\delta}$, **WAITFOR** $= \cup$**WAITFOR**$_{t,\delta}$ $(\delta = 0)$, where the subsets are ordered in increasing order of time (t, δ).
Threading Model	User-level or OS kernel-level	***OS kernel-level***	
Run Time Scheduling	Event delivery in-order in delta-cycle loop. Time advance in-order in outer loop.		Event delivery out-of-order if no conflicts exist. Time advance out-of-order if no conflicts exist.
	Only one thread is active at one time. ***No parallelism***. ***No SMP utilization***.	Threads at same cycle run ***in parallel***. ***Limited parallelism***. ***Inefficient SMP utilization***.	Threads at same cycle or with no conflicts run ***in parallel***. ***More parallelism.*** ***Efficient SMP utilization.***
Compile Time Analysis	No synchronization protection needed.	***Need synchronization protection for shared resources***, e.g. any user-defined and hierarchical channels.	
	No conflict analysis needed.		Static conflict analysis derives Segment Graph (SG) from CFG, analyzes variable and event accesses, passes conflict table to scheduler. ***Compile time increases.***

  - Limitations:
    - Only allows one thread to be active at one time which makes it impossible to utilize the multiple resources on multi-core simulation hosts
  - Suitable models: models with no / limited parallelisms, i.e. the MP3 decoder.

- **Synchronous Parallel Discrete Event Simulation (SPDES)**
  - Advantages:
    - Allows multiple threads in the same simulation cycle to run in parallel which can exploit the multiple resources on multi-core simulation hosts to increase simulation speed

    * Is scalable on multi-core simulation hosts, i.e. the number of threads running in parallel can be easily set

  - Limitations:

    * Uses OS kernel-level multithreading library which has heavy context switch and thread management overhead

    * Needs thread synchronization and communication protection for parallel execution by the compiler which introduces a small amount of compilation overhead

    * Only allows threads in the same simulation cycle to run in parallel. The multi-core CPUs can be idle when there are not enough parallel threads available.

  - Suitable models: models with fully balanced (workload and timing) parallelisms, i.e. the image edge detector, the well balanced H.264 video decoder (Chapter 3).

- **Out-of-order Parallel Discrete Event Simulation (OoO PDES)**

  - Advantages:

    * Allows threads in different simulation cycle to run in parallel which can leverage the CPU utilization of the multi-core simulation host by reducing the CPU idle time

    * Is scalable on multi-core simulation hosts

    * Can mitigate the gap between logic parallelism that is understandable in the model and the real parallelism during simulation which is more limited due to the discrete event execution semantics

  - Limitations:

    * Uses OS kernel-level multithreading library which has heavy context switch and thread management overhead

    * Needs compiler support to generate conflict information
    * Needs scheduler support for dynamic conflict detection to preserve the execution semantics
    * Needs more scheduling than DES and SPDES. DES and SPDES only schedule when all the threads meet at the simulation cycle barrier. OoO PDES has to schedule at every thread scheduling point since the global simulation cycle barrier is removed by localizing into each thread
    * Needs thread synchronization and communication protection for parallel execution by the compiler
  - Suitable models: models with imbalanced (workload and timing) and independent (fewer conflicts) parallelism, i.e. the JPEG encoder, the H.264 decoder, the video edge detector, and the H.264 encoder.

The three discrete event simulation approaches have their own advantages and disadvantages.

In general, the sequential DES has the lightest implementation and is most suitable for models with no or very limited parallelism, such as communication intensive models.

Both synchronous PDES and out-of-order PDES are capable to exploit the multiple resources on multi-core simulation hosts when simulating parallel models. Synchronous PDES works best for well balanced models with respect to both workload and timing among the parallel threads. Out-of-order PDES has the most potential to exploit the parallelism during simulation. It works better than synchronous PDES for general models which have nontrivial timing annotations and event synchronizations. However, out-of-order PDES has heavier implementation overhead in terms of static conflict analysis and dynamic scheduling conflict detection.

It should also be emphasized that all efforts for parallel simulation are limited by the amount of exposed parallelism in the application. The Grand Challenge still remaining is the problem

of how to efficiently parallelize applications.

7 Utilizing the Parallel Simulation Infrastructure

In order to utilize parallel processing for low power and high performance in embedded systems, ESL models must contain explicit and efficient parallelism. Notably, parallelization is a particularly important but also very difficult task in system modeling.

Most reference code for embedded applications is sequentially specified. To parallelize the application, the designer must first identify suitable functions that can be efficiently parallelized, and then recode the model accordingly to expose the parallelism. Since identifying effective thread-level parallelism requires the designer's knowledge and understanding of the algorithm and therefore is a manual task, the model recoding is typically a tedious and error-prone process. Automating the coding, validation, and debugging is highly desirable.

In this chapter, we focus on the debugging support for writing parallel models. In particular, we use the parallel simulation infrastructure to increase the observability of developing parallel system-level models. A dynamic race condition detection approach will be proposed to help the designer narrow down the debugging targets in ESL models with parallelism [109].

7.1 Introduction

At the starting point of the electronic system-level (ESL) design flow, a well-defined specification model of the intended embedded system is critical for rapid design space exploration and efficient synthesis and refinement towards a detailed implementation at lower abstraction levels. Typically, the initial specification model is written using system-level description languages (SLDLs), such as SystemC and SpecC. In contrast to the original application sources, which usually are specified as unstructured sequential C source code, a well-defined system model contains a clear structural hierarchy, separate computation and communication, and explicit parallelism.

In this chapter, we address the problem of ensuring that the functionality of the model remains correct during parallel execution. In particular, the system model must be free of *race conditions* for all accesses to any shared variables, so that a safe parallel execution and implementation is possible.

Ensuring that there are no race conditions proves very difficult for the system designer, especially because the typically used discrete event simulation often does not reveal such mistakes in the model. It should be emphasized that the absence of errors during simulation does not imply the absence of any dangerous race conditions in the model. Even though parallel simulation on multi-core hosts has higher likelihood to have simulation error caused by race conditions, it cannot guarantee that all situations are exposed. Furthermore, it is very hard to debug as the cause for an invalid output value may be hidden deep in the complex model.

To solve this race condition problem, we use our parallel simulation infrastructure to provide specific debug information to the designer. Specifically, we propose a combination of (a) an advanced static conflict analysis in the compiler, (b) a table-based checking and tracing engine in the parallel simulator, and (c) a novel race-condition diagnosis tool. Together, these tools not only discover all race conditions that can potentially affect the concurrent execution, but also

provide the designer with detailed source line information of where and when these problems occur.

7.2 Overview and Approach

In the context of validating a parallel system model, our proposed approach utilizes the compiler and simulator tools from a parallel simulation infrastructure to automatically detect and diagnose potential data hazards in the model. As a result, the system designer can quickly resolve the hazards due to race conditions and produce a safe parallel model.

7.2.1 Creating Parallel System Models

Exposing thread-level parallelism in sequential applications models requires three main steps:

1. *Identify the blocks to parallelize*: The first step is to understand the application and its algorithms. With the help of statistics from a profiler, the designer can then identify suitable blocks in the application with high computational complexity for which parallelization is desirable and likely beneficial.

2. *Restructure and recode the model*: Through partitioning of functional blocks and encapsulating them into SLDL modules, the application is transformed into a system-level model with proper structure and hierarchy [110]. In particular, parallel execution is exposed explicitly.

   With parallelism inserted into the model structure, affected variables may need to be recoded appropriately to ensure correct functionality of the model. For example, the variables involved in the restructuring may need to be duplicated, relocated into appropriate scope (localized), or wrapped in channels with explicit communication. Here, proper data

dependency analysis is a critical component in resolving access conflicts due to parallelization. Performed manually, this is a tedious and error-prone task especially if the model is of modest or large size. The designer must locate affected variables, identify their all their read and write accesses, and make sure that no invalid accesses exist due to race conditions.

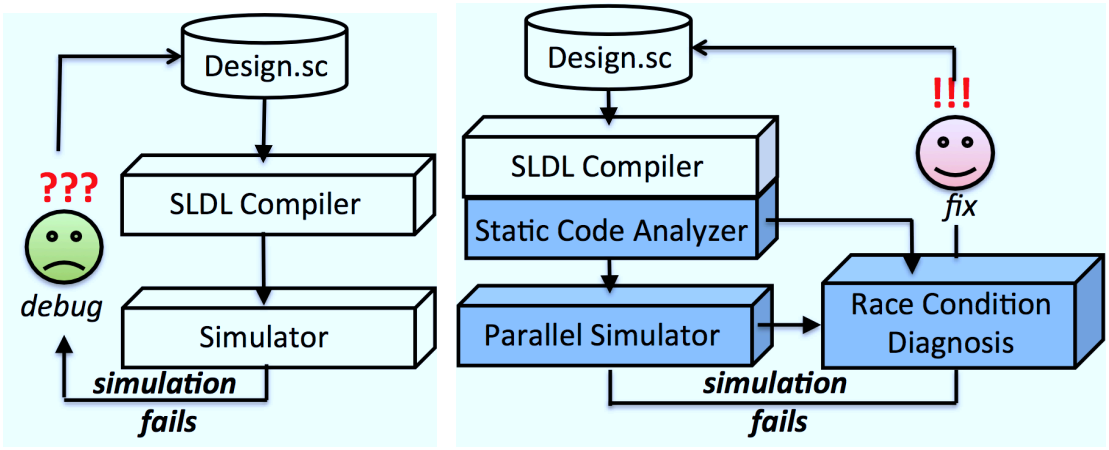
3. *Validate the model*: The correct functionality of the model is typically validated through simulation. However, regular simulators hide many potential access conflicts due to their sequential execution. Parallel simulators (Chapter 3), on the other hand, executes on multi-core CPUs in parallel and can thus expose some access conflicts and race conditions. If these lead to invalid simulation results, it tells the designer that the model has a problem, but not where the problem is. Most often it is then very difficult to locate the cause and correct the problem.

   Nevertheless, no existing simulation technique can prove the absence of race conditions. We will address this short-coming in this chapter by an extension of a parallel simulation infrastructure.

7.2.2 Shared Variables and Race Conditions

Variables shared among parallel modules in the system model can cause data hazards, i.e. read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) conflicts. Thus, invalid parallel accesses to shared variables in the system model must be prevented.

As discussed earlier, designers traditionally design the specification model and handle shared variables manually. As shown in Fig. 7.1a, model validation is then performed by compiling and simulating the model using a traditional DE simulator. If the simulation fails, the designer needs to identify the problem, locate it in the code, and revise the model for another iteration. This debugging is typically a lengthy and error-prone process.



(a) Manual Approach (b) Proposed Approach

Figure 7.1: Validation and debugging flow for parallel system models.

Moreover, even if the model is incorrect due to race conditions regarding shared variables, the simulation may actually succeed when using traditional sequential DE simulation. This might lead the designer to believe the model is correct, whereas in fact it is not. In other words, traditional simulation can lead to false validation of parallel models.

Discrete Event Simulation (DES)	**Synchronous Parallel Discrete Event Simulation (SPDES)**	**Out-of-order Parallel Discrete Event Simulation (Out-of-order PDES)**	**Race Condition Diagnosis for Parallel Models**
Regular SLDL compiler	Regular SLDL compiler Multithreading protection instrumentation	Regular SLDL compiler Multithreading protection instrumentation Static Conflict Analysis	Regular SLDL compiler Multithreading protection instrumentation Static Conflict Analysis
Sequential simulator	Multi-core Parallel simulator	Multi-core Out-of-order Parallel Simulator	Multi-core Parallel Simulator

Figure 7.2: Reusing essential tools from a parallel simulation infrastructure to diagnose race conditions in parallel models.

As shown in Fig. 7.2, traditional DE simulation uses a regular SLDL compiler to generate the executable model and then uses sequential simulation for its validation. In comparison, synchronous PDES also uses the regular SLDL compiler, but instruments the design with any

needed synchronization protection for true multi-threaded execution. An extended simulator is then used for multi-core parallel simulation.

The advanced out-of-order PDES approach, in contrast, uses the PDES compiler extended by an additional static code analyzer to generate potential conflict information. The corresponding scheduler in the simulator is also extended to utilize the compiled conflict information for issuing threads early and out of the order for faster simulation.

The infrastructure for advanced PDES motivates our idea for dynamic race condition diagnosis. The compiler for out-of-order PDES can analyze the design model statically to generate the needed information about potential data conflicts. Also, the synchronous PDES simulator allows threads in the same simulation cycle to run in parallel. Combining the two, we can therefore pass the conflict information generated by the compiler to the scheduler for dynamic race condition diagnosis among the parallel executing threads. As illustrated in Fig. 7.1b, the system designer can thus detect and obtain a diagnosis about parallel accesses to shared variables automatically and fix the problem quickly.

7.3 Automatic Race Condition Diagnosis

Fig. 7.3 shows the detailed tool flow for our proposed race condition diagnosis in parallel design models. Here, we are using a SpecC-based compiler and simulator framework.

The flow starts with an initial design model, i.e. *Design.sc*, as the input to the SLDL compiler. The compiler parses and checks the syntax of the model, builds an internal representation (extended abstract syntax tree) of the model, and then generates the C++ representation (*Design.cc* and *Design.h*) that can be compiled by the standard C++ compiler, e.g. *g++*, to produce the executable file for simulation.

In our proposed tool flow, we add the *Static Code Analyzer* to analyze the internal design rep-

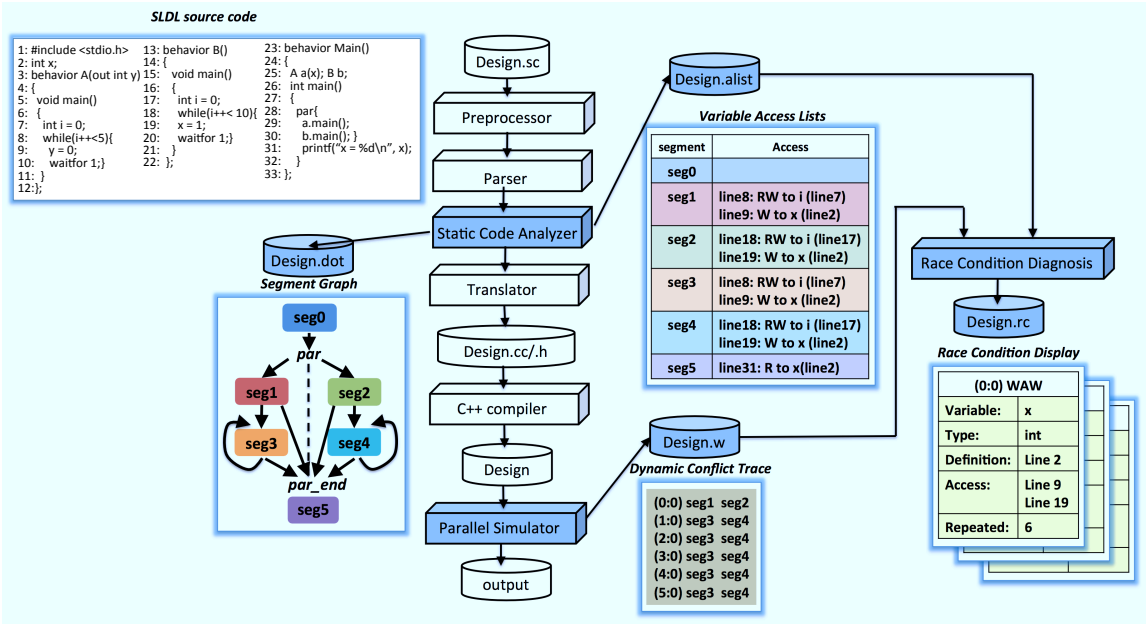


Figure 7.3: Tool flow for automatic race condition diagnosis among shared variables in parallel system models.

resentation for potentially conflicting accesses to shared variables during the parallel execution. The *Segment Graph* representing the parallel execution flow in the model and the *Variable Access List* are computed in this step. Using the segment graph and the variable access lists for the segments, the static analyzer constructs then the *Data Conflict Table* that lists any potential access conflicts in the design. This data conflict table is then passed to the simulator via instrumentation into the model.

The model is then validated by a *Parallel Simulator*, a synchronous PDES simulator extended with dynamic conflict checking. Whenever there are two threads running at the same simulation and delta time, the simulator checks the data conflict table for any conflicts between the segments the threads are running in. If there is a conflict, the simulator has detected a race condition and reports this in a *Dynamic Conflict Trace* file (*Design.w*). Note that this conflict checking is based on fast table look-ups which introduces very little simulation overhead.

After the simulation completes, the *Race Condition Diagnosis* tool processes the generated

Variable Access List and *Dynamic Conflict Trace* files and displays the detected race conditions to the designer. As shown in Fig. 7.3, critical parallel accesses to shared variables are listed with detailed information, including time stamp, access type, variable name and type, and line number and source file location where the variable is defined and where the access occurred. Since there may be many reports for the same race condition due to iterations in the execution, our tool combines these cases and lists the first time stamp and the number of repetitions.

Given this detailed information, the designer can easily find the cause of problems due to race conditions and resolve them.

7.4 Race Condition Elimination Infrastructure

As outlined above, we use (a) an advanced static code analysis to generate potential conflict information for a model, and then (b) a parallel simulation for dynamic conflict detection.

7.4.1 Static Code Analysis

Careful source code analysis at compile time is the key to identify potential access conflicts to shared variables.

During simulation, threads switch back and forth between the states of **RUNNING** and **WAITING**. Each time, threads execute different *segments* of their code. Access hazards exist when two segments contain accesses to the same variables. Except when two segments contain only read accesses (RAR), any write access creates a potential conflict (RAW, WAR, or WAW). These potential conflicts are called *race conditions* when they occur at the same simulation time, i.e. in the same simulation cycle.

Due to the (intended) non-deterministic execution in the simulator, race conditions may or may not lead to invalid values for the affected variables. Since this is often dangerous, race conditions

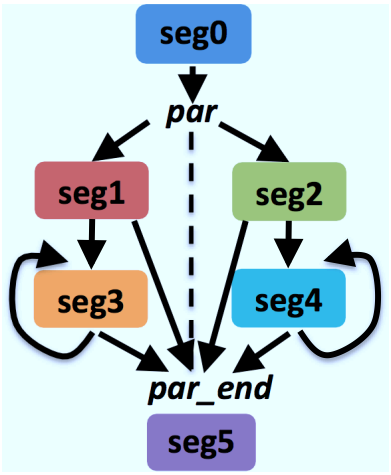
must be eliminated (or otherwise handled) for parallel design models to be safe.

```
1: #include <stdio.h>                    s0
2: int x;                                s0
3: behavior A(out int y)                 s1
4: {                                     s1
5:   void main()                         s1
6:   {                                   s1
7:     int i = 0;                        s1
8:     while(i++<5){                     s3 s1
9:       y = 0;                          s3 s1
10:      waitfor 1;}                     s3
11:  }                                   s3 s1
12:};                                    s3 s1

13: behavior B()                         s2
14: {                                    s2
15:   void main()                        s2
16:   {                                  s2
17:     int i = 0;                       s2
18:     while(i++< 10){                  s4 s2
19:     x = 1;                           s4 s2
20:     waitfor 1;}                      s4
21:   }                                  s4 s2
22:  };                                  s4 s2

17: behavior Main()                      s0
18: {                                    s0
19:   A a(x); B b;                       s0
20:   int main()                        s0
21:   {                                  s0
22:     par{                             
23:       a.main();                      s1
24:       b.main(); }                    s2
25:       printf("x = %d\n", x);         s5
26:     }                                s5
27: };
```

(a) SpecC source code



(b) Segment graph

seg	Access lists
0	
1	x(W)
2	x(W)
3	x(W)
4	x(W)
5	x(R)

(c) Access lists

seg	0	1	2	3	4	5
0	F	F	F	F	F	F
1	F	T	T	T	T	T
2	F	T	T	T	T	T
3	F	T	T	T	T	T
4	F	T	T	T	T	T
5	F	T	T	T	T	F

(d) Data conflict table

Figure 7.4: A parallel design example with a simple race condition.

For our proposed race condition analysis, we reuse the formal definitions of following terms (previously defined in Chapter 4):

- **Segment** seg_i: statements executed by a thread between two scheduling steps.
- **Segment Boundary** b_i: SLDL primitives which call the scheduler, e.g. *wait*, *wait-for-time*, *par*.

Here, segment boundaries b_i start segments seg_i. Thus, a directed graph is formed by the segments, as follows:

- **Segment Graph (SG)**: **SG**=(**V**, **E**), where **V** = $\{v \mid v_i$ is segment seg_i started by segment boundary $b_i\}$, **E**=$\{e_{ij} \mid e_{ij}$ exists if seg_j is reached after $seg_i\}$.

From the control flow graph of a design model, we can derive the corresponding segment graph (Section 4.3.1).

For example, Fig. 7.4(a) and (b) show a simple system model written in SpecC SLDL and its corresponding segment graph. Starting from the initial segment seg_0, two separate segments seg_1 and seg_2 represent the two parallel threads after the *par* statement in line 22. New segments are created after each segment boundary, such as *waitfor 1* (lines 10 and 20). The segments are connected following the control flow of the model. For instance, seg_3 is followed by itself due to the *while* loop in lines 8-10.

Given the segment graph, we next need to analyze the segments for statements with potentially conflicting variable assignments. We first build a variable access list for each segment, and then compile a conflict table that lists the potential conflicts between the N segments in the model:

- **Variable Access List:** $segAL_i$ is the list of the variables that are accessed in seg_i. Each entry for a variable in this list is a tuple of (*Var, AccessType*).
- **Data Conflict Table (CT[N,N])**:

$$CT[i,j] = \begin{cases} true & \text{if } seg_i \text{ has data conflict with } seg_j \\ false & \text{otherwise} \end{cases}$$

Note that $CT[N,N]$ is symmetric and can be built simply by comparing pairs of the variable access lists.

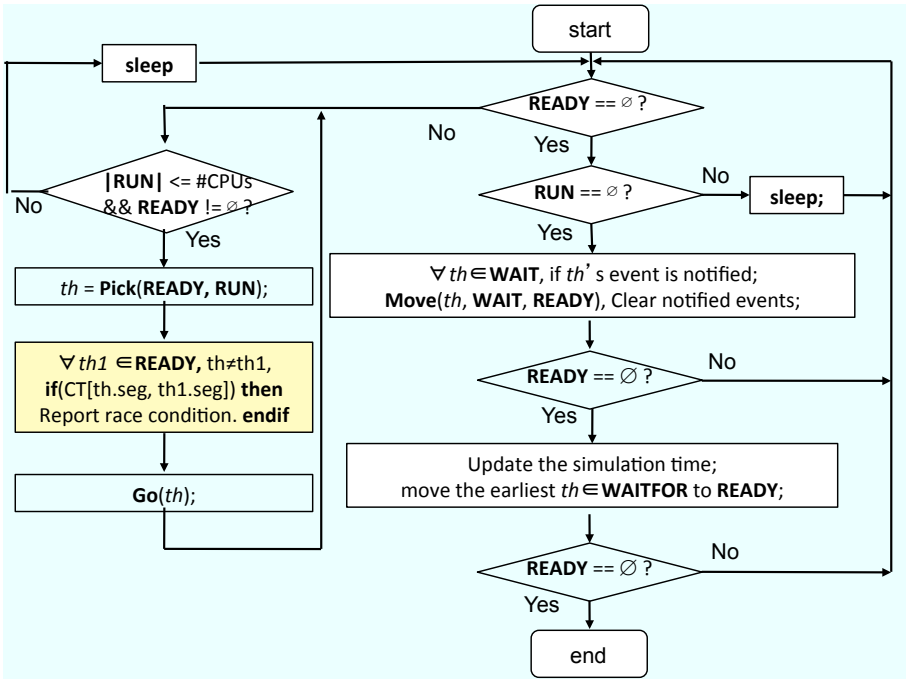


Figure 7.5: Parallel simulation algorithm with dynamic race condition checks.

7.4.2 Dynamic Race Condition Checking

We detect race conditions dynamically at runtime when the simulator schedules the execution of the threads. Fig. 7.5 shows the scheduling algorithm for synchronous PDES extended with the needed checking. The simulator performs the regular discrete event scheduling on the right side of Fig. 7.5 in order to deliver events and advance simulation time, following the usual *delta* and *time* cycles. On the left, the algorithm issues suitable threads to run in parallel as long as CPU cores are available.

Whenever it issues a thread for execution, the scheduler consults the data conflict table provided by the compiler in order to report detected race conditions. As shown on the left side of Fig. 7.5, the scheduler checks every thread for conflicts with the other threads that are **READY** to run. Again, we emphasize that these checks are simple table look-ups so that the overhead of race condition detection is minimal.

While we are using our parallel simulator for speed reasons here, we should note that the same detection approach can also be integrated in a traditional sequential DE simulator.

7.5 Experiments and Results

In this section, we report the results of using our approach on several embedded application examples. We describe how the tool set helped us to detect and diagnose a number of race conditions. Several reports on invalid parallel accesses to shared variables turned out to be the actual cause of simulation errors which we then could fix. Other reports could be ruled out for various reasons described below. All experiments have been performed on the same host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz.

Table 7.1: Experimental results on tool execution time for diagnosing race conditions in embedded multi-media applications

Applications	Compiler diagnosis [sec]		Simulator diagnosis [sec]		Race Condition Diagnosis
	off	on	off	on	
H.264 video decoder	12.51	4.05	18.51	19.56	1.04
H.264 video encoder	29.52	29.72	110.58	111.72	13.97
MP3 decoder (fixed)	1.14	1.19	4.34	4.44	0.08
MP3 decoder (floating)	3.63	3.89	13.82	13.87	0.34
GSM vocoder	3.90	4.00	1.46	1.50	0.07
JPEG encoder	4.01	4.09	1.54	1.56	0.02

7.5.1 Case study: A Parallel H.264 Video Decoder

During the development of our parallel H.264 video decoder model ([111], Section 3.6.1), we used the regular sequential SLDL simulator to validate the functionality. This showed 100% correct outputs. However, when we later used parallel simulation, the model *sometimes* ran through a few frames and then terminated with various assertion failures, or even terminated immediately with a segmentation fault. This was frustrating because such non-deterministic errors are very hard to debug.

Table 7.2: Experimental results on diagnosed race conditions in embedded multi-media applications

Embedded Applications	Lines of Code	#Vars/ #Trace Entries	Resolved Race Conditions	Unresolved Race Conditions
H.264 video decoder	40k	40 / 1201	3 resolved, localized to class scope 15 store const values to each frame, safe to share 1 structure accessed without overlap, safe 1 debugging value, temporarily used only 20 in channels, safely protected	0
H.264 video encoder	70k	68 / 712911	14 resolved, localized to stack variables 15 constant with current parameters, OK 1 identical in all parallel blocks, OK 1 array variable resolved by splitting the array 37 in channels, safely protected	0
MP3 decoder (fixed)	7k	7 / 82	7 in channels, safely protected	0
MP3 decoder (floating)	14k	13 / 75	12 in channels, safely protected 1 array variable resolved by splitting the array	0
GSM vocoder	16k	2 / 253	1 resolved, duplicated for parallel modules 1 resolved, localized to stack variable	0
JPEG encoder	2.5k	9 / 66	9 in channels, safely protected	0

We used the new race condition diagnosis tool to check the model, resulting in reports on 40 different shared variables. The model designer went through the variable list one by one to eliminate problems caused by race conditions. As reported in Table 7.1 and Table 7.2, half of the reported variables were defined in channels and therefore protected from access conflicts by the SpecC execution semantics (implicit locks). No change was necessary for these.

Another 15 variables were determined as storing values that are constant to each frame and thus can be shared safely when the decoder processes the frames in parallel. One report was about a complex structure that is actually accessed by parallel tasks only to non-overlapping members. Another variable was a global counter used purely for debugging. Again, no change to the model was necessary for these cases.

However, the remaining three of the reported variables actually caused the simulation problems. The model designer resolved the race conditions for them by relocating the variables from global to class scope. This way, each parallel unit has its own copy of these variables. As a result, the dangerous race conditions were eliminated and the model now simulates correctly also in the parallel simulator.

7.5.2 Case study: A Parallel H.264 Video Encoder

As a second large application example, we have converted the reference C source code of a H.264 encoder into a SpecC model ([112], Section 4.4.2). To allow parallel video encoding, we restructured the model for parallel motion estimation distortion calculation. When using the regular sequential simulator, the simulation result of the parallelized model matched the reference implementation. However, after we switched the simulator with a newly developed parallel simulator, the encoding result became inconsistent. That is, the encoding process finished properly, but the encoded video stream differed from time to time.

At the beginning of debugging, we had no idea about the cause of the encoding errors. Moreover, we were not even sure whether the problem was caused by the model itself or by our new parallel simulator. Literally thousands of lines of code, in both the model and the simulator, were in question.

At this point, the advanced static code analysis used in our out-of-order PDES simulator (Section 4.3.2) sparked the idea of using it to attack such debugging problems.

When we used this tool to analyze the H.264 video encoder model, we indeed found a total of 68 variables accessed in race conditions by the parallel motion estimation blocks. Specifically, the encoding malfunction was caused by read/write accesses to 14 global variables which stored the intermediate computation results in the parallelized behaviors. After localizing those global variables to local variables on the stack of the executing thread, the encoding result was correct, matching the output of the reference code.

For the remaining reported variables, as listed in Table 7.1 and Table 7.2, there was no immediate need to recode them. For example, variables which remain constant during the encoding process in our model and for our fixed parameters (for example *height_pad*), we decided to leave these unchanged.

7.5.3 Additional Embedded Applications

We also use the proposed tool set also to double-check embedded application models that had been developed earlier in-house based on standard reference code.

The first application is a fixed-point MP3 decoder model for two parallel stereo channels ([113]). As shown in Table 7.1 and Table 7.2, our diagnosis tool reports 7 shared variables that are subject to race conditions, out of a total of 82 conflict trace entries. We have looked at these variables and found that they are all member variables of channel instances which are protected by implicit locks for mutual exclusive accesses to channel resources. Thus, we successfully confirmed that this model is free of race conditions.

The second application is another parallel MP3 decoder model based on floating-point operations ([113]). Our diagnosis tool lists 9 shared variables out of 75 trace file entries. Eight of those variables are channel variables which are free from data hazards. The remaining variable, namely *hybrid_blc*, is an array of 2 elements. Each element is used separately by the two stereo channels, so there is no real conflict. We can resolve the race condition report by splitting this array into two instances for the two channels. Thus, this parallel model is also free of data hazards.

The third embedded application is a GSM Vocoder model whose functional specification is defined by the European Telecommunication Standards Institute (ETSI) ([114]). Only two variables are reported by our diagnosis tool for race condition risks. The first one, *Overflow*, is a Boolean flag used in primitive arithmetic operations. It can be resolved by replacing it with local variables on the stack of the calling thread. The second one, *old_A*, is an array which stores the previous results for an unstable filter. This variable is incorrectly shared by two parallel modules. We can resolve this situation by duplicating the variable so that each parallel instance has its own copy. We should note that these two bugs have been undetected for more than a decade in this in-house example.

The last application is the JPEG encoder for color images (Section 3.6.2). There are three variables reported as potential race conditions out of 253 entries in the trace log. Since all three are members of channel instances which are implicitly protected by locks, it is safe to have them in the parallel model.

In summary, Table 7.1 and Table 7.2 list all our experimental results, including the size of the models and the performance of the tools. While our application examples are fairly large design models consisting of several thousand lines of code, the overhead of race condition diagnosis is negligible for both compilation and simulation. Also, the diagnosis tool itself runs efficiently in less than a few seconds.

7.6 Conclusions

Writing well-defined and correct system-level design models with explicit parallelism is difficult. Race conditions due to parallel accesses to shared variables pose an extra challenge as these are often not exposed during simulation.

In this chapter, we propose an automatic diagnosis approach that enables the designer to ensure that a developed model is *free of race conditions*. The infrastructure of our proposed tool flow includes a compiler with advanced conflict analysis, a parallel simulator with fast dynamic conflict checking, and a novel race-condition diagnosis tool. This flow provides the designer with detailed race condition information that is helpful to fix the model efficiently when needed.

The proposed approach has allowed us to reveal a number of risky race conditions in existing embedded multi-media application models and enabled us to efficiently and safely eliminate these hazards. Our experimental results also show very little overhead for race condition diagnosis during compilation and simulation.

8 Conclusions

The large size and complexity of modern embedded systems poses a great challenge to design and validation. In order to address this challenge, system-level design usually starts with an abstract model of the intended system and follows different design methodologies to get the final implementation with the help of design automation tools. A well-defined system-level model contains the essential features such as structural and behavioral hierarchy, separate computation and communication, and explicit parallelism. Typically, system-level models are written in C-based System-level Description Languages (SLDLs), and rely on simulation to validate their functional correctness and estimate design metrics, including timing, power, and performance. Fast yet accurate simulation is highly desirable for efficient and effective system design.

The simulation kernel of the C-based SLDLs is usually based on Discrete Event (DE) simulation driven by events notifications and simulation time advancements. The existing reference simulators for SLDLs use the cooperative multithreading model to express the explicit parallelisms in ESL models. It is impossible to utilize the multiple computational resources that are commonly available in todays multi-core simulation hosts. Moreover, the discrete event execution semantics impose a total order on event delivery and time advances for model simulation. The global simulation cycle barrier is a significant impediment to parallel simulation.

In this work, we proposed a parallel simulation infrastructure for system-level models written in SLDLs. The infrastructure, including a scheduler for multithread parallel scheduling and a compiler for static code analysis, can effectively exploit the multiple computational resources

on multi-core simulation platforms for fast simulation, and increase model observability for system-level development.

8.1 Contributions

We summarize our contributions in the following sections.

8.1.1 A model of computation for system-level design

In Chapter 2, we have discussed the relationship between C-based system description languages and the abstract design models they describe. We've outlined the need for a model of computation behind the syntax of the description languages.

We proposed the *ConcurrenC* model of computation as a practical approach for abstract system modeling which fills the gap between the theoretical model of computation, such as KPN and SDF, and the practical SLDLs, such as SpecC and SystemC. *ConcurrenC* is a concurrent, hierarchical system model of computation with abstraction of both communication and computation. It can be expressed by both the SpecC and SystemC SLDLs for execution and validation. We emphasis the parallel execution semantics as a critical feature for system-level modeling and design.

A real-world driver application, H.264 decoder is used to demonstrate how the *ConcurrenC approach* matches the system modeling requirements.

8.1.2 A synchronous parallel discrete event simulator

While parallelism is usually explicitly expressed in system-level and multi-core computer platforms are commonly available nowadays, most of the reference simulation kernels for system-

level description languages do not support parallel simulation due to their implementation with respect to discrete event execution semantics.

In Chapter 3, we propose a kernel extension to the SpecC SLDL simulator for synchronous parallel discrete event simulation on shared-memory multi-core platforms. The extended simulation kernel has the following features:

- It can issue multiple threads in the same simulation cycle to run simultaneously during simulations.
- It uses the preemptive OS kernel-level threads and relies on the operating system to schedule the concurrent threads on the underlying CPUs in the multi-core simulation host.
- It supports an automatic channel locking scheme for proper synchronization and communication protection.
- It protects the synchronization and communication automatically through source code instrumentation by the SpecC compiler.

The extended simulator is scalable on multi-core computer platforms, and is transparent for the designer to use without any model modification to protect parallel simulation.

We conduct case studies on two well balanced embedded applications, a H.264 video decoder and a JPEG Encoder applications, respectively. Our experimental results demonstrate a significant reduction in simulation run time for transaction level models at various levels of abstraction.

8.1.3 An advanced parallel discrete event simulation approach

The discrete event execution semantics imposes a strict total order on event handling and timing advances. Although parallelism are usually explicitly expressed in the model and mapped to

working threads accordingly, very limited of them can truly run concurrently during simulation. The global simulation time is the major barrier which prevents multiple working threads from running in parallel.

In Chapter 4 and Chapter 5, we propose the out-of-order parallel discrete event simulation approach to address the issue of limited simulation parallelism in the synchronous parallel discrete event simulation. This novel approach contributes in the following ways:

- It breaks the global simulation cycle barrier by locating them into each thread so as to allow threads in different simulation cycles to run in parallel.
- It allows as many working threads as possible to run concurrently so as to maximize the multi-core CPU utilization.
- The simulation semantics and timing accuracy are fully preserved with the help of the static code analysis on the design models and the dynamic conflict detection during simulation.
- The optimized static code analysis using the idea of instance isolation can reduce the number of false conflicts. It also helps to reduce the analysis complexity.
- The optimized scheduling using prediction information can reduce the number of false dynamic conflicts, and therefore help to increase simulation parallelism.

We demonstrate the benefits and cost of this out-of-order approach on a set of experiment examples, including three highly parallel benchmark applications and six embedded applications. Experimental results show that, with only a small increase in compile time, our simulator is significantly faster than the traditional single-threaded reference implementation, as well as the synchronous parallel simulator for generic system-level models with imbalanced timing and workload.

8.1.4 An infrastructure for increasing modeling observability

Writing correct system-level model is difficult since it involves parallelism insertion into the reference sequential code. It is a tedious and error-prone process to build parallel system-level models, and is also very difficult to catch bugs caused by parallel race conditions by using existing development tools.

In Chapter 7, we propose a tool flow to increase the observability in parallel models. In particular, the infrastructure of the proposed tool flow includes a compiler with advanced conflict analysis, a parallel simulator with fast dynamic conflict checking, and a novel race-condition diagnosis tool. It can provide the designer with detailed race condition information that is helpful to fix the model efficiently when needed.

The proposed approach has allowed us to reveal a number of risky race conditions in our in-house embedded multi-media application models, and enabled us to efficiently and safely eliminate these hazards. Our experimental results also show very little overhead for race condition diagnosis in both compilation and simulation.

8.2 Future Work

Related future research work is worth pursuing in the following areas:

8.2.1 Model Parallelization

Serial applications cannot take advantage of parallel simulation. Therefore, the Grand Challenge for fast parallel simulation remains in the problem of how to expose the parallelism in system-level models.

Constructing explicit parallelism in system-level models will not only help to achieve the goal of

fast simulation, but is also highly desirable from the design perspective with respect to SW/HW partition, architecture mapping, etc. While great research efforts in the compiler community have shown that explicit parallelization is almost infeasible to get in a fully automatic way, it is promising to involve the developers, who have intelligence and knowledge, in the loop and assist them with the automatic analysis and optimization tools to achieve better parallelization results.

8.2.2 Multithreading Library Support

As the infrastructure for parallel simulation, the performance of the underlying multithreading libraries is critical for efficient validation on multi-core simulation hosts. Preemptive multi-threading libraries, such as Posix Threads, have heavy overheads in terms of context switching, thread creation and deletion, etc. Research work is desirable to reduce the threading overhead on the emerging computer architectures, such as many core systems and heterogeneous mobile platforms.

8.2.3 Extension to the SystemC SLDL

The work in this dissertation is equally applicable to the SystemC language, which is the industry standard for system-level design. A front-end tool which can build the C++ abstract syntax tree and extract the SystemC semantics for system-level features is needed for the purpose of model analysis and design automation. Research study is also needed to ensure the SystemC parallel execution semantics (Section 3.2.1).

8.2.4 Parallel Full System Validation

The system validation phase usually expands through the whole design life cycle of a product as it includes system-level, RTL, and post-silicon stages [116, 117]. Validation speed is very

critical to products' time to market.

For system-level validation, simulation performance of the models at lower abstraction levels is the real bottleneck in todays system design flows. The simulation can take several days or weeks to run for industrial applications. Efficient parallel full system simulation is highly desirable for fast and low cost system validations.

8.3 Concluding Remarks

In conclusion, the work presented in this dissertation provides an advanced parallel simulation infrastructure for efficient and effective system-level model validation and development. It helps the embedded system designers to build better products in a shorter development cycle.

Bibliography

[1] Peter Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems Series*. Springer Publishing Company, Incorporated.

[2] Embedded System Definition. http://www.linfo.org/embedded_system.html. The Linux Information Project.

[3] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, April 2009.

[4] Preeti Ranjan Panda and Nikil D. Dutt. Memory Architectures for Embedded Systems-On-Chip. In *the 9th International Conference on High Performance Computing*, pages 647–662, 2002.

[5] Wilfried Elmenreich and Richard Ipp. Introduction to TTP/C and TTP/A, 2003.

[6] Road Vehicles – Controller area network (CAN) – Part 1: Data Link Layer and Physical Signaling, 2003. The International Organization for Standardization (ISO).

[7] ARINC Specification 629-3 : IMA Multi-transmitter Databus, 1994. Airlines Electronic Engineering Committee.

[8] Josef Berwanger, Christian Ebner, Anton Schedl, Ralf Belschner, and et al. Flexray - the communication system for advanced automotive control systems, technical paper 2001-01-0676. Technical report, Society of Automotive Engineers (SAE), 2001.

[9] AMBA Specification home page. http://www.arm.com/products/system-ip/amba. ARM Holdings plc.

[10] TTEthernet - A Powerful Network Solution for All Purposes, 2007. TTTech Computertechnik AG.

[11] Wolfgang Ecker, Wolfgang Mueller, and Rainer Doemer. *Hardware-dependent Software - Principles and Practice*. Springer Publishing Company, Incorporated.

[12] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 27(10):1701–1713, 2008.

[13] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 2009.

[14] Alberto Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[15] International Semiconductor Industry Association. International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net, 2004.

[16] International Semiconductor Industry Association. International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net, 2007.

[17] Rainer Doemer. *System-level Modeling and Design with the SpecC Language*. PhD thesis, Universität Dortmund, April 2000.

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

[19] Daniel D. Gajski and Robert H. Kuhn. Guest Editors Introduction: New VLSI Tools. *IEEE Computer*, December 1983.

[20] International Semiconductor Industry Association. International Technology Roadmap for Semiconductors (ITRS). http://www.itrs.net, 2011.

[21] ARM Holdings plc. The ARM Processors. http://www.arm.com/products/processors/index.php.

[22] Imagination Technologies. The MIPS Technologies Group plc. http://www.imgtec.com/mips/mips-processor.asp.

[23] Areoflex Gaisler Research. The LEON Processor. http://gaisler.com/index.php/products/processors.

[24] Freescale Semiconductor, Inc. The Freescale ColdFire Processor. http://www.freescale.com/webapp/sps/site/homepage.jsp?code=PC68KCF.

[25] Analog Devices, Inc. The Blackfin Processor. http://www.analog.com/en/processors-dsp/blackfin/products/index.html.

[26] Xilinx, Inc. The MicroBlaze Processor. http://www.xilinx.com/tools/microblaze.htm.

[27] Daniel D. Gajski. Nisc: The ultimate reconfigurable component. Technical Report CECS-TR-03-28, Center for Embedded Computer Systems, University of California, Irvine, October 2003.

[28] IEEE Standard VHDL Language Reference Manual. *IEEE Standard 1076-2008 (Revision of IEEE Standard 1076-2002)*, 2009.

[29] Frank Vahid and Roman Lysecky. *VHDL For Digital Design*. John Wiley, 2007.

[30] IEEE Standard Verilog Hardware Description Language. *IEEE Standard 1364-2001*, 2001.

[31] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Springer Publishing Company, Incorporated, 2002.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[33] Bruce Eckel. *Thinking in C++*. Prentice Hall, 1995.

[34] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[35] Stephen A. Edwards. Design Languages for Embedded Systems. Technical report, 2003.

[36] Daniel D. Gajski, Jianwen Zhu, Rainer Doemer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[37] Rainer Doemer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[38] Thorsten Groetker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[39] David C. Black and Jack Donovan. *SystemC: From the Ground Up*. Springer Publishing Company, Incorporated, 2005.

[40] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[41] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Standard 1800-2009*, 2009.

[42] Herbert Schildt. *The annotated ANSI C Standard American National Standard for Programming Languages - C: ANSI/ISO 9899-1990*. Osborne/McGraw-Hill, 1990.

[43] Jianwen Zhu and Daniel D. Gajski. Compiling SpecC for simulation. In *Proceedings of*

the Asia and South Pacific Design Automation Conference (ASPDAC), Yokohama, Japan, January 2001.

[44] Richard E. Nance. A History of Discrete Event Simulation Programming Languages, TR-93-21. Technical report, Department of Computer Science, Virginia Polytechnic Institute and State University, 1993.

[45] Rainer Doemer. The SpecC Internal Representation (SIR Manual V2.2.0), Internal Technical Report. Technical report, Center for Embedded Computer Systems, University of California, Irvine, 2005.

[46] Rainer Doemer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D. Gajski. System-on-chip Environment: A SpecC-based Framework for Heterogeneous MPSoC Design. *EURASIP Journal on Embedded Systems*, 2008(3):1–13, 2008.

[47] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. A Survey of Multicore Processors. *IEEE Signal Processing Magazine*, 26(6):26–37, November 2009.

[48] David Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7):28–32, July 2010.

[49] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 2009.

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

[51] Posix Threads Programming. https://computing.llnl.gov/tutorials/pthreads/.

[52] OpenMP. https://computing.llnl.gov/tutorials/openMP/.

[53] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Dongarra Jack. *MPI: The Complete Reference*. MIT Press, 1995.

[54] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, Red Hat, Inc., 2005.

[55] Aaron Cohen and Mike Woodring. *Win32 Multithreaded Programming*. O'Reilly Media, 1997.

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

[57] The Clik Project. http://supertech.csail.mit.edu/cilk/.

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

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

[60] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[61] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[62] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. Multithreading in Java: Performance and Scalability on Multicore Systems. *IEEE*, 60(11):1521–1534, November 2011.

[63] K.Mani Chandy and Jayadev Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, September 1979.

[64] Richard Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

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

[66] Kai Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele. Scalably Distributed SystemC Simulation for Embedded Applications. In *International Symposium on Industrial Embedded Systems, 2008. SIES 2008.*, pages 271–274, June 2008.

[67] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, October 2003.

[68] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Proceedings of the Design Automation Conference (DAC)*, 2011.

[69] Andreas Gerstlauer. Host-Compiled Simulation of Multi-Core Platforms. In *Proceedings of the International Symposium on Rapid System Prototyping (RSP), Washington, DC, June 2010.*

[70] SystemC TLM-2.0. http://www.accellera.org/downloads/standards/systemc/tlm.

[71] Scott Sirowy, Chen Huang, and Frank Vahid. Online SystemC Emulation Acceleration. In *Proceedings of the Design Automation Conference (DAC)*, 2010.

[72] Mahesh Nanjundappa, Hiren D. Patel, Bijoy A. Jose, and Sandeep K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2010.

[73] Rohit Sinha, Aayush Prakash, and Hiren D. Patel. Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[74] Shubhendu Mukherjee, Steven Reinhardt, Babak Falsafi, Mike Litzkow, Mark Hilland David Wood, Steven Huss-Lederman, and James Larus. Wisconsin Wind Tunnel II:

a fast, portable parallel architecture simulator. *IEEE Concurrency*, 8(4):12–20, October-December 2000.

[75] Dukyoung Yun, Sungchan Kim, and Soonhoi Ha. A Parallel Simulation Technique for Multicore Embedded Systems and Its Performance Analysis. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 31(1):121–131, January 2012.

[76] Weiwei Chen and Rainer Doemer. ConcurrenC: A New Approach towards Effective Abstraction of C-based SLDLs. In *Embedded System Design: Topics, Techniques and Trends*, 2009.

[77] Embedded System Environment. http://www.cecs.uci.edu/~ese/.

[78] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Ji Gong. *Specification and Design of Embedded Systems*. Prentice Hall, 1994.

[79] Stephen A. Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, 85(3), March 1997.

[80] MoC wikipedia. http://en.wikipedia.org/wiki/Model_of_computation.

[81] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 17(12), December 1998.

[82] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Electrical Engineering and Computer Science, University of California, Berkeley, December 1995.

[83] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[84] Greet Bilsen, Marc Engels, Rudy Lauwereins, and J. A. Peperstraete. Cyclo-Static Dataflow. *Proceedings of the IEEE*, 44(2):397–408, Febuary 1996.

[85] Alain Girault, Bilung Lee, and Edward A. Lee. Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems (TCAD)*, 18(6), June 1999.

[86] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya. Parameterized Dataflow Modeling of DSP Systems. In *International Conference on Acoustics, Speech, and Signal Processing*, June 2000.

[87] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, Electrical Engineering and Computer Science, University of California, Berkeley, 1993.

[88] Gang Zhou. Dynamic Dataflow Modeling in Ptolemy II, Technical Memorandum No. UCB/ERL M05/2. Technical report, University of California, Berkeley, December 2004.

[89] Setphen A. Edwards and Olivier Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. *IEEE Transactions on VLSI Systems*, 14(8):854–867, 2006.

[90] James L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[91] Joint Video Team of ITU-T and ISO/IEC JTC 1. *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 — ISO/IEC 14496-10 AVC)*. Document JVT-G050r1, 2003.

[92] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[93] Rainer Doemer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-Core Parallel

Simulation of System-Level Description Languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2011.

[94] Weiwei Chen, Xu Han, and Rainer Doemer. ESL Design and Multi-Core Validation using the System-on-Chip Environment. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2010.

[95] Weiwei Chen and Rainer Doemer. An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolatio. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[96] Weiwei Chen, Xu Han, and Rainer Doemer. Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment. *IEEE Design and Test of Computers*, 28(3):20–31, May/June 2011.

[97] Wolfgang Mueller, Rainer Doemer, and Andreas Gerstlauer. The formal execution semantics of SpecC. In *Proceedings of the International Symposium on System Synthesis*, pages 150–155, 2002.

[98] Masahiro Fujita and Hiroshi Nakamura. The standard SpecC language. In *Proceedings of the 14th international symposium on Systems synthesis (ISSS)*, 2001.

[99] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560 –576, July 2003.

[100] H.264/AVC JM Reference Software. http://iphome.hhi.de/suehring/tml/.

[101] Lucai Cai, Junyu Peng, Chun Chang, Andreas Gerstlauer, Hongxing Li, Anand Selka, Chuck Siska, Lingling Sun, Shuqing Zhao, and Daniel D. Gajski. Design of a JPEG Encoding System. Technical Report ICS-TR-99-54, Information and Computer Science, University of California, Irvine, November 1999.

[102] Weiwei Chen, Xu Han, and Rainer Doemer. Out-of-order Parallel Simulation for ESL Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

[103] Weiwei Chen and Rainer Doemer. Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2013.

[104] Xu Han, Yasaman Samei, and Rainer Doemer. System-Level Modeling and Refinement of a Canny Edge Detector. Technical Report CECS-TR-12-13, Center for Embedded Computer Systems, University of California, Irvine, November 2012.

[105] Che-Wei Chang and Rainer Doemer. System Level Modeling of a H.264 Video Encoder. Technical Report CECS-TR-11-04, Center for Embedded Computer Systems, University of California, Irvine, June 2011.

[106] Weiwei Chen, Xu Han, , Che-Wei Chang, and Rainer Doemer. Advances in Parallel Discrete Event Simulation for Electronic System-Level Design. *IEEE Design and Test of Computers*, 2013.

[107] time(1) - Linux man page. http://linux.die.net/man/1/time.

[108] Russel Sandberg, David Golgberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Innovations in internetworking. chapter Design and implementation of the Sun network filesystem, pages 379–390. Artech House, Inc., 1988.

[109] Weiwei Chen, Che-Wei Chang, Xu Han, and Rainer Doemer. Eliminating Race Conditions in System-LevelModels by using Parallel Simulation Infrastructure. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2012.

[110] Pramod Chandraiah and Rainer Doemer. Computer-aided recoding to create structured

and analyzable system models. *ACM Transactions in Embedded Computing Systems*, 11S(1):23:1–23:27, June 2012.

[111] Xu Han, Weiwei Chen, and Rainer Doemer. A Parallel Transaction-Level Model of H.264 Video Decoder. Technical Report CECS-TR-11-03, Center for Embedded Computer Systems, University of California, Irvine, June 2011.

[112] Che-Wei Chang and Rainer Doemer. System Level Modeling of a H.264 Video Encoder. Technical Report CECS-TR-11-04, Center for Embedded Computer Systems, University of California, Irvine, June 2011.

[113] Pramod Chandraiah and Rainer Doemer. Specification and design of an MP3 audio decoder. Technical Report CECS-TR-05-04, Center for Embedded Computer Systems, University of California, Irvine, May 2005.

[114] Andreas Gerstlauer, Shuqing Zhao, Daniel D. Gajski, and Arkady M. Horak. Design of a GSM vocoder using SpecC methodology. Technical Report ICS-TR-99-11, Information and Computer Science, University of California, Irvine, March 1999.

[115] Xu Han, Weiwei Chen, and Rainer Doemer. Designer-in-the-Loop Recoding of ESL Models using Static Parallel Access Conflict Analysis. In *Proceedings of the workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2013.

[116] Validation, verification and certification of embedded systems. Technical report, NATO Research and Technology organization, 2005.

[117] Frank Schirrmeister. System-level market trends. Technical report, Synopsys Interoperability Forum, 2010.

Index

Accuracy, 7

Behavioral Domain, *see* The Y-Chart

Combined Conflict Prediction Table, 118

Computer Aided Design, 8

ConcurrenC Model of Computation, 44

Cooperative multi-threading in SystemC, 55

Creating Parallel System Models, 138

Current Time Advance Table, 90

Data Conflict Table, 87

Data Conflict Table for Predictions, 113

Discrete Event Simulation, 18, 52

- Scheduler, 19
- Simulation Cycles, 18
- Working Threads, 19

Discrete Event Simulation Data Structures and Operations, 52

Dynamic Race Condition Checking, 146

Electronic System-Level (ESL) Design, 11

Embedded Computer Systems, 1

Embedded Hardware System, 1, 4

Embedded Operating Systems, 3

Embedded Software System, 2, 4

Estimation, 13

Event Notification Table, 89

Event Notification Table for Predictions, 116

False Conflicts, 98

Hardware Description Languages (HDLs), 14

Hardware-Depend Software (HdS), 3

Host-compiled Simulation, 38

Implementation Models, 13

In-order Discrete Event Execution, 74

Instance Isolation, 100

International Technology Roadmap for Semiconductors (ITRS), 4

Levels of Abstraction, 6

Models, 6
Models of Computation (MoCs), 42
Dataflow Graph (DFG), 43
Dateflow Process Network(DFPN), 42
Finite State Machine with Datapath (FSMD), 43
Kahn Process Network(KPN), 42
Petri Net, 43
Program State machine (PSM), 43
Software/hardware integration medium (SHIM), 43
Synchronous Dataflow (SDF), 42
Transaction-Level modeling (TLM), 43
Modified Segment Adjacency Matrix, 114
Moore's Law, 4
Multi-core technology, 36
Multithreaded programming, 36

Next Time Advance Table, 90

On-the-fly Instance Isolation, 102
Optimized Scheduling Using Predictions, 109
Data Hazard Prediction, 113
Event Hazard Prediction, 116
Time Hazard Prediction, 116
Optimized Static Code Analysis, 104
Out-of-order Parallel Discrete Event Simulation, 76
Dynamic Conflict Detection, 91
Notations, 76
Simulation Event Tuple, 77
Simulation Invariants, 78
Simulation Queues, 77
Simulation State Transition, 79
Simulation States, 78
Simulation Time Tuple, 77
Scheduling Algorithm, 79
Static Conflict Analysis, 81

Parallel Benchmark Models, 125
Parallel Fibonacci Calculation, 126
Parallel Fibonacci Calculation with Timing Information, 127
Parallel Floating-point Multiplications, 125
Parallel Discrete Event Simulation (PDES), 37
Conservative PDES, 37
Distributed Parallel Simulation, 37
Optimistic PDES, 37
Physical Domain, *see* The Y-Chart
Pre-emptive multi-threading in SpecC, 56

Race Condition Diagnosis, 141
Real-time Operating System (RTOS), 3
Recoding, 138

Refinement, 12
Architecture refinement, 12
Backend Tasks, 13
Communication refinement, 13
Network refinement, 13
Scheduling refinement, 12
Register-Transfer Level (RTL) Design, 40

Segment, 81
Segment Adjacency Matrix, 113
Segment Boundary, 81
Segment Graph, 82
Shared Variables and Race Conditions, 139, 143
Simulation, 11, 17
Continuous Simulation, 17
Discrete Event Simulation, 17
Monte Carlo Simulation, 17
SLDL Multi-threading Semantics, 55
Software Languages, 15
Specification, 11
Static Conflict Analysis, 86
Data Hazards, 87
Event Hazards, 89
SLDL Variables, 88
Time Hazards, 89
Direct Timing Hazard, 90
Indirect Timing Hazard, 91
Structural Domain, *see* The Y-Chart
Synchronous Parallel Discrete Event Simulation, 58
Channel Locking Scheme, 62
Code Instrumentation for Communication Protection, 63
Multi-Core Simulation Implementation Optimization, 66
Protecting Communication, 60
Protecting Scheduling Resources, 59
Synchronization for Multi-Core Parallel Simulation, 59
System Design Challenge, 4
System-level Description Languages (SLDLs), 16
System-level Design, 4, 5
System-level Design Methodologies, 8
Bottom-up Design, 8
Meet-in-the-middle Design, 9
Top-down Design, 8
System-level Model of a DVD Player, 93
System-level Model of a H.264 Video Decoder, 49, 67, 95, 121, 128, 147
System-level Model of a H.264 Video Encoder, 121, 130, 149

System-level Model of a JPEG Image Encoder, 72, 94, 127
System-level Model of a MP3 Audio Decoder, 130
System-level Model of a Video Edge Detector, 120, 129
System-on-Chip (SoC), 2

Temporal Decoupling, 38
The SpecC Language, 24
Behavioral Hierarchy, 25
Concurrent Execution, 25
Finite State Machine (FSM) Execution, 26
Parallel Execution, 27
Pipelined Execution, 27
Sequential Execution, 25, 26
Communication, 28
Exception Handling, 29
Abortion, 29
Interrupt, 29
Exporter, 30
Importer, 30
Library Support, 30
Persistent Annotation, 30
Refinement Tools, 30
SpecC Internal Representation (SIR), 30
Structural Hierarchy, 24
Synchronization, 28
Timing, 29
Exact Timing, 29
Timing Constraints, 29
Types, 25
The System-on-Chip Environment (SCE), 32
Architecture Exploration, 34
Communication Synthesis, 34
Network Exploration, 34
RTL Synthesis, 35
Scheduling Exploration, 34
Software Synthesis, 35
The SystemC Language, 31
The Y-Chart, 7
Time Advance Table for Predictions, 116
Transaction-Level Modeling, 38

Validation, 11
Verification, 11

Well-defined System Model, 137