UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Transaction-Level Modeling of Deep Neural Networks for Efficient Parallelism and Memory Accuracy

Permalink
https://escholarship.org/uc/item/8zt5p9kh

Author
Malekzadeh Arasteh, Emad

Publication Date
2022

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Transaction-Level Modeling of Deep Neural Networks for Efficient Parallelism and Memory Accuracy

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Emad Malekzadeh Arasteh

Dissertation Committee:
Professor Rainer Dömer, Chair
Professor Fadi Kurdahi
Professor Ian G. Harris

2022

© 2022 Emad Malekzadeh Arasteh

DEDICATION

To my family

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

LIST OF ALGORITHMS **x**

ACKNOWLEDGMENTS **xi**

VITA **xii**

ABSTRACT OF THE DISSERTATION **xvi**

1 Introduction **1**
1.1 Electronic System Level and Transaction Level Modeling 1
1.2 Parallel SystemC Simulation 3
1.3 Memory Bottleneck 4
1.4 Deep Learning and Convolutional Neural Networks 5
1.4.1 GoogLeNet Structure 6
1.4.2 Single Shot MultiBox Detector Structure 7
1.5 Related Work 9
1.6 Goals 12

2 Untimed SystemC Model of DNN **16**
2.1 Introduction 16
2.2 System-level Modeling Framework 17
2.3 DNN Layer Implementation 19
2.4 Validation by Simulation 22
2.5 Experimental Results 22
2.5.1 Performance Setup 23
2.5.2 Simulation Results 24
2.5.3 Analysis 26
2.6 Conclusion 27

3 Improving Parallelism in System Level Models 29
3.1 Introduction . . . 30
3.2 Parallelism in TLM-1 Modeling . . . 31
3.2.1 TLM-1 Modeling of DNNs . . . 31
3.2.2 Channel type . . . 32
3.2.3 Buffering scheme . . . 32
3.3 Parallelism in TLM-2.0 Modeling . . . 36
3.3.1 TLM-2.0 Modeling of DNNs . . . 36
3.3.2 Feed-forward events mechanism . . . 37
3.3.3 Back-pressure events mechanism . . . 39
3.4 Parallelism Direction . . . 40
3.5 Experimental Measurements and Results . . . 42
3.5.1 Simulation Setup . . . 42
3.5.2 Simulation Results . . . 43
3.6 Conclusion . . . 46

4 Fast Loosely-Timed System Models with Accurate Memory Contention 48
4.1 Introduction . . . 49
4.2 TLM-2.0 Loosely-timed (LT) Model . . . 52
4.2.1 Memory . . . 53
4.2.2 Computation . . . 54
4.2.3 Interconnect . . . 56
4.3 TLM-2.0 Approximately-timed (AT) Model . . . 58
4.4 TLM-2.0 Loosely-timed Contention-aware (LT-CA) Model . . . 60
4.5 Transaction Level Model Generator . . . 62
4.6 Experiments and Results . . . 64
4.6.1 Simulation setup . . . 65
4.6.2 Memory load estimation . . . 66
4.6.3 Comparison of LT, AT, and LT-CA models . . . 67
4.6.4 Contention visualization . . . 71
4.7 Conclusion . . . 75

5 Toward Cycle Accuracy in System-Level Memory Modeling 77
5.1 Introduction . . . 77
5.2 TLM-2.0 AT Model with Cycle-Accurate Memory . . . 79
5.3 Enhanced Delay Estimation in TLM-2.0 LT-CA Model . . . 82
5.4 Processing-in-Memory (PIM) with Local Memories . . . 85
5.5 Experiments and Results . . . 88
5.5.1 Simulation setup . . . 88
5.5.2 DRAMSys . . . 89
5.5.3 LT, LT-CA and DDR . . . 91
5.5.4 Single Shot MultiBox Detector (SSD) . . . 97
5.6 Conclusion . . . 99

6 Conclusion **101**
6.1 Contributions . . . 101
6.1.1 DNN System Level Modeling with Exposed Parallelism . . . 102
6.1.2 Improvement of System Level Parallelism . . . 102
6.1.3 Fast Loosely-Timed System Models with Accurate Memory Contention 103
6.1.4 Cycle Accuracy in System-Level Memory Modeling . . . 103
6.2 Future Work . . . 104
6.2.1 Extend the TLM Modeling Framework . . . 104
6.2.2 Explore Local Memory Topologies to Minimize Cost and Contention . 105
6.2.3 Programming Models for Processing-In-Memory . . . 105
6.2.4 New Memories for Compute-Memory Integration . . . 105

Bibliography **106**

LIST OF FIGURES

Page

1.1 Top-down ESL design methodology 2
1.2 Architecture of LeNet-5, a CNN for digits recognition [39] 6
1.3 GoogLeNet network with all the bells and whistles [62] 7
1.4 Single Shot MultiBox Detector (SSD) network schematic 9

2.1 Convolution layer 21
2.2 Top-level test bench 23
2.3 Speedup comparison on different platforms based on the source of parallelism 28

3.1 Simulator parallelism, model parallelism and simulation speedup forms a 3-dimensional space[2] 30
3.2 Inception module in GoogLeNet 34
3.3 TLM-1 model diagram of inception module in GoogLeNet 35
3.4 TLM-2.0 model diagram of GoogLeNet 37
3.5 (a) TLM-2.0 feed-forward model connections (b) TLM-2.0 back-pressure events connections 38
3.6 Communication mechanism versus number of available buffers in TLM models 41
3.7 Simulation speedup for different parallel simulation modes on a 16-core host 46

4.1 Transaction Level Modeling (TLM) of GoogLeNet DNN with focus on exposing parallelism (Phase I) and memory contention (Phase II) 50
4.2 Deep neural network (DNN) transaction level modeling (TLM) exploration framework 50
4.3 TLM-2.0 LT model module connections 53
4.4 (a) TLM-2.0 LT model with an interconnect (b) TLM-2.0 LT model with an interconnect and multiple memories 57
4.5 Internal structure of the `netspec` TLM generator for transaction-level design space exploration 64
4.6 Total memory accesses for pass of 100 images in GoogLeNet 67
4.7 Simulated time of GoogLeNet for a pass of 100 images across 4 computational capacities and 4 memory latencies (a) Loosely-timed (b) Loosely-timed contention-aware (c) Contention ratio 70
4.8 Simulated time of the inception_3a module without (left) and with (right) contention (1000 GFLOPS, 1ns memory latency) 72

4.9 Simulated time of all 9 inception modules without contention (1000 GFLOPS, 1ns memory latency) . . . 74
4.10 Simulated time of all 9 inception modules with contention for image #0 (1000 GFLOPS, 1ns memory latency) . . . 74
4.11 Simulated time of all 9 inception modules with contention for image #75 (1000 GFLOPS, 1ns memory latency) . . . 75

5.1 TLM-2.0 AT model with an interconnect and DRAM subsystem implemented by DRAMSys4.0 [32] framework . . . 80
5.2 TLM-2.0 LT-CA model with local interconnects and private memories . . . 85
5.3 (a) Inception module input local memory structure (multi-consumer) (b) Inception module output local memory structure (multi-producer) . . . 87
5.4 Strip plot of memory transaction delays for GoogLeNet AT model with DRAMSys DDR3-1600 (1000 GFLOPS) . . . 90
5.5 Histograms of memory transaction delays for GoogLeNet AT model with DRAMSys DDR3-1600 in partitions (1000 GFLOPS) . . . 91
5.6 Simulated time of GoogLeNet for a pass of 100 images across 4 computational capacities and bandwidth utilization factors β sweeping (0.1, 1) with a step of 0.1 (a) LT (b) LT-CA (in milliseconds) for memory running on 800MHz, burst length of 8 and data width of 8B . . . 95
5.7 Contention ratio of GoogLeNet for a pass of 100 images with memory bandwidth utilization factor of 10%-100% across 4 computational capacities . . . 96
5.8 Simulated time of LT and LT-CA models of GoogLeNet for a pass of 100 images across memory bandwidth utilization factor and categorized for each computational capacity (a) 1000GFLOPS (b) 100GFLOPS (c) 10GFLOPS (d) 1GFLOPS (in milliseconds) . . . 97
5.9 From left: AT delay histogram, LT-CA delay histogram and kernel density estimation of LT-CA delays versus a normal distribution N(LT-CA mean, LT-CA variance) for (a) DDR3-1600 and (b) DDR4-1866 . . . 98
5.10 Simulated time of fc7 branch without (left) and with (right) contention (1000 GFLOPS, 0.625ns memory latency) . . . 99

LIST OF TABLES

Page

1.1 GoogLeNet layer summary 7
1.2 Single Shot Detector layer summary 8

2.1 Simulation platforms specification 24
2.2 Measurement results on 4-core host (HTT off) 24
2.3 Measurement results on 16-core host (HTT off) 25
2.4 Measurement results on 8-core host (HTT on) 25
2.5 Measurement results on 32-core host (HTT on) 25

3.1 TLM models summary 41
3.2 Platform specification 43
3.3 Measurements of elapsed time for parallel simulations (color scale red-to-green means slow-to-fast) 44
3.4 Measurements of elapsed time for SEQ execution (color scale green-to-red means increasing workload) 45

4.1 Table of parameterized features for `netspec` model generation and component customization 65
4.2 GoogLeNet total memory footprint 68
4.3 Total simulated time of GoogLeNet for different computational capacities and memory latencies (in seconds) 69
4.4 Total simulator run-time of GoogLeNet for different computational capacities and memory latencies (in seconds) on a 32-core host 71

5.1 DRAM memory specifications 89
5.2 GoogLeNet memory bandwidth figures on DDR3-1600 and DDR4-1866 (burst length 8, memory bus width 8B) 90
5.3 Total simulated time of GoogLeNet for LT, LT-CA, and DDR3-1600 models (in milliseconds) 92
5.4 Total simulated time of GoogLeNet for LT, LT-CA, and DDR4-1866 models (in milliseconds) 93
5.5 Total simulator run-time of GoogLeNet for LT, LT-CA and DDR models (in seconds) on a 8-core host 93

5.6 Total simulated time of GoogLeNet for different computational capacities and bandwidth utilization factors β sweeping (0.1, 1) with a step of 0.1 (in milliseconds) for memory running on 800MHz, burst length of 8 and data width of 8B 94

LIST OF ALGORITHMS

Page

1 Main thread in each module in a TLM-2.0 LT model 54
2 Maintaining busy state in b_transport inside interconnect 62

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Rainer Dömer for guidance and continuous support over the past years. I am also grateful to my committee members, Professor Fadi Kurdahi and Professor Ian G. Harris for their advice and support.

I would like to thank all my colleagues at Lab for Embedded Computer Systems (LECS), Dr. Zhongqi Cheng, Daniel Mendoza, Vivek Govindasamy, and Yutong Wang for collaborations and friendships.

I have had the opportunity of mentorship by unique mentors and individuals in my career. I would like to give special thanks to Prof. Viktor Öwall, Lars Viklund, Erik Persson, Ola Hugosson, Prof. Abedin Vahedian, Kenny Ranerup, Dr. Dave Garret, Dr. Sumanth Jannyavula Venkata, Tor Silfverberg, Igor Lantsov, and Dr. Jalil Kamail.

My dissertation has been supported in part by funding from Intel Corporation and I thank Intel for their support.

VITA

Emad Malekzadeh Arasteh

EDUCATION

Doctor of Philosophy in Computer Engineering **2022**
University of California, Irvine *Irvine, California*

Master of Science in Electronic Design **2011**
Lund University *Lund, Sweden*

RESEARCH EXPERIENCE

Graduate Research Assistant **2018–2022**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Instructor **2022**
University of California, Irvine *Irvine, California*

Cyber-Physical Systems Case Studies 2022

Teaching Assistant **2018–2022**
University of California, Irvine *Irvine, California*

Advanced C Programming 2022
Embedded Systems Modeling and Design 2021
Security and Privacy in Cyber-Physical Systems 2020
Computational Methods in Electrical and Computer Engineering 2020
Computational Methods in Electrical and Computer Engineering 2019
Computational Methods in Electrical and Computer Engineering 2018

INDUSTRY EXPERIENCE

System Architecture Intern **Summer 2021**
Samsung Semiconductor Inc *San Jose, USA*

Hardware Engineering Intern **Summer 2019**
Syntiant Corp *Irvine, USA*

Experienced ASIC Engineer **2015–2018**
Axis Communications *Lund, Sweden*

ASIC Engineer & DSP Engineer **2012–2014**
Ericsson Modems *Lund, Sweden*

Embedded Software Engineer **2011–2012**
Hövding Sverige AB *Malmö, Sweden*

Hardware Placement Student **2010–2011**
ARM Sweden AB *Lund, Sweden*

REFEREED JOURNAL PUBLICATIONS

Emad Arasteh, Rainer Dömer, **Fast Loosely-Timed System Models with Accurate Memory Contention**, Journal of ACM Transactions on Embedded Computing Systems (under review), 2022

REFEREED BOOK CHAPTERS

Rainer Dömer, Zhongqi Cheng, Daniel Mendoza, Emad Arasteh, **Pushing the Limits of Parallel Discrete Event Simulation for SystemC**, Chapter 7 in "A Journey of Embedded and Cyber-Physical Systems" by Jian-Jia Chen, Springer Nature, Switzerland, August 2020. (ISBN 978-3-030-47487-4)

REFEREED CONFERENCE PUBLICATIONS

Emad Arasteh, Rainer Dömer, **Improving Parallelism in System Level Models by Assessing PDES Performance**, Proceedings of Forum on Specification and Design Languages, Antibes, France, September 2021

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

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

Emad Arasteh, Rainer Dömer, **An Untimed SystemC Model of GoogLeNet**, Proceedings of the International Embedded Systems Symposium, "Embedded Systems: Design, Analysis and Verification" (ed. G. Schirner, M. Götz, A. Rettberg, M. Zanella, F. Rammig), Springer, Friedrichshafen, Germany, September 2019.

TECHNICAL REPORTS

Emad Arasteh, Rainer Dömer, **Systematic Evaluation of Six Models of GoogLeNet using PDES**, Center for Embedded and Cyber-Physical Systems, Technical Report 21-03, September 2021

PRESENTATIONS

E. Arasteh:
Explore Parallelism and Memory Contention in System Level Models by Assessing PDES Performance,
PhD Forum at Design Automation Conference, San Francisco, California, July 2022

E. Arasteh:
Exploration of Memory-Accurate Models at Different Levels of Abstraction,
AMD Research, Advanced Intelligent Memory Group, United States, January 2022

E. Arasteh:
Transaction Level Modeling for Deep Neural Networks,
Center of Embedded and Cyber-physical Systems, UC Irvine, December 2020

E. Arasteh:
Out-of-Order Parallel Simulation of GoogLeNet,
Intel, Virtual Platform Modeling Group, United States, April 2019

ABSTRACT OF THE DISSERTATION

Transaction-Level Modeling of Deep Neural Networks
for Efficient Parallelism and Memory Accuracy

By

Emad Malekzadeh Arasteh

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2022

Professor Rainer Dömer, Chair

The emergence of data-intensive applications, such as Deep Neural Networks (DNNs), exacerbates the well-known memory bottleneck in computer systems and demands early attention in the design flow. Electronic System-Level (ESL) design using Transaction Level Modeling (TLM) enables early performance estimation, efficient design space exploration, and gradual refinement. In this dissertation, we present our exploratory modeling framework for hardware-software codesign based on IEEE SystemC TLM with particular focus on exposing parallelism and memory contention. We demonstrate the effectiveness of our approach for representative large DNNs such as GoogLeNet and Single Shot MultiBox Detector.

First, we study the impact of communication mechanisms on the available parallelism in TLM models. Specifically, we demonstrate the impact of varying synchronization mechanisms and buffering schemes on the exposed parallelism using different modeling styles of a DNN. We measure the performance of aggressive out-of-order parallel discrete event simulation and analyze the available parallelism in the models. Our study suggests that increased parallel simulation performance indicates better models with higher amounts of parallelism exposed.

Second, we explore the critical aspects of modeling and analysis of timing accuracy with the respect to memory contention. A major hurdle in tackling the memory bottleneck is

the detection of memory contention late in the design cycle when detailed timed or cycle-accurate models are developed. A bottleneck detected at such a late stage can severely limit the available design choices or even require costly redesign. To explore new architectures prior to RTL implementation, we propose a novel TLM-2.0 loosely-timed contention-aware (LT-CA) modeling style that offers high-speed simulation close to traditional loosely-timed (LT) models, yet shows the same accuracy for memory contention as low level approximately-timed (AT) models.

Finally, we further refine the TLM-2.0 AT model by adding a cycle-accurate model of a memory subsystem. This model provides a higher timing accuracy for contention analysis. Hence it provides more accurate estimation of the performance. We revise our LT-CA memory delay modeling to provide further accuracy comparable to the cycle-accurate AT model of the shared memory subsystem. The high amount of contention on the shared memory suggests the need to move toward new processor architectures with local memories.

Chapter 1

Introduction

With the rapid growth in complexity of electronic devices and the drastic reduction in time to market, Electronic System Level (ESL) methodology has been proposed for modeling systems at higher levels of abstraction [6, 20]. ESL ideas resulted in defining System-level Description Languages (SLDL), such as SpecC [22] and SystemC [24], that can model both hardware and software components and their detailed interactions.

1.1 Electronic System Level and Transaction Level Modeling

ESL techniques focus on Transaction Level Modeling (TLM) which separates computation from communication in the model [8]. This allows refinement of computation and communication independently as well as on different abstraction levels. In this way, TLM can speed up simulation significantly by replacing many pin-level events in RTL simulation with an abstract function call. In general, the higher the level of abstraction is, the faster the simulation runs. Naturally, this simulation speedup typically comes at the price of lower model

accuracy.

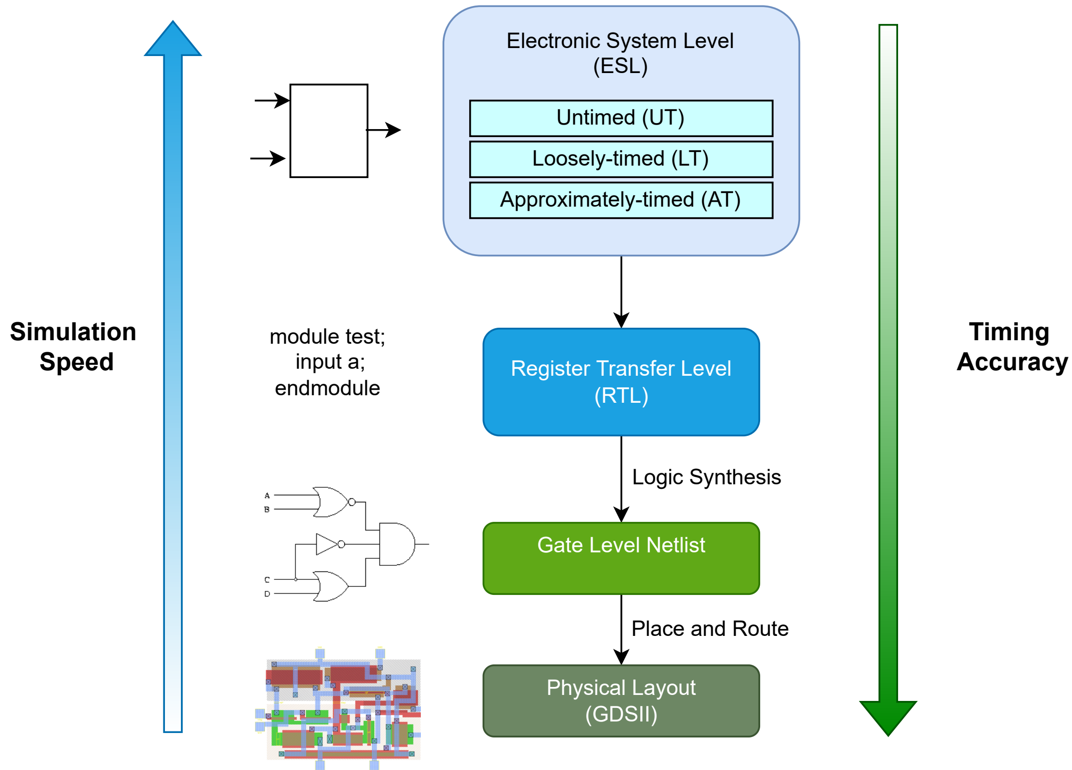


Figure 1.1: Top-down ESL design methodology

Figure 1.1 illustrates this classic trade-off of speed/accuracy in ESL design methodology. Electronic system-level models specify the behavior of the design in the highest level of abstraction by presenting both hardware and software components in parallel. Instead of defining a strict taxonomy of abstraction levels, TLM standard defines a set of application programming interfaces (APIs) and describes a set of coding styles that are appropriate for various system-level modeling use cases. As such, the loosely-timed (LT) coding style is appropriate for the use cases of software development and performance optimization. LT models simulate fast and have sufficient timing details to boot an operating system. On the other hand, the approximately-timed (AT) coding style is suitable for the use cases of architecture exploration and detailed performance analysis. AT models simulate slower but carry better timing accuracy than LT models [27].

In summary, ESL and TLM raise the design abstraction above RTL to overcome the challenges of designing today's complex system-on-chips (SoCs). In particular, TLM provides an agile hardware-software codesign framework for early exploration of wide ranges of design metrics and evaluation of design candidates. Moreover, TLM provides a codesign environment wherein software can be developed in parallel with hardware. This is beneficial not only for earlier system integration, but also for rapid feedback to system designers.

1.2 Parallel SystemC Simulation

The Accellera Systems Initiative maintains the official IEEE standard SystemC language and also provides an open source proof-of-concept library to simulate SystemC design models [49]. However, this reference simulator implements the classic discrete event simulation (DES) scheme which runs sequentially. Hence, the simulator can not utilize the available parallel computing resources in today's multi-core and many-core processor hosts. To achieve faster execution, parallel discrete event simulation (PDES) techniques can be employed [16].

Earlier works on PDES, such as [9], focused on distributed simulation hosts. Fujimoto [19] presented the first initial work on parallel and distributed hosts. While significant obstacles for standard-complaint parallel SystemC simulation exist [15], many parallel SystemC simulation approaches have been proposed [13, 51, 55]. Beyond these synchronous PDES techniques, out-of-order parallel simulation was first proposed in [11] to maximize simulation speed for ESL designs at any abstraction level. The out-of-order PDES approach proposes to allow threads in different cycles to run in parallel if those threads do not have potential data or event conflicts [11]. OoO PDES maximizes multi-core and many-core CPU utilization by localizing global simulation time for each thread and performing conservative analysis of potential conflicts among the active threads.

OoO PDES approach is realized in Recoding Infrastructure for SystemC (RISC) that performs parallel SystemC simulation in maximum compliance with the IEEE standard semantics using a dedicated SystemC compiler that automatically analyzes existing conflicts in the model [42]. RISC identifies all potential race conditions, and instruments the model to avoid any conflicts. This transformation is completely automatic and it does not require any manual recoding or application-specific knowledge [16].

Despite the fact that RISC maximizes the number of threads to run in parallel, we optimize TLM models such that they exhibit further parallelism opportunities so that RISC achieves even faster simulation.

1.3 Memory Bottleneck

The term von Neumann bottleneck, widely known as the memory bottleneck, was coined by John Backus in 1978 [4]. Von Neumann computers are built around an inherent bottleneck that is “the word-at-a-time tube connecting the CPU to the memory” [4]. Since the birth of the first von Neumann computer in 1945, various innovations have developed to alleviate the memory bottleneck. Multi-level cache hierarchies, shared scratchpad memory, multi-channel memory architecture, Non-Uniform Memory Access (NUMA) architecture, and more recently, computation-in-memory [59] are only a few of the inventions to tackle the memory bottleneck in computer systems. Despite all these efforts, the memory bottleneck still remains as one of the grand challenges of computer science and engineering.

1.4 Deep Learning and Convolutional Neural Networks

Deep Learning (DL) is a known technique in machine learning to extract useful features from input data, perform data transformations, and arrive at a final meaningful representation. One of the main application areas of DL is visual recognition and in particular, image classification, which is the problem of assigning a descriptive label to an input image from a fixed set of categories.

A Convolutional Neural Networks (CNN) mainly consists of alternating convolution layers and pooling (sub-sampling) layers. Each convolution layer extracts features in the input by applying trainable filters to the input. Later, the convolved feature is fed to an activation function, for example a Rectifier Linear Unit (ReLU) to introduce nonlinearity and obtain activation maps. Each pooling layer downsamples the activation maps to reduce computation and memory usage in the network. Features extracted from previous convolution and pooling layers are fed to a fully connected layer to perform classification. Typically, a softmax activation function can be placed following the final fully connected layer to output the probability corresponding to each classification label.

Early work on CNN dates back to 1989 with the LeNet network for handwritten digit recognition [38]. For example, LeNet-5, a CNN for digit recognition, as depicted in Figure 1.2, contains three convolution layers, two sub-sampling layers, and one fully connected layer [39]. However, the early 2010s started a new era for CNN applications by the introduction of AlexNet [34] for image classification. Growth of computing power, availability of huge datasets that can be used for training, and rapid innovation in deep learning architectures have paved the way for the success of deep learning techniques in recent years [61].

Choosing a state-of-the-art deep CNN for TLM modeling enables the means to investigate parallelism opportunities and the memory bottleneck problem. Therefore, we select GoogLeNet [62], a deep CNN for image classification and detection, and start with a refer-

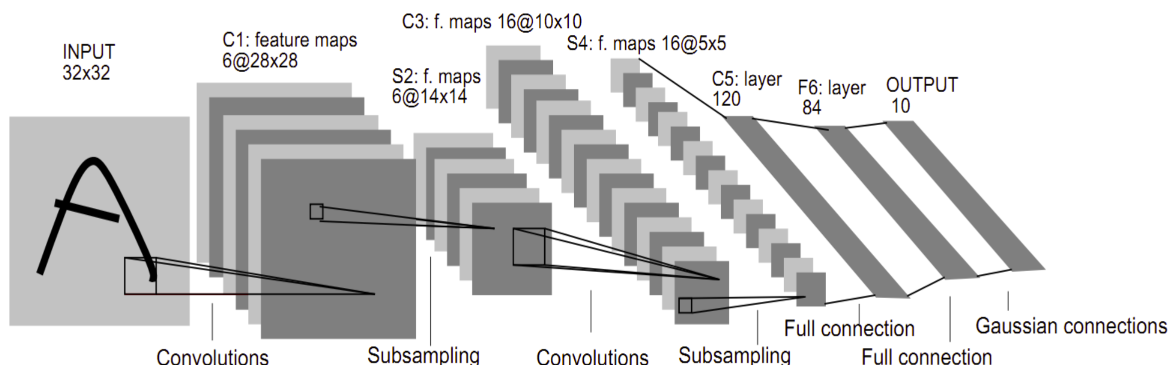


Figure 1.2: Architecture of LeNet-5, a CNN for digits recognition [39]

ence model in SystemC [1].

1.4.1 GoogLeNet Structure

GoogLeNet is a deep CNN for image classification and detection. It won the ImageNet Large Scale Recognition Competition (ILSVRC) in 2014 with only 6.67% top-5 error [62]. GoogLeNet was proposed and designed with computational efficiency and deployability in mind. The two main features of GoogLeNet are (1) using 1x1 convolution layers for dimension reduction and (2) applying Network-in-Network architecture to increase the representational power of the neural network [62].

GoogLeNet is 22 layers deep when counting only layers with parameters. As detailed in Table 1.1, the overall number of independent building blocks is 142 distinct layers. The main constituent layer types are convolution, pooling, concatenation, and classifier. GoogLeNet includes two auxiliary classifiers that are used during training to combat the so-called vanishing gradient problem. Our focus for now is on inference by using the proposed neural network architecture, and not the training for fine-tuning network parameters or suggesting improved network architecture. Therefore, our model does not include the two auxiliary classifier layers. The detailed types of layers inside GoogLeNet and the number of each type

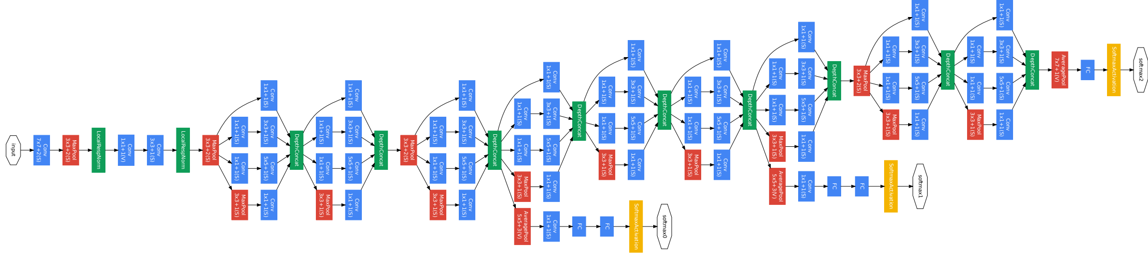


Figure 1.3: GoogLeNet network with all the bells and whistles [62]

of layers are summarized in Table 1.1.

Table 1.1: GoogLeNet layer summary

Layer type	Count
Convolution	57
ReLU	57
Pooling	14
LRN	2
Concat	9
Dropout	1
InnerProduct	1
Softmax	1
Total	142

A schematic view of GoogLeNet is depicted in Figure 1.3. An image is fed in on the left, and processed by all layers. Then, a vector with probabilities for the set of categories comes out on the right. The index of a class with a maximum probability is looked up in a table of synonym words that outputs the class of the object in the image, i.e. "space shuttle".

1.4.2 Single Shot MultiBox Detector Structure

Single Shot Detector (SSD) is our second state-of-the-art deep CNN application that was introduced in 2016 for object detection. Object detection is a computer vision technique to detect objects of a certain class in an image. SSD is an efficient method for detecting objects in images and videos with a competitive accuracy of 74.3 % mean average precision

at 59 FPS for a 300x300 input image in the VOC2007 dataset [44]. In addition to localizing objects in the image and generating scores for detected objects in a single forward pass of the network, SSD creates bounding boxes with proposed coordinates around each detected object. Such object detection has a wide range of use cases in autonomous driving, health care and machine safety. A summary of the layers in the SSD network is shown in Table 1.2.

Table 1.2: Single Shot Detector layer summary

Layer type	Count
Convolution	35
ReLU	23
Flatten	13
Permute	12
PriorBox	6
Pooling	5
Concat	3
Reshape	1
Normalize	1
Softmax	1
DetectionOutput	1
Total	101

SSD object detection network comprises base network to extract feature maps, and convolutional predictors to detect objects. A key feature of SSD is elimination of bounding box proposals to achieve real-time detection speed [44]. Similar to GoogLeNet, our focus is on the inference using pre-trained neural network parameters which is ready for deployment.

Figure 1.4 illustrates the schematic of SSD using Netron [54], a viewer for neural network, deep learning, and machine learning models. An image is fed to the network from the top, after layers of processing, the network generates a set of bounding boxes coordinates and class labels for objects detected in the image with their corresponding class probabilities.

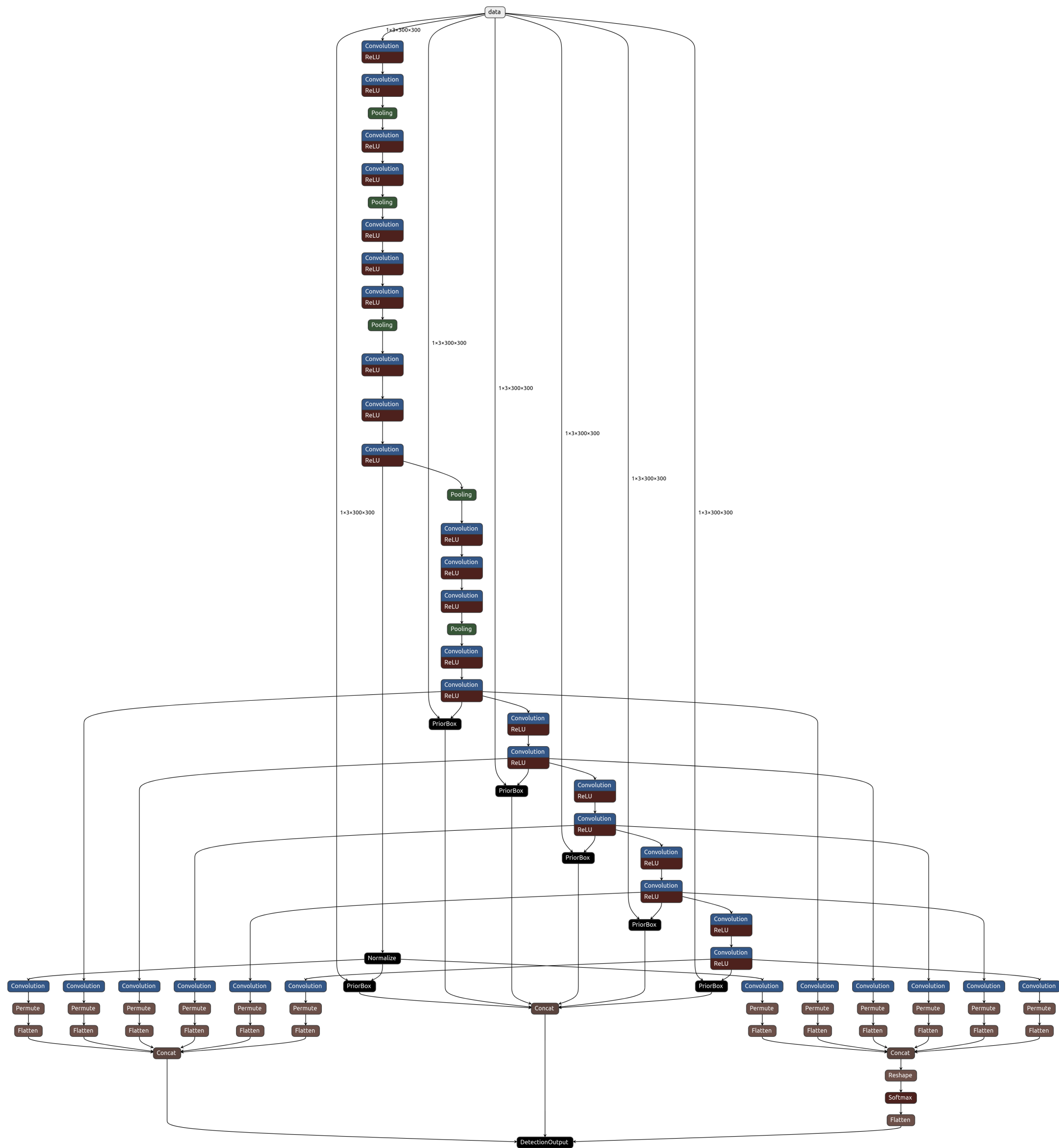


Figure 1.4: Single Shot MultiBox Detector (SSD) network schematic

1.5 Related Work

Static analysis of SystemC models and TLM modeling techniques for parallel SystemC simulation has been studied in other works. The SystemC-clang framework [33] analyzes SystemC models at register-transfer level and transaction-level with support for some TLM 2.0 constructs. Authors in [63], [65], and [66] propose modified parallel SystemC simulation kernels

that require users to manually translate their sequential models into safe parallel models. [47] provides SystemC designers with a set of primitives to manually parallelize SystemC tasks for loosely-timed models. These techniques require the designer to manually instrument the model for safe parallel simulation. In contrast to prior works, our approach leverages from the complete automatic parallelization in the RISC to increase PDES simulation performance in a safe and standard-compliant fashion.

Automatic generation of a set of RTL primitives by analyzing CNN architecture and parameters to be used on FPGA has been carried out in [69]. To the best of our knowledge, there is no similar work on improving the parallelism in SystemC TLM for CNN.

There is a large body of research on performance modeling and memory contention modeling and analysis. Most methods for system level performance analysis can be broadly categorized into the two main classes of analytical and simulation based methods. In analytical approaches, the system is mathematically modeled and its performance is analytically derived as a function of workload and input parameters. Frank et al. [18] define an analytical contention model in parallel algorithms on a multiprocessor workstation. Chen et al. [10] use queueing theory to model contention in bus-based system design. Analytical models are dependent on the architecture described, and a new model must be developed for each new architecture or application [7]. Moreover, analytical modeling does not take into account the dynamic behavior of the system and often use of more realistic assumptions makes meaningful analysis difficult [5].

Simulation-based approaches can capture many dynamic and complex interactions in a system. SpecC [22] and SystemC [24] are widely-used system level description languages for modeling, simulation and validation of complex system-on-chip models. The SystemC C++ class library is an IEEE standard that enables both system and transaction level modeling using discrete event simulation (DES) [27]. Simulation techniques often suffer from long simulator run-times at lower abstraction levels. Furthermore, there are high costs associated

with manually building simulation models and debugging them.

Gajski et al. [21] introduces estimation trade-off metrics such as accuracy, fidelity, and speed in design modeling. A systematic and quantitative analysis of the speed/accuracy trade-off in TLM has been studied in [57]. A method of overcoming this general trade-off for the specific case of processor models is proposed in [58].

To overcome strictly simulation-based methods, a hybrid approach of analytical and simulation methodologies has been proposed. Lunzli et al. [35] propose a method to combine SystemC based simulation with formal analysis based on real-time calculus. Borbek et al. [7] also combine simulation with an analytical method with focus on the analysis of shared resource contention. While these mixed methodologies help to shorten simulator run-times, the coverage for corner cases in simulation still remains difficult [64]. Furthermore, [7] operates at a much higher level of abstraction than TLM and thus sacrifices some accuracy for higher simulation speedup.

Aside from analytical and simulation based modeling approaches, there are also experimental techniques to measure the effect of memory contention. More recently, DNN library profilers such as PyTorch Profiler [53] and performance profilers such as Intel VTune Profiler [28] provide some coarse-grain measures on memory usage and footprint. However, the results are valid only for a specific processor architecture and memory hierarchy. This hardware dependency is not helpful for design space exploration or refinement to lower-level abstraction.

Exploration of alternative memory architectures for programmable embedded systems has been carried out in [25] and efficient utilization of scratch-pad or working memories in embedded processor application has been described in [52]. Moreover, integrated circuit (IC) technology roadmaps plan for a close integration of memory-logic fabrics in the future SoCs [67]. As such, UPMEM [14] is the first designed and fabricated Processing in-Memory (PIM)

architecture for real-world data-intensive applications which is publicly available. UPMEM PIM architecture consists of thousands of processors placed within the DRAM memory chips. This allows to offload the computation directly into the memory chips where the data resides to drastically reduce off-chip data movements.

Our proposed TLM framework is based on the well-defined top-down ESL design methodology which SystemC makes easy to deploy. Our automatic model generation dramatically reduces the burden of constructing and debugging simulation models. Furthermore, memory contention is modeled accurately and simulates fast, enabling efficient early design space exploration.

1.6 Goals

The commonality between emerging computing applications is being data-centric. Artificial intelligence, machine learning and deep neural networks all demand for high-capacity and high-bandwidth memories for data storage and processing. While latest technological advances allow availability of massively parallel processor arrays in a single SoC [48, 70], a growing productivity gap exists in the hardware and software design of embedded systems [17]. The continuing trends towards data-intensive applications and rapid advances in 3-D integration of memory-logic fabrics [56] necessitate new hardware-software codesign approaches with particular emphasis on parallelism and memory contention. Such a system-level modeling framework will be a cornerstone to build the next generation intelligent SoCs that are efficient, safe, and reliable.

In this dissertation, we aim to propose system-level models of deep neural networks which expose the design opportunities and challenges early in the development cycle. Specifically, we aim to propose a holistic system-level modeling framework to explore parallelism and

memory accuracy in early stages of the design. These efforts significantly improve the overall design time, cost and quality of final physical implementations. Our goals include:

1. **DNN System Level Modeling with Exposed Parallelism**: We need system-level models of DNNs for analysis and simulation. We aim to design and simulate a system-level model of a DNN based on SystemC in a modular and reusable manner. Later, we aim to explore the effects of thread-level parallelism at SystemC level and multi-threaded parallelism at application level on simulation speedup.

2. **Improvement of System Level Parallelism**: TLM explicitly exposes inherent parallelism in the application by modeling concurrency, hierarchy, synchronization and timing. We aim to exploit the available parallelism in the TLM simulation model with a parallel simulator for maximum simulation speedup. Moreover, we aim to find out if the model that simulates faster in PDES is also a better model for further refinement.

3. **Fast Loosely-Timed System Models with Accurate Memory Contention**: TLM enables early performance estimation, efficient design space exploration, and gradual refinement. However, memory contention is often not detectable before detailed TLM-2.0 approximately-timed or cycle-accurate RTL models are developed. A memory bottleneck detected at such a late stage can severely limit the available design choices or even require costly redesign. We aim to find a modeling approach that breaks the speed/accuracy tradeoff between regular loosely-timed and approximately-timed models and offers fast and accurate observation and visualization of memory contention early in the design cycle.

4. **Cycle Accuracy in Memory Modeling**: Cycle-accurate memory models exhibit the most accurate estimation of memory contention in the design. We aim to design TLM models that can accurately show the behavior of real-world memory subsystems. To avoid the orders of magnitudes slower simulation in cycle-accurate models, we aim

to find a technique to improve the accuracy and fidelity of our system models without losing simulation performance. Furthermore, we aim to deploy our modeling framework to propose alternative memory architectures to mitigate memory contention in DNNs.

In the rest of this dissertation, we present our work and contributions to achieve the above goals:

In Chapter 2, we introduce a newly designed untimed SystemC model of GoogLeNet using OpenCV library [1]. We successfully validate the functionality of the model using Accellera SystemC 2.3.1 simulator. Then, we use RISC (Recoding Infrastructure for SystemC) to speed up the simulation by exploiting thread-level parallelism. We also explore the effect of multi-threaded parallelism at application level on simulation speedup and report extensive experimental results.

In Chapter 3, we propose and analyze a set of non-invasive standard-compliant modeling techniques to increase parallelism in IEEE SystemC TLM-1 and TLM-2.0 models [2]. In particular, we demonstrate the impact of varying synchronization mechanisms with simulator run time using six modeling styles of a DNN. To quantify the parallelism in these six models, we measure the performance of aggressive out-of-order PDES in RISC [2, 3]. Our study suggests that increased parallel simulation performance indicates better models with higher amounts of parallelism exposed.

In Chapter 4, we propose a novel TLM-2.0 loosely-timed contention-aware (LT-CA) modeling style that offers high-speed simulation close to traditional loosely-timed (LT) models, yet shows the same accuracy for memory contention as low level approximately-timed (AT) models [45]. We describe our extensible SystemC model generator that automatically produces desired TLM-1 and TLM-2.0 models from a DNN architecture description for design space exploration with focus on parallelism and memory contention. The experimental results show that the proposed LT-CA modeling is 46x faster in simulation than equivalent

AT models with an average error of less than 1% in simulated time.

In Chapter 5, we augment our modeling framework to include cycle-accurate memory models of Dynamic Random Access Memory (DRAM) subsystems. We also propose an enhancement of our memory delay estimation for LT-CA modeling to improve timing accuracy and fidelity of the design model. We report extensive experimental results with cycle-accurate memory models and analyze accuracy of LT, LT-CA and cycle-accurate memory modeling. As a result, we propose a local memory architecture as an alternative to conventional approaches to mitigate memory contention in DNNs.

Finally, Chapter 6 summarizes the contributions of this dissertation and presents the potential research problems for future work.

Chapter 2

Untimed SystemC Model of DNN

In this chapter [1], we develop an untimed SystemC model of GoogLeNet [62], a state-of-the-art deep CNN. Following the top-down specification approach for a classical system on chip design [23, 24], our goal is to separate communication parts from computation parts. To achieve this, we exploit the fact that a neural network is a directed graph where the nodes are different layers in the network and edges connect neighboring layers. Later, we explore the effects of multi-threaded parallelism at application level, thread-level parallelism at SystemC level, and also combination of both sources of parallelism on simulation speedup in multi-core processors.

2.1 Introduction

Latest trends in cutting edge deep neural network architectures like GoogLeNet (2014) [62], ResNeXt (2016) [68], FractalNet (2016) [37], DenseNet [26] (2017), etc. show a substantial increase in the number of multiple parallel connections between layers in the network. This comes with a high level of thread-level parallelism, which parallel simulators can take

advantage of for faster simulations.

The rest of this chapter is organized as follows: Section 2.2 introduces the system-level modeling framework and modeling strategies. Section 2.3 describes SystemC modeling details of each layer. Section 2.4 describes the top-level test-bench structure of untimed SystemC model of a DNN. Section 2.5 presents sequential and parallel simulation results with an analysis of valuable observations. At last, Section 2.6 concludes this case study.

2.2 System-level Modeling Framework

A well-defined modeling strategy is essential to manage the system complexity and provide maximum flexibility. Our system modeling framework follows three criteria introduced in [1]:

1. *Generic layers*: Since a CNN is composed of a handful of layer types, the layers shall be parameterized by their attributes using a custom constructor. For example, a pooling layer shall be parameterized by its type (max-pooling or average pooling), its kernel size, its stride, and the number of padding pixels.
2. *Self-contained layers*: Each layer shall implement the functionality it requires without the need of an external scheduler to load its input or in some cases load its parameters. For example, a convolution layer shall have a dedicated method to load its parameters (weight matrix and bias vector) used only at the time of construction.
3. *Reusability and modularity*: Since most CNNs share a common set of layers, the code shall be structured in a way to enable the feeding of any kind of CNN with minimum effort. For example, the layer implementation shall be organized as code template blocks and the SystemC model shall be automatically generated using only the network model defined by the AI framework.

To obtain pre-trained network parameters, we have used the Caffe (Convolutional Architecture for Fast Feature Embedding) model zoo. Caffe is a deep learning framework originally developed at University of California, Berkeley, and is available under BSD license [29]. Caffe models come with (1) a binary file `.caffemodel` that contains network parameters, and (2) a text file `.prototxt` that specifies the network architecture. Class labels are also provided in a text file format that includes a synonym ring or synset of those labels.

Our SystemC models rely on efficient optimized code inside OpenCV 3.4.1. OpenCV is a library of computer vision functions mainly aimed for real-time applications written in C/C++ [50]. The OpenCV library was originally developed by Intel and is now free for use under the open-source BSD license. OpenCV uses an internal data structure to represent an n-dimensional dense numerical single-channel or multi-channel array, a so called `Mat` class. Therefore, our models use the `Mat` data type to store images, weight matrices, bias vectors, feature maps, and class scores. This becomes practical while interacting with various OpenCV application programming interfaces (APIs).

Furthermore, OpenCV provides an interface class, `Layer`, that allows for construction of constituent layers of neural networks. A `Layer` instance is constructed by passing layer parameters and is initialized by storing its learned parameters. A `Layer` instance computes an output `Mat` given an input `Mat` by calling its `forward` method. We refer to this class as OpenCV `layer` for the rest of this dissertation. OpenCV also provides utility functions to load an image and read a Caffe model from `.prototxt` and `.caffemodel` files.

Note that these goals will allow us to easily generate a SystemC model also for other Caffe CNNs. At the same time, the models generated will have a well-organized structure that enables static analysis. Specifically, this allows us to perform parallel simulation with RISC [41]

Our system-level modeling framework follows the well-known Specify-Explore-Refine (SER)

methodology [20] which is successive, stepwise refinement of design models. We now describe how we design a DNN SystemC model based on the pure functional C++ model.

2.3 DNN Layer Implementation

Each layer in the CNN is modeled as a SystemC `sc_module` with input and output ports. Ports in each module are defined as `sc_port` and are parameterized either by primitive or by user-defined interface classes. The user-defined interfaces are derived from `sc_interface` and declare `read` and `write` access methods with a granularity of `Mat`. The choice of `Mat` for the granularity of port parameterization simplifies the design by focusing on the proper level of abstraction at this level of modeling. As an example, the module definition of the first convolution layer `conv1_7x7_s2` is shown in Listing 2.1.

As shown in lines 41-53 of Listing 2.1, each module has several attributes that are all defined as data members inside the class definition. For example, a convolution module is defined by its name, number of outputs, number of pixels for padding, kernel size, and number of pixels for stride. If a layer also has learned parameters, two Mat objects are defined as member variables to store the weight matrix and the bias vector. In that case, their values are initialized at the time of module construction. For example, a convolution module has a designated load method that reads pre-trained Caffe model files and stores weight and bias values in the `weights` and `bias` member variables.

```
1 class conv1_7x7_s2_t : sc_core::sc_module
2 {
3
4 public:
5   sc_core::sc_port<mat_in_if>  blob_in;
6   sc_core::sc_port<mat_out_if> blob_out;
7
```

```
8   SC_HAS_PROCESS(conv1_7x7_s2_t);
9
10  conv1_7x7_s2_t(sc_core::sc_module_name n_,
11      String name_,
12      unsigned int num_output_,
13      unsigned int pad_,
14      unsigned int kernel_size_,
15      unsigned int stride_,
16      unsigned int dilation_,
17      unsigned int group_) :
18    sc_core::sc_module(n_),
19    name(name_),
20    num_output(num_output_),
21    pad(pad_),
22    kernel_size(kernel_size_),
23    stride(stride_),
24    dilation(dilation_),
25    group(group_),
26    weights(4, weight_sz, CV_32F, weight_data),
27    bias(4, bias_sz, CV_32F, bias_data)
28  {
29    load();
30    SC_THREAD(main)
31  }
32
33  void load();
34  void main();;
35  void run(std::vector<Mat> &inpVec,
36      std::vector<Mat> &outVec);
37
38 private:
39
40  String            name;
```

```
41   unsigned int        num_output;
42   unsigned int        pad;
43   unsigned int        kernel_size;
44   unsigned int        stride;
45   unsigned int        dilation;
46   unsigned int        group;
47   static const int    weight_sz[4];
48   unsigned int        weight_data[64*3*7*7];
49   static const int    bias_sz[4];
50   unsigned int        bias_data[64];
51   Mat                 weights;
52   Mat                 bias;
53
54 };
```

Listing 2.1: Conv1_7x7_s2 module definition

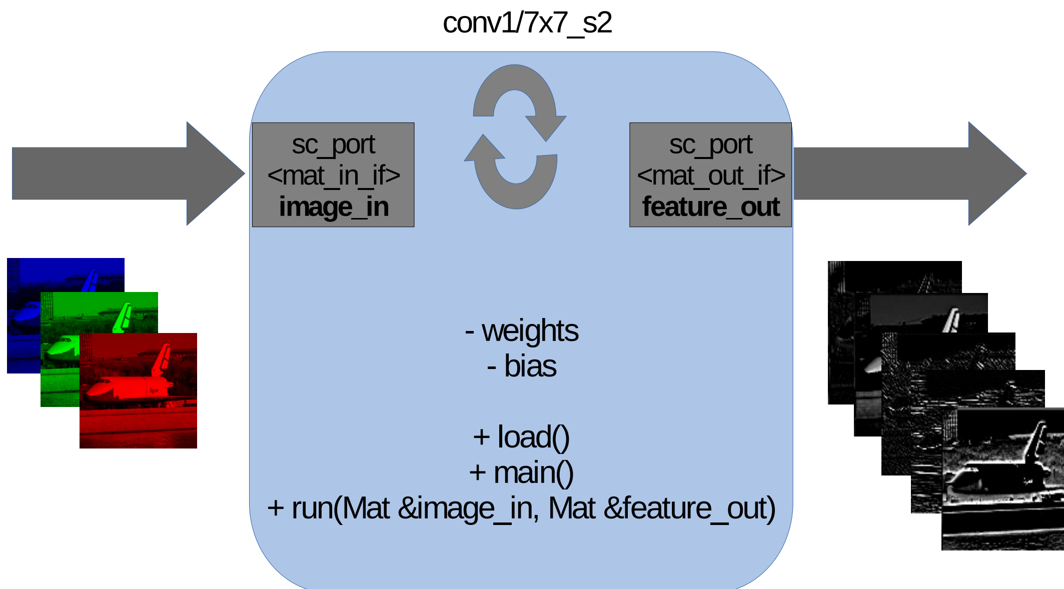


Figure 2.1: Convolution layer

Each module has a main thread that continuously reads its input port, computes results, and writes those to its output port. Data processing is handled by a `run` method that interacts

with the OpenCV library. The `run` method creates an instance of OpenCV `layer` and calls its `forward` method by passing references to input `Mat` and output `Mat` objects.

As an example, Figure 2.1 illustrates the module defining the first convolution layer in GoogLeNet. The input to the module is a `Mat` object containing 3 color channels of 224x224 pixels of the input "space shuttle" image and the output is another `Mat` object containing 64 feature maps with the size of 112x112 pixels.

2.4 Validation by Simulation

A top level test bench validates our GoogLeNet SystemC model against the reference OpenCV implementation. The test bench instantiates our SystemC GoogLeNet module which contains all modules inside the network with all the interconnecting queues as Design under Test (DUT). It also instantiates a stimulus module to feed the design with images of size 224x224 with three color channels, and a monitor module to read the final class scores and output the label with the maximum probability (Figure 2.2). To measure the performance of the model, our test bench can also be configured to continuously feed in a stream of images. In that case, a checker module is plugged inside the monitor to check the correct classification and its probability against the reference model.

2.5 Experimental Results

Our untimed SystemC model of GoogLeNet compiles and simulates successfully with Accellera SystemC 2.3.1. For parallel simulation, we also compile and simulate the model using RISC V0.5.1 to speed up simulator run time. Both simulation results match the OpenCV reference model output.

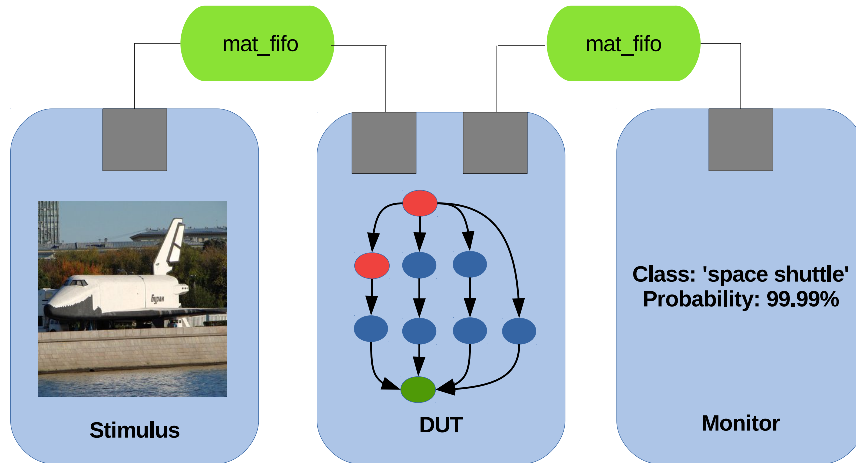


Figure 2.2: Top-level test bench

2.5.1 Performance Setup

We use two different computer platforms to benchmark the simulations. The specifications of each platform are shown in Table 2.1. We name platforms based on the number of logical cores visible to the operating system. The number of logical cores is double the number of physical cores when hyper-threading technology (HTT) is enabled.

To have reproducible experiments, the Linux CPU scaling governor is set to 'performance' to run all cores at the maximum frequency, and file I/O operations, i.e. *cout*, are minimized. SystemC 2.3.1 and OpenCV 3.4.1 are built with debugging information [1].

Moreover, the OpenCV library can be built with support for several parallel frameworks, such as POSIX threads (pthreads), Threading Building Blocks (TBB), and Open Multi-Processing (openMP), etc. We build OpenCV with the support for pthread to run in multithreaded mode and also without support for pthread to run only on a single-thread. Lastly, the stimulus module is configured to feed 500 images with size of 224x224 pixels to the model.

[1] OpenCV has built with -O0 flag meaning (almost) no compiler optimizations.

Table 2.1: Simulation platforms specification

Platform name	4-core host	8-core host	16-core host	32-core host
OS	CentOS 7.6	CentOS 7.6	CentOS 6.10	CentOS 6.10
CPU Model name	Intel E3-1240	Intel E3-1240	Intel E5-2680	Intel E5-2680
CPU frequency	3.4 GHz	3.4 GHz	2.7 GHz	2.7 GHz
#cores	4	4	8	8
#processors	1	1	2	2
#threads per core	1	2	1	2

Table 2.2: Measurement results on 4-core host (HTT off)

	Single-thread		Multithreaded	
Time (sec)	Accellera	RISC	Accellera	RISC
User time	627.19	651.59	680.01	664.02
System time	1.55	1.11	34.26	18.26
Elapsed time	629.49	253.29	199.44	234.36
CPU utilization	99%	257%	358%	291%
Speedup	1x	2.48x	3.15x	2.68x

2.5.2 Simulation Results

For benchmarking, we measure simulation time using Linux */usr/bin/time* under CentOS. This time function provides information regarding the system time, the user time, and the elapsed time. Measurements are reported for sequential SystemC simulation using Accellera SystemC compiled with POSIX threads. Parallel simulation is performed using RISC simulator V0.5.1 in non-prediction (NPD) mode. Tables 2.2 to 2.5 show the measurements for each simulation mode on the four different platforms using the single-thread and multithreaded OpenCV. In case of parallel simulations, we set the maximum number of concurrent threads allowed by the RISC simulator to the number of available logical cores on each platform.

Table 2.3: Measurement results on 16-core host (HTT off)

	Single-thread		Multithreaded	
Time (sec)	Accellera	RISC	Accellera	RISC
User time	912.79	921.95	1164.69	960.48
System time	34.76	42.19	705.22	134.25
Elapsed time	947.93	275.29	154.45	260.7
CPU utilization	99%	350%	1210%	419%
Speedup	1x	3.44x	6.13x	3.63x

Table 2.4: Measurement results on 8-core host (HTT on)

	Single-thread		Multithreaded	
Time (sec)	Accellera	RISC	Accellera	RISC
User time	621.49	961.44	1164.13	1046.39
System time	1.52	1.28	84.06	34.85
Elapsed time	622.68	254.07	184.09	232.57
CPU utilization	100%	378%	678%	464%
Speedup	1x	2.45x	3.38x	2.67x

Table 2.5: Measurement results on 32-core host (HTT on)

	Single-thread		Multithreaded	
Time (sec)	Accellera	RISC	Accellera	RISC
User time	911.98	1177.02	2124.87	1299.86
System time	35.31	52.76	1838.35	224.84
Elapsed time	947.7	273.27	155.72	274.29
CPU utilization	99%	450%	2544%	555%
Speedup	1x	3.46x	6.08x	3.45x

2.5.3 Analysis

Table 2.2 allows the following observations:

1. **RISC introduces thread-level parallelism**

   RISC is faster than single-thread OpenCV with Accellera and it speeds up simulator run time up to 2.48x on the 4-core machine.

2. **OpenCV parallelism is even faster than RISC**

   We observe that multithreaded OpenCV speeds up simulator run time using Accellera up to 3.15x on the 4-core machine. Therefore, thread-level parallelism in OpenCV primitives is more efficient than thread-level parallelism at SystemC level.

3. **Combining OpenCV and RISC parallelism does not deliver the best speedup**

   Since RISC and OpenCV threads unknowingly from each other compete for resources, exploiting parallelism in RISC and OpenCV at the same time does not increase the speedup. For example, multithreaded OpenCV using RISC (2.68x) performs worse than multithreaded OpenCV using Accellera (3.15x) on the 4-core machine.

4. **RISC performance improves slightly with OpenCV parallelism**

   RISC gains small speedup by also using parallelism in OpenCV. For example, RISC speeds up multithreaded OpenCV (2.68) in comparison with single-threaded OpenCV (2.48x).

Table 2.3 supports observations 1 through 4 as well. It also allows for the following observation:

5. **Performance does not scale by the number of cores**

Quadratic increase in the number of cores only leads to double increase in performance. Relative good speed up to 3.15x on the 4-core machine does not scale to 16-core machines and only gets 6.13x speedup compared to sequential single-thread simulator run time.

Table 2.4 and 2.5 use hyper-threading technology (HTT) and allow for the following observations:

6. **HTT is ineffective for this application**

   Enabling HTT slightly improves speedup from 3.15x on the 4-core machine without HTT to 3.38 on the 4-cores with HTT (8-cores). In the case of 16-cores to 32-cores, performance has not improved at all.

7. **HTT substantially increases user and system time**

   We observe that the user and system times increase significantly with HTT turned on. At this point, the origin of this time increase is unclear for us. We will investigate this further in more detailed future research.

In summary, Figure 2.3 shows the speedups for different sources of parallelism: single-threaded OpenCV using **RISC**, multithreaded **OpenCV** using Accellera and multithreaded **OpenCV** using **RISC**. The illustration shows a significant speedup using parallelism introduced by RISC and multithreaded OpenCV. It also demonstrates that combining OpenCV and RISC parallelism does not provide a remarkable speedup.

2.6 Conclusion

In this chapter, we have described an untimed SystemC model of GoogLeNet using OpenCV 3.4.1 library. We also developed a tool to automatically generate SystemC code from Caffe

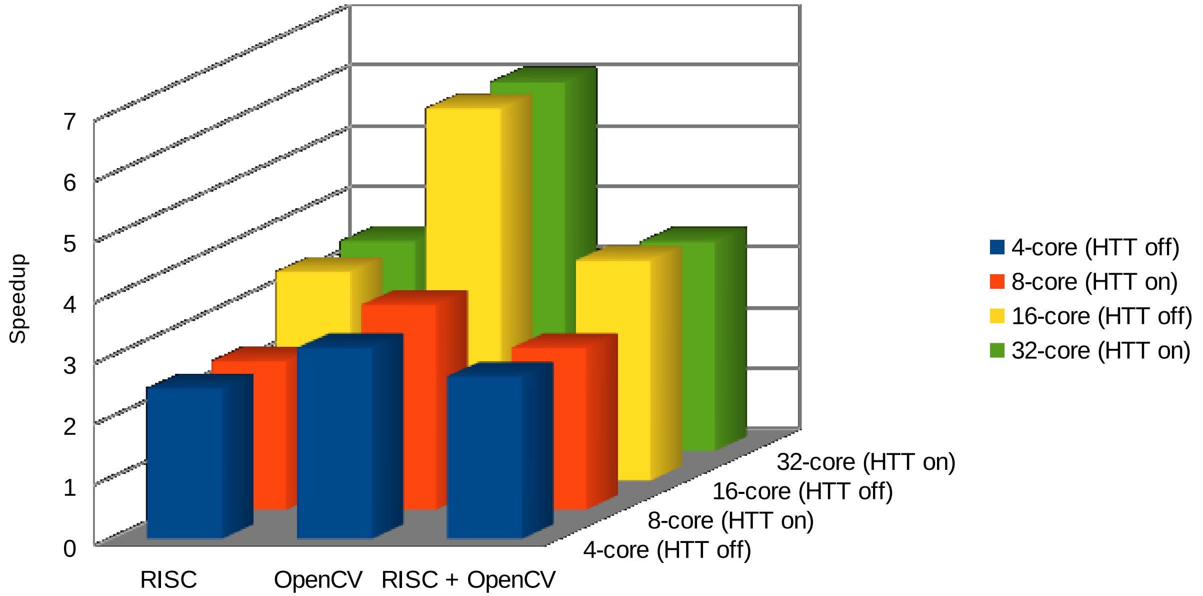


Figure 2.3: Speedup comparison on different platforms based on the source of parallelism

model files. We successfully simulated the generated model using Accellera SystemC 2.3.1 and RISC V0.5.1.

Experimental results show significant simulation speedups using RISC, as well as using multithreaded OpenCV. Results also show that combining OpenCV and RISC parallelism did not deliver significant speedup.

Chapter 3

Improving Parallelism in System Level Models

Transaction level modeling (TLM) explicitly exposes inherent parallelism in the application by modeling concurrency, hierarchy, synchronization and timing. TLM guidelines use different methods to model communication of concurrent modules in the design. TLM-1 focuses on modeling communication using channels and TLM-2.0 focuses on modeling address-accurate communication using memory-mapped buses. The choice of synchronization and communication mechanisms in TLM models affect the available level of parallelism. Parallel Discrete Event Simulation (PDES) is an attractive approach to measure parallelism of TLM models and compare simulation performance between models using different parallelization techniques [2, 3].

3.1 Introduction

Exponential growth of computational requirements of new emerging applications such as deep learning, puts an extra demand on finding parallelism opportunities and simulation performance. To this end, fast and yet standard-compliant simulation of design candidates will enable rapid design space exploration and hence, shorter time to market.

In this chapter, we propose a set of IEEE standard-compliant modeling techniques to increase available parallelism in SystemC TLM-1 and TLM-2.0 models for parallel discrete event simulation. As shown in Figure 3.1, we illustrate simulator parallelism, model parallelism and simulation speedup in a 3-dimensional space. As the red arrow indicates, both higher model parallelism and simulator parallelism achieve the maximum simulation speedup. Moreover, by increasing model parallelism opportunities in one dimension, the simulator can better leverage its parallelization capabilities for the maximum simulation speedup. In particular, we demonstrate our proposed techniques on SystemC TLM models of a DNN using out-of-order parallel simulation.

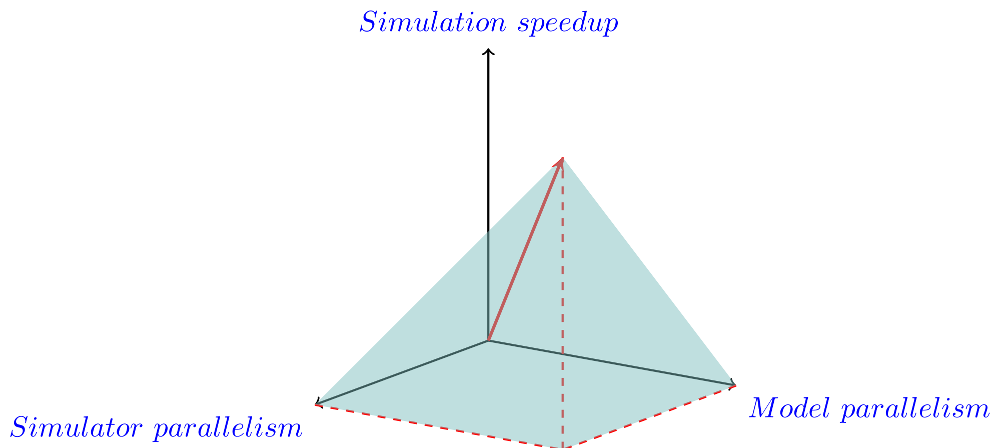


Figure 3.1: Simulator parallelism, model parallelism and simulation speedup forms a 3-dimensional space[2]

Our key contributions in this chapter are as follows:

(1) A systematic analysis of parallelism opportunities in SystemC TLM-1 and TLM-2.0

models of a representative DNN (GoogLeNet) for parallel simulation

(2) A proposal of less restrictive communication mechanisms and transaction types for enhanced parallelism with out-of-order parallel simulation of TLM-1 and TLM-2.0 models

(3) Experimental results that demonstrate the improved parallelism in a given reference model [1] with simulator run time reduced by 38%

3.2 Parallelism in TLM-1 Modeling

Following the distinction between simulator parallelism and model parallelism introduced in Figure 3.1, we propose alternative channel constructs to increase parallelism opportunities. We also analyze how the number of buffers inside channels can increase parallel simulation performance.

3.2.1 TLM-1 Modeling of DNNs

TLM-1 implements message-passing semantics with the primary purpose to separate communication from computation. Through well-defined TLM-1 interface method calls, any internal state changes in one SystemC module are hidden from other modules [27]. Following TLM-1 coding style, channels are modeled as queues with FIFO semantics, allowing to consume/produce data in a first-in, first-out discipline. These channels implement interface methods for `read` and `write` access. By encapsulating communication in channels, various communication mechanisms and buffer sizes can be modeled independently from the module functionality. This exploratory approach provides early feedback on the amount of available parallelism and local communication interactions.

3.2.2 Channel type

Starting from the model proposed in the previous chapter (Section 2.3), we improve the communication and synchronization mechanism in this chapter. To this end, we propose three channel types according to the channel synchronization mechanism:

1. *Blocking channel*: In a blocking channel, synchronization is handled using a set of two `wait` statements in read and write access functions.
2. *Non-blocking channel*: In a non-blocking channel, the write access function does not block and synchronization is handled using only one `wait` statement in the read access function.
3. *SystemC FIFO channel*: This channel is built on the predefined primitive channel `sc_fifo` with default read and write member functions which use the `request_update` mechanism.

In a blocking channel two `sc_event`s ensure synchronization between each consumer and producer. In a non-blocking channel, we design a synchronization scheme between producer and consumer that uses only one `wait` statement and one `sc_event`. Lastly, we design SystemC channels that do not require any calls to the `wait` construct. The improved communication techniques increase the potential that an out-of-order PDES simulator schedules threads more aggressively.

3.2.3 Buffering scheme

The TLM-1 model of a data processing application can be considered as a graph data structure with modules as nodes and channels as edges connecting neighboring modules. Each module continuously fetches data from its input channel(s), processes the data and writes

its result(s) into output channel(s). These data processing modules often form a pipeline structure that execute in parallel while buffers in channels hold intermediate results between pipeline stages. The more buffers exist in the channels, the more possibilities there are for pipelining of data in the graph. This gives a parallel simulator more freedom to schedule even more parallel threads at the same delta cycle.

In particular, having only a single buffer inside blocking channels, modules can only process data in every other delta cycle. However, with double buffers inside channels, a producer can write to the back buffer while a consumer can read from the front buffer. This results in more active threads that perform their tasks in fewer delta cycles. This increase in the level of parallelism gives the parallel simulator more opportunities to aggressively schedule threads and minimize simulation run time.

Figure 3.2 illustrates the inception module, the main building block of GoogLeNet. The inception module forms an unbalanced graph structure with four parallel tracks, each running a different workload. Note that the four parallel tracks in each inception module contain (2, 4, 4, 3) modules to process, respectively. In the absence of a balanced graph topology, the number of buffers in channels should address the imbalance to enable the maximum number of modules to run in parallel.

A TLM-1 modeling diagram of the inception module with double buffering scheme is shown in Figure 3.3. As shown, modules read/write data from/to channels via their input port(s) and output port(s). Note that the output channels for **relu_1x1** and **relu_pool_proj** keep 4 and 3 buffers, respectively, to compensate for the unbalanced graph structure.

In the case of non-blocking channels, the `write` access method does not incur any `wait` statement. Therefore, the number of buffers in non-blocking channels needs to be increased to avoid any buffer overflow.

Overall GoogLeNet forms a graph with a depth of 62 layers. In the worst case scenario, all

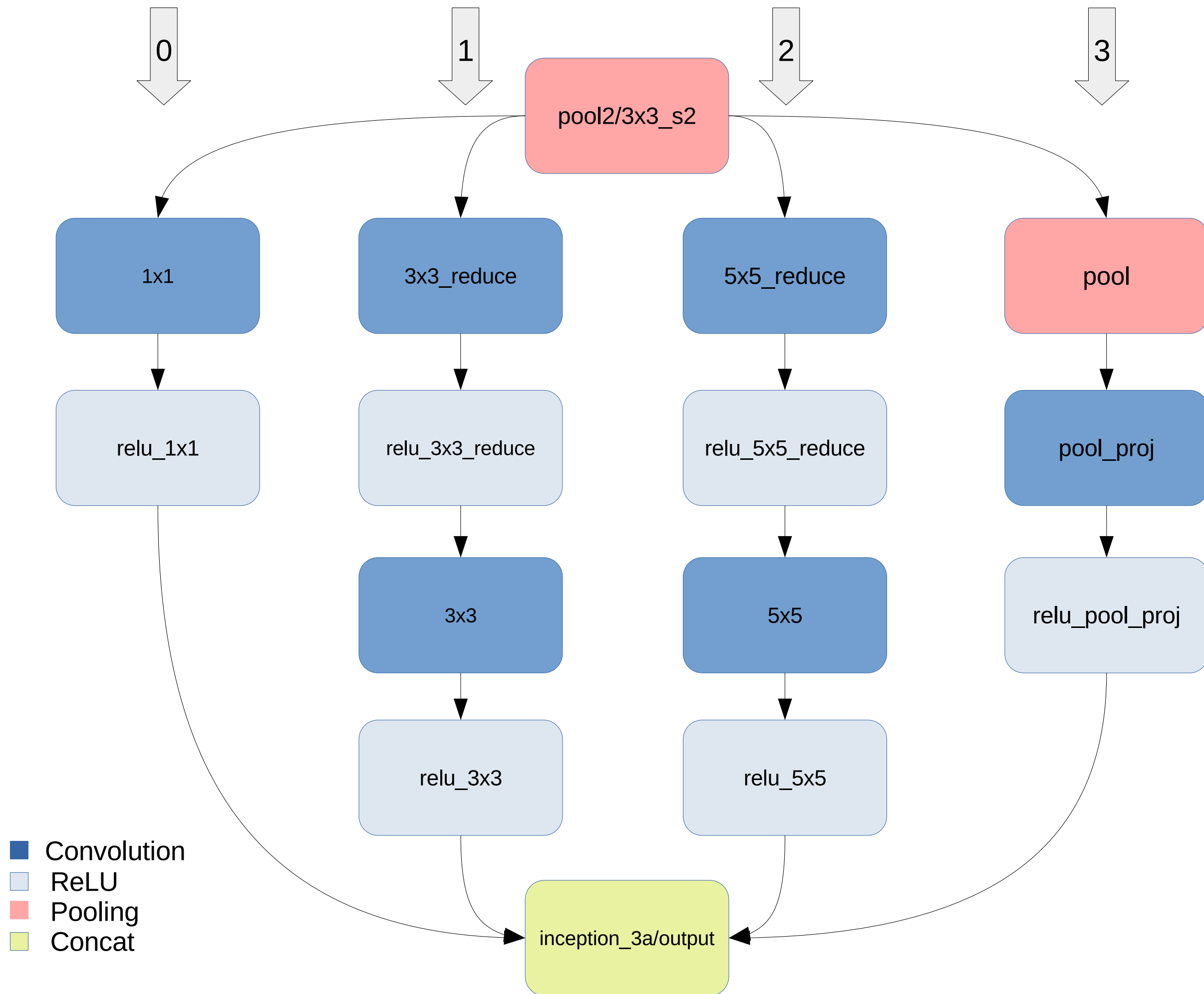


Figure 3.2: Inception module in GoogLeNet

producer layers in each level of the graph write to the channel before consumer layers read the data. To dimension the channel sizes for this worst case scenario, non-blocking channels should have space for the maximum depth of the graph plus one for the stimulus module, namely 63 buffer elements.

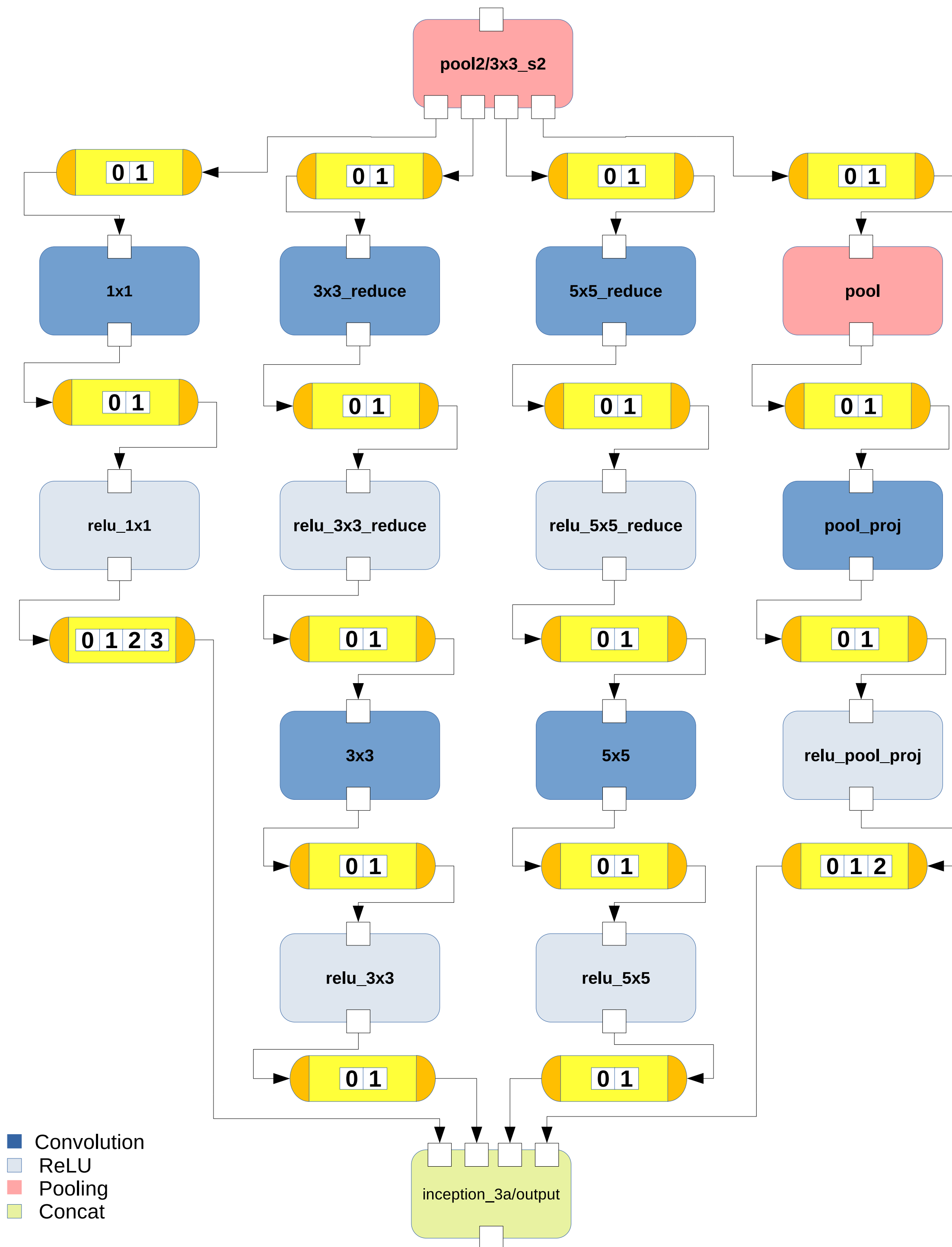


Figure 3.3: TLM-1 model diagram of inception module in GoogLeNet

3.3 Parallelism in TLM-2.0 Modeling

While TLM-1 gives early feedback on parallelism and local communication, it is not specifically intended for bus modeling, interoperability and architectural exploration. SystemC TLM-2.0 introduces generic payload and core transport interfaces for the abstract modeling of memory-mapped buses. However, the notion of channels from TLM-1 has disappeared from TLM-2.0 modeling and each module instead uses pointers to access memory locations in other modules. The lack of an encapsulating channel construct allows simulation threads to directly access data of other modules, making synchronization of such accesses a difficult task for parallel simulators in a standard-compliant fashion [12] and has been identified as an obstacle for safe and fast parallel simulation [15]. On the other hand, TLM-2.0 models feature address-accurate memories.

In this section, we briefly describe the TLM-2.0 modeling of data processing applications such as DNN. Later, we introduce a feed-forward events mechanism that can be used for synchronous parallel simulation. Lastly, we propose the back-pressure events mechanism devised for safe out-of-order parallel simulation.

3.3.1 TLM-2.0 Modeling of DNNs

TLM-2.0 introduces a generic payload and blocking/non-blocking transport interfaces for the abstract modeling of memory-mapped buses. In TLM-2.0 modeling, a socket should be instantiated within each initiator and each target for every transaction level connection. Therefore, module input and output ports in TLM-1 models are replaced with an initiator socket. The generic payload captures the information to pass with each bus transaction between initiator and target. The initiator module instantiates the generic payload transaction object and sets its attributes before passing a reference to this object to a target module via

its transport interface.

In our proposed TLM-2.0 models, the initiator sockets are connected to target sockets of a shared memory. Figure 3.4 shows the connections of initiator sockets of layers in GoogLeNet to target sockets of shared TLM-2.0 memory. Each module has a dedicated address space in the memory to read and write its buffers. This model uses a blocking transport interface (`b_transport`) to pass transactions between initiator and target memory. The transaction is a `tlm_generic_payload` object and its data pointer points to the start address of the input/output buffer. The data length of the generic payload is set to the entire buffer inside the shared memory. Since the model is untimed, the timing annotation argument of `b_transport` is set to a delta cycle.

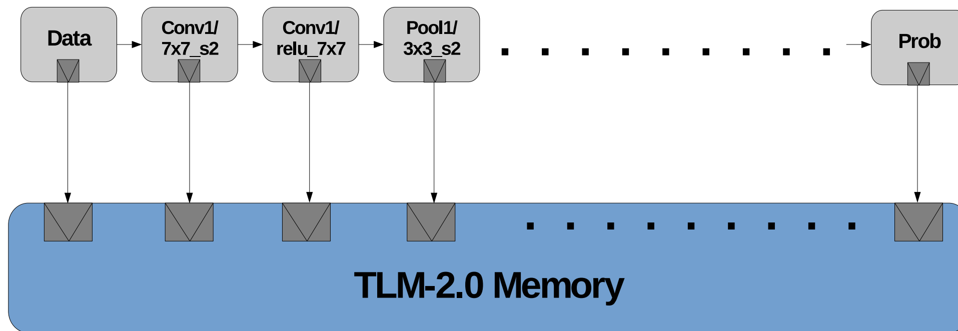
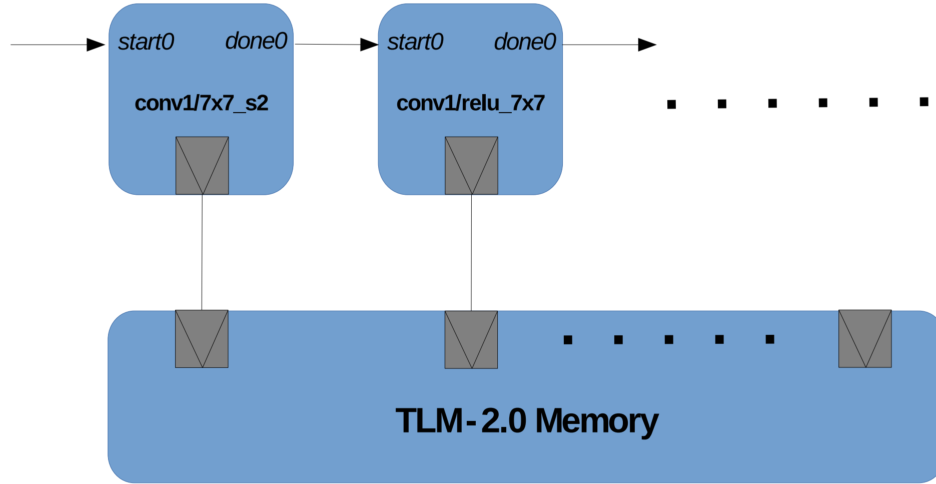


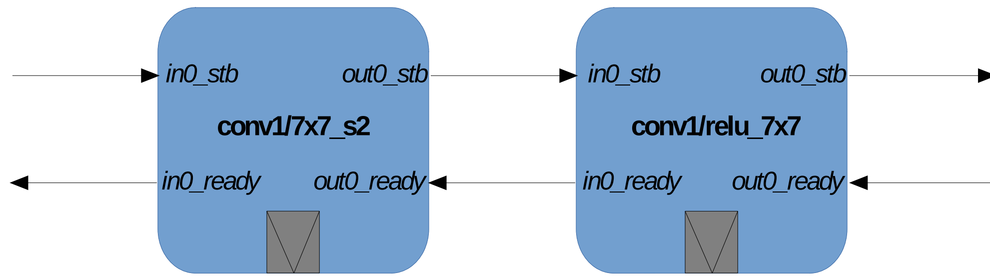
Figure 3.4: TLM-2.0 model diagram of GoogLeNet

3.3.2 Feed-forward events mechanism

The communication mechanism in a feed-forward model is as follows: each producer places its output into a buffer in the shared memory. Each consumer reads its input from the same shared buffer. To avoid race conditions, each consumer waits for an event notification from its producer. The arrows between the modules in Figure 3.5a illustrate this feed-forward notification mechanism. Therefore, each module contains one `sc_event` for each input (`start`) and each output (`done`). For example, as seen in Figure 3.5a, the `start` event of `conv1/relu_7x7` is connected to `done` event of `conv1_7x7_s2`. Furthermore, since



(a)



(b)

Figure 3.5: (a) TLM-2.0 feed-forward model connections (b) TLM-2.0 back-pressure events connections

adjacent modules share a common buffer inside the memory, the memory footprint of the model is minimized.

In the feed-forward model with only a single buffer between neighboring modules, modules only accept and process data every other delta cycle. By increasing the number of buffers between modules, modules can instead process data every delta cycle. This gives the parallel simulator the opportunity to schedule more threads in each delta cycle, utilizing available parallelism in the processor for minimizing the simulation run time. Moreover, such a model achieves its maximum theoretical throughput, generating output every delta cycle.

In the absence of a balanced graph topology, event synchronization for multiple producers or multiple consumers requires delta-cycle delay compensation. Such behavior can be seen in every inception module in GoogeLeNet, as shown in Figure 3.2. Note that the four parallel tracks contain 2, 4, 4, and 3 modules, respectively. Since events occur at precise points in simulation time, our proposed untimed model compensates for these irregularities to guarantee correct synchronization between the modules. To guarantee correct event synchronizations, delay elements must be inserted in those tracks with less modules to form a balanced graph structure. This means 2 delay elements in track 0 and 1 delay element in track 3 are required. Furthermore, the output of the last modules in tracks with less modules require extra buffers to store results generated during those delay cycles. This means for supporting double buffering, the last module in track 0, `relu_1x1`, and the last module in track 3, `relu_pool_proj`, require 4 and 3 output buffers, respectively. These extra buffers ensure a continuous stream of data in every delta cycle into the design, increasing model parallelism and maximizing model throughput. This forms the model **E** in Figure 3.6.

3.3.3 Back-pressure events mechanism

The model without back-pressure mechanism is not safe for aggressive out-of-order scheduling. Only a conservative in-order scheduling approach will execute the feed-forward TLM-2.0 model correctly due to the missing back-pressure mechanism. Therefore, we devise a back-pressure events mechanism to safely execute the TLM-2.0 model in the aggressive OoO parallel simulation for maximum speedup.

Event connections for the first convolution and ReLU layers in GoogLeNet are depicted in Figure 3.5b. Each module has a set of two `sc_event`s for each input and output. The `stb` event is notified once a module has valid data inside the memory to be read and the `ready` event signals a module is ready to read new data. By connecting events between all

subsequent modules, the model forms a robust back-pressure mechanism that safely controls the flow of data inside the pipeline.

Support for the back-pressure events mechanism is extended to all neighboring modules in the TLM-2.0 model. The double-buffering scheme guarantees a continuous stream of data inside the design pipeline, maximizing model parallelism and model throughput with the minimum number of buffers in the memory. This forms the TLM-2.0 untimed model with back-pressure events mechanism. This forms model the **F** in Figure 3.6.

3.4 Parallelism Direction

To demonstrate degrees of freedom for parallel simulators to find parallelism opportunities in TLM-1 and TLM-2.0 models, we create an XY chart with communication mechanism and buffering scheme on x- and y-axes, respectively. As depicted in Figure 3.6, we map the number of buffers on the x-axis and communication mechanism on the y-axis. On the x-axis, *min* refers to a single buffer, *mul* to double buffer taking into account that certain layers requires multiple buffers due to the imbalanced graph structure, and *max* refers to the total depth of TLM-1 model graph. On the y-axis, we map communication mechanisms from the most restrictive type for parallel simulation, namely, *TLM-1 blocking*, to the least restrictive type, namely, *TLM-2.0 back-pressure*. Increasing the number of buffers and utilizing communication with less restrictive synchronization mechanisms creates more freedom for out-of-order simulators to schedule threads in different delta cycles. This maximizes multi-core utilization and hence results in shorter simulator run time.

Given the proposed communication mechanisms and buffering schemes, we have designed a set of TLM-1 and TLM-2.0 models for GoogLeNet. As marked in Figure 3.6, the reference Model **A** [1] uses blocking channels with only a single buffer in channels. We designed Model

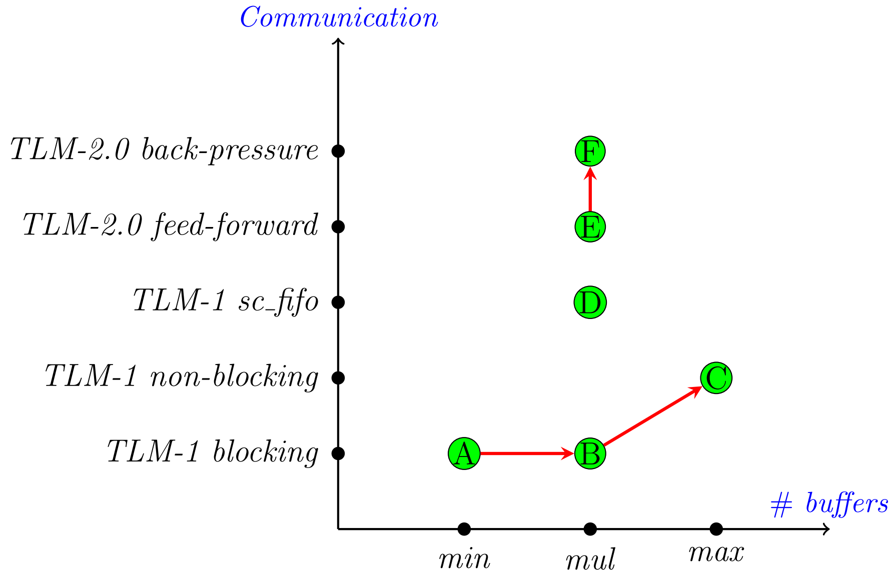


Figure 3.6: Communication mechanism versus number of available buffers in TLM models

B with a double-buffering scheme with blocking-channels. In Model **C**, we replaced blocking channels with non-blocking channels with buffer size of 63 elements, the total depth of the graph. We designed Model **D** using `sc_fifo`s with a double-buffering scheme. Model **E** is a TLM-2.0 model that uses the feed-forward events mechanism as an inter-module communication and modules have double-buffers inside a shared memory. Finally, we designed Model **F** using our back-pressure mechanism to guarantee safe communication between modules for aggressive out-of-order scheduling with double buffers for maximum parallelism and maximum throughput. Table 3.1 summarizes the properties of all designed TLM-1 and TLM-2.0 models.

Table 3.1: TLM models summary

Model name	Standard	Communication	Buffers
Model A [1]	TLM-1	Blocking channels	Single buffer
Model B	TLM-1	Blocking channels	Double buffers
Model C	TLM-1	Non-blocking channels	Buffer size of 63
Model D	TLM-1	SystemC FIFO channels	Double buffers
Model E	TLM-2.0	Feed-forward	Double buffers
Model F	TLM-2.0	Back-pressure	Double buffers

Since each SystemC module has specific attributes based on its layer type and its corresponding TLM modeling style, writing module definitions by hand is a tremendously error-prone task. Furthermore, interconnecting all modules at the top level using either queues or events is a tedious task. Therefore, we have extended the generator tool from Model A [1] to automatically generate all the other TLM-1 and TLM-2.0 models based on modeling style, communication type and buffer architecture. In the case of TLM-2.0 models, the Python 3 generator automatically produces an address map file based on buffer architecture and supports memory address generation for multiple buffers for any layer in the network.

3.5 Experimental Measurements and Results

Parallelism opportunities introduced in transaction-level SystemC models can be quantified and measured using a SystemC parallel simulator. To exploit the available parallelism in our TLM-1 and TLM-2.0 models of GoogLeNet, we describe our extensive measurement results using RISC and provide valuable insights gained from analyzing the results.

3.5.1 Simulation Setup

We use a 16-core host computer platform with hyper-threading technology (HTT) to benchmark the simulations. The specifications of the platform are shown in Table 3.2. To have reproducible experiments, the Linux CPU scaling governor is set to ‘performance’ to run all cores at the maximum frequency, and file I/O operations, i.e. *cout*, are minimized.

Table 3.2: Platform specification

Platform name	16-core (Phi)	32-core (Phi HT)
OS	CentOS 6.10	CentOS 6.10
CPU Model name	Intel Xeon E5-2680	Intel Xeon E5-2680
CPU frequency	2.7 GHz	2.7 GHz
#cores	8	8
#processors	2	2
#threads per cores	1	2

3.5.2 Simulation Results

For benchmarking, we measure simulator run time using Linux */usr/bin/time* under CentOS 7. Measurements are reported for the sequential SystemC simulation using Accellera SystemC and for the parallel simulations using RISC simulator V0.6.2 in three modes: synchronous (SYN), non-prediction (NPD) and out-of-order (OOO) parallelism. For reliability of the results, each measurement is performed three times. Later, if the distance of each recorded value from its median is greater than ±2%, that entire measurement is ignored.

We analyze the measurement results obtained from the simulations of six TLM models. We create various heat map tables to identify the relevant results regarding parallelism in transaction types and transaction level modeling as follows:

(A) Less restrictive transaction types enable higher parallelism

Table 3.3 shows the elapsed time of the models in SYN, NPD and OOO simulation modes using RISC V0.6.2.

Considering the 16-core machine (phi), model A uses blocking channels with a single buffer. The elapsed time of Model A for the SYN mode is the highest. Model B uses multiple

Phi			
Elapsed time	SYN	NPD	OOO
Model A	266.66	194.77	193.94
Model B	229.04	193.63	193.08
Model C	231.79	220.32	200.29
Model D	197.02	197.93	194.90
Model E	198.31		
Model F	198.66	199.31	170.44

Phi HT			
Elapsed time	SYN	NPD	OOO
Model A	276.04	198.40	197.17
Model B	237.69	196.73	197.47
Model C	235.12	224.39	202.90
Model D	201.13	195.35	195.40
Model E	203.29		
Model F	203.17	204.89	170.68

Table 3.3: Measurements of elapsed time for parallel simulations (color scale red-to-green means slow-to-fast)

buffers to increase the potential for pipelining. Model C removes wait statements in the write function to let the OOO scheduler schedule multiple threads together. Model D uses SystemC FIFOs to implement channels. SystemC FIFO forces synchronous simulation. Hence, the elapsed time of Model D for SYN, NPD and OOO are almost identical as reflected in the fourth row. Model E and Model F are TLM-2.0 models without any usage of primitive channels. As previously stated, Model E is not safe for out-of-order parallel simulation, so elapsed time for NPD and OOO simulations are not reported for this model. As can be seen in the second to six rows, the elapsed time for SYN simulation mode decreases steadily. However, the aggressive OOO simulation exploits the maximum parallelism introduced in each model and reports the shortest elapsed time for Model F. The exact same pattern applies to the other TLM-1 and TLM-2.0 models on machines with a higher number of logical cores.

Notably, our efforts on increasing the potential of parallelism in the models pay off with a

significant simulation speedup. Comparing the synchronous reference TLM-1 model (276.04s with hyper-threading enabled) with the OOO simulation of the TLM-2.0 model with safe back-pressure (170.68s), shows the simulator run time reduced by 38%. Note that this applies despite the higher workload the TLM-2.0 models carry, as we will show in the next section.

(B) Abstract TLM-1 models carry less workload than memory accurate TLM-2.0 models

Table 3.4 shows the heat map table for elapsed time of all six models in sequential simulation mode. The last two rows for TLM-2.0 models indicate longer elapsed time than the first four rows for TLM-1 models. This can be explained due to the difference in number of memory copies in TLM-1 and TLM-2.0 models. TLM-1 models use shallow copy for storing/loading items in/from channels. However, TLM-2.0 models use two memory copies to read and write from/to the memory module. This distinction shows that the actual simulator workload for the TLM-2.0 models has increased in comparison to the TLM-1 models.

	Phi	Phi HT
Elapsed time	SEQ	SEQ
Model A	949.02	949.56
Model B	940.12	939.61
Model C	941.93	940.93
Model D	945.24	943.98
Model E	956.28	955.81
Model F	956.28	956.01

Table 3.4: Measurements of elapsed time for SEQ execution (color scale green-to-red means increasing workload)

(C) Increased parallel simulation performance indicates better models with higher amount of parallelism exposed

Figure 3.7 illustrates simulation speedups in different parallel simulation modes on our 16-core host (phi). As shown, the maximum simulation speedup (5.6x) is achieved by model F in out-of-order (OoO) parallel simulation mode. This indicates model F has the highest level of parallelism available in comparison with other TLM models. Therefore, model F is the right design candidate for further model refinements and lower-level implementation.

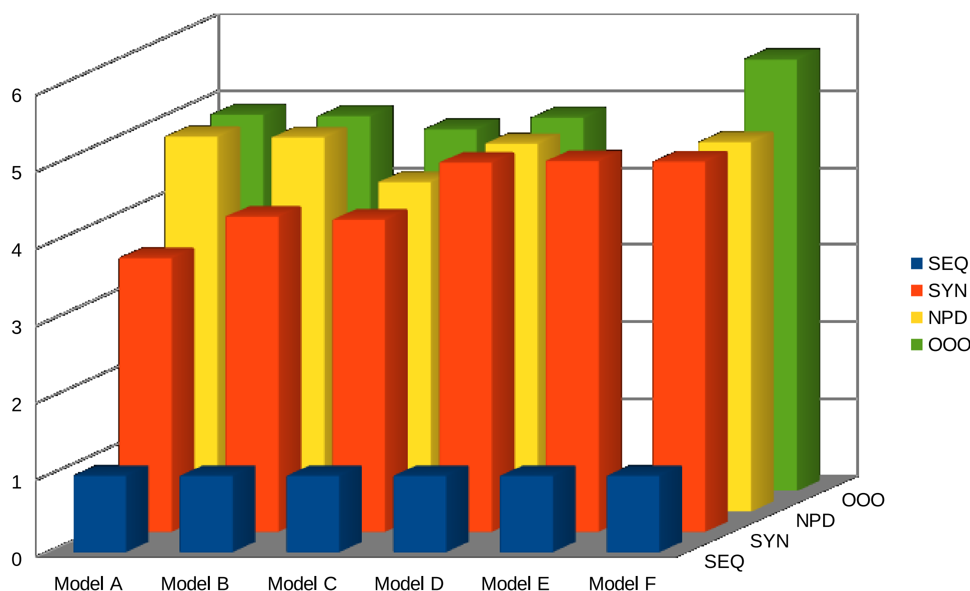


Figure 3.7: Simulation speedup for different parallel simulation modes on a 16-core host

3.6 Conclusion

In this chapter, the impact of synchronization and communication mechanisms on available parallelism in transaction level modeling (TLM) has been studied. We have demonstrated the impact of varying synchronization mechanisms on the exposed parallelism using six modeling styles of a state-of-art deep neural network (DNN), GoogLeNet. We further have quanti-

fied the improved parallelism in the improved SystemC TLM-1 and TLM-2.0 models by measuring the performance of aggressive out-of-order parallel simulation in the Recoding Infrastructure of SystemC (RISC). The experimental results show that our standard-compliant parallelization techniques result in a significantly increased simulation speed up to 5.6x on a 16-core host machine. Notably, the results support the hypothesis that higher speed in aggressive parallel simulation is a significant indicator of higher level of parallelism in design models which enables better implementation at later stages in the design flow.

While this chapter has focused on communication and synchronization aspects, we explore timed models at lower abstraction, and include a wider range of DNN applications in the following chapters.

Chapter 4

Fast Loosely-Timed System Models with Accurate Memory Contention

Emerging computing applications create an ever-increasing demand for higher memory bandwidth and lower access latency. While massively parallel processor arrays [48, 70] allow an order-of-magnitude improvement in computational capacity, a severe performance gap exists in the state-of-the-art memory architectures. Additionally, the low-power requirements of embedded systems create extra design challenges to achieve on-par performance improvements.

Advances in 3-D integration of memory-logic fabrics [56] and continuing trends towards data-intensive applications necessitate new hardware-software codesign approaches with particular emphasis on memory contention. A system-level memory-aware modeling framework is a cornerstone to build the next generation system-on-chips, capable of addressing memory bandwidth and latency issues. Such a modeling framework provides the means to identify memory bottlenecks and explore new architectures prior to RTL implementation with a high degree of accuracy and faster simulation speed [45].

4.1 Introduction

In Chapter 3, we have studied the impact of communication mechanisms on the available parallelism in transaction level modeling (TLM). Specifically, we have demonstrated the impact of varying synchronization mechanisms and buffering schemes on the exposed parallelism using different modeling styles of a deep neural network (DNN), GoogLeNet. In this Phase I modeling, six untimed SystemC TLM-1 and TLM-2.0 models have been developed. Figure 4.1 places the six models generated in this Phase I, model **A** to **F** (green nodes), in a chart with the number of buffers indicated on the x-axis and the communication mechanism on the y-axis. We have further quantified the improved parallelism in the above models by measuring the performance of aggressive out-of-order parallel simulation in the Recoding Infrastructure of SystemC (RISC) [43]. As a result, we have demonstrated that the design with the highest amount of parallelism exposed, i.e. model **F**, is suited best for further refinement in the system design flow [2].

Expanding on our prior chapter, this chapter explores the critical aspects of modeling and analysis of timing accuracy and memory contention. In this Phase II modeling, we further refine the untimed *TLM-2.0 back-pressure* model **F** with *double-buffering* to loosely-timed (**G**) and approximately-timed (**H**) models. Moreover, we propose a new loosely-timed contention-aware modeling style to expose memory contention in a fast and yet accurate manner (model **I**).

Furthermore, we define a system-level exploration framework to automatically generate TLM from an abstract DNN specification. As illustrated in Figure 4.2, the DNN specification together with modeling parameters constitute the inputs to our proposed model generator, `netspec`. Based on user-specified design metrics, `netspec` (green box) automatically creates models at desired abstraction levels.

In general, TLM trades off timing accuracy for the sake of simulation speed. This allows

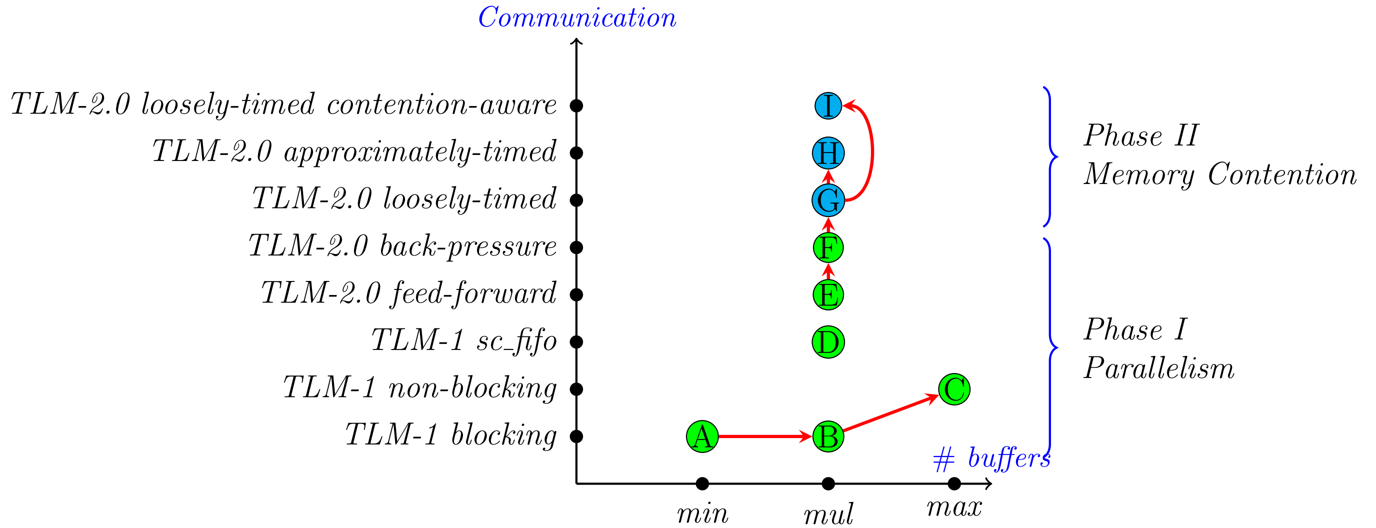


Figure 4.1: Transaction Level Modeling (TLM) of GoogLeNet DNN with focus on exposing parallelism (Phase I) and memory contention (Phase II)

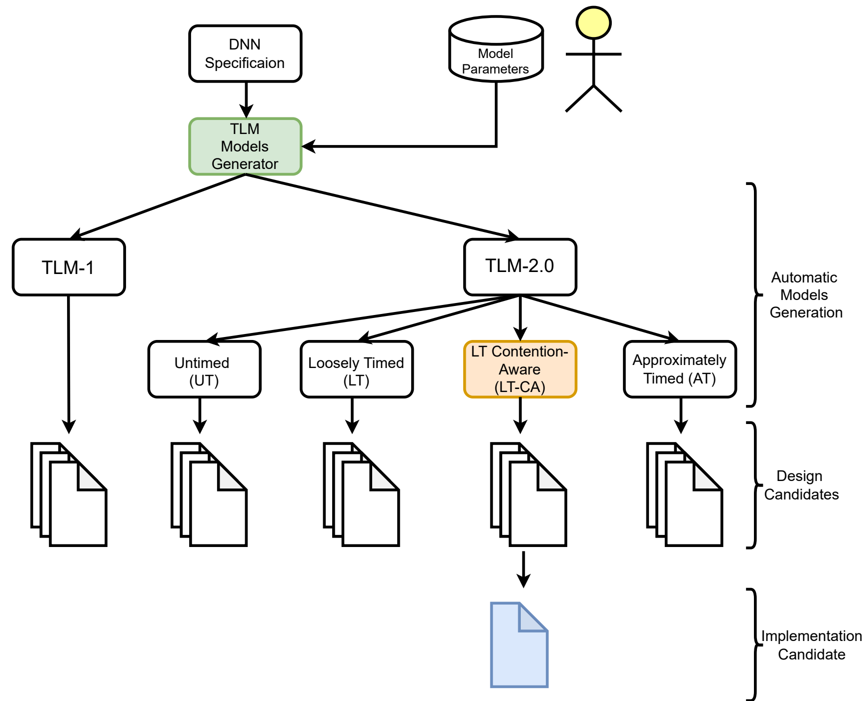


Figure 4.2: Deep neural network (DNN) transaction level modeling (TLM) exploration framework

system designers and chip architects to rapidly prototype and verify their design candidates before generating detailed RTL. RTL simulations tend to be orders of magnitude slower

than SystemC TLM. Traditionally, higher-level abstractions of TLM (i.e. loosely-timed, LT) mainly focus on functionality and define the programmer's view of the design for early software development. On the other hand, lower-level TLM (approximately-timed, AT) can represent finer-grained timing details at the price of sacrificing simulation speed. However, with the enormous increase in today's design complexity, running lower-level models has turned into a severe obstacle in agile hardware development. Accurate *and* fast high-level TLM that can expose the critical aspect of memory contention without sacrificing simulation performance is needed to efficiently build future computing platforms (orange box in Figure 4.2).

Having a fast and accurate contention model, design candidates can be rapidly evaluated for metrics such as programmability, performance, power, etc. for a lower-level implementation, e.g. RTL (blue box in Figure 4.2). Here, a data visualization tool that can generate transaction-level timing diagrams for early feedback to system designers is beneficial to analyze and address any memory contention in the design.

To summarize, the key contributions in this chapter are the following:

(1) A novel system-level modeling framework and automatic SystemC model generator for design space exploration (DSE) with focus on mitigating memory contention, lowering memory footprint, and increasing performance of DNNs (green box)

(2) Early contention modeling in SystemC loosely-timed (LT) models with high accuracy and fast simulation speed (orange box)

(3) Extensive performance measurement results and data visualization to generate transaction-level timing diagrams for memory contention analysis (blue box)

The rest of this chapter is organized as follows: In Section 4.2, we present the latency model for memory, computation and interconnect following the LT coding style, model **G** in Figure

4.1. In Section 4.3, we refine the entire model following the AT coding style to achieve higher timing accuracy, model **H** in Figure 4.1. In Section 4.4, we describe our novel method for loosely-timed interconnect and memory contention modeling that simulates as fast as a LT model yet shows memory contention as accurate as an AT model, model **I** in Figure 4.1. Section 4.5 describes the structure of our TLM generator for DSE. Finally, we present our extensive results and analysis in Section 4.6 and conclude this study in Section 4.7.

Our previous study in Section 2.5.3 shows that the multi-threaded OpenCV library delivers the highest level of parallelism for simulation speedup compared to existing thread-level parallelism at SystemC level. Therefore, we rely on multi-threaded OpenCV with a sequential SystemC simulator in this work instead of using a parallel simulator such as the Recoding Infrastructure for SystemC (RISC) [43] for better simulation performance. Moreover, RISC does not yet have the support for all language constructs needed for approximately-timed modeling.

4.2 TLM-2.0 Loosely-timed (LT) Model

Given that the TLM-2.0 untimed model **F** provides only causal ordering between processes, timing is introduced at the next lower abstraction level. Our LT approach models the start and end times of a transaction using the blocking transport interface with a timing annotation, providing a good trade-off between timing accuracy and simulation speed.

The LT model adds three sources of latency: (1) memory, (2) computation, and (3) interconnect. Our step-wise approach incrementally refines the model, adding one source of latency at each step.

4.2.1 Memory

To add memory latency, the inter-module communication must be revisited. Since events occur at precise points in simulation time, as soon as a consumer incurs a delay due to a memory access, it would miss events from the producer. Therefore we replace the feed-forward event notification with a pair of `sc_signals` for each input and output in every module (Figure 4.3). Once a producer fills the shared buffer, it increments the `num_sent` output signal to inform the waiting consumer that new data is available. When the consumer finishes reading the buffer, it increments the `num_rcvd` input signal. To implement a back-pressure mechanism, the module waits with the new write transaction when the output buffer is full.

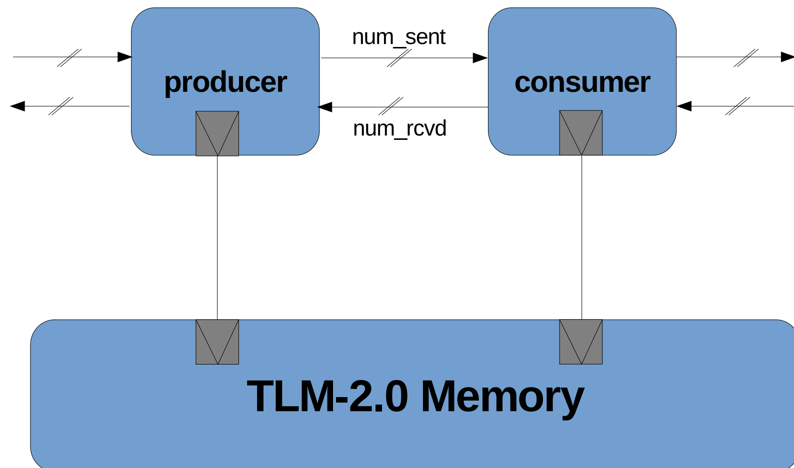


Figure 4.3: TLM-2.0 LT model module connections

Algorithm 1 lists the pseudo-code used in our TLM-2.0 LT model. The initiator module writes the layer output using `b_transport` send transactions to the target memory. When the memory serves the request, it updates the delay object inside the timing annotation argument with the memory latency and returns immediately, because the simulation is faster when `b_transport` does not block. The memory latency value depends on the access type and the data size of the transaction and is configurable for each LT memory module.

Algorithm 1: Main thread in each module in a TLM-2.0 LT model

```
while (1) do
    while (in_num_sent - in_num_rcvd == 0) do
        wait(in_num_sent.value_changed_event()); // wait until there is a
         buffer to read
    end
    read layer input via an initiator socket;
    wait for read latency;
    in_num_rcvd++; // signal producer that the buffer has been read
    run layer computation;
    wait for computation latency;
    while (out_num_sent - out_num_rcvd >= num_out_buf) do
        wait(out_num_rcvd.value_changed_event()); // wait until there is a
         free slot to write
    end
    write layer output via an initiator socket;
    wait for write latency;
    out_num_sent++; // signal consumer that a new buffer is ready to be
     read
end
```

4.2.2 Computation

To estimate the computational latency, we analyze the computational complexity of the most common constituent layers in a DNN in terms of the number of multiplications (N_{mul}) and the number of additions (N_{add}). Given a 32-bit single-precision floating-point multiply-accumulate (FP32-MAC) unit available, we assume the total computational latency of a layer to be the product of the number of MAC operations and the inverse of the peak floating-point operations per second (FLOPS): $N_{MAC} \cdot \frac{s}{flop}$. Here, the peak FLOPS value is the maximum number of single-precision floating-point MAC operations that a processing element (PE) can perform per second. A PE is a basic arithmetic component that at least includes a 32-bit floating-point multiplier and an accumulator register. It is worth mentioning that the maximum throughput of a PE is the main focus at this stage of modeling. In other words, implementation details of the PE, such as its clock frequency, number of parallel MAC units, and the amount of control logic and congestion overhead, are all abstracted away.

We describe the timing estimation separately for each layer type. The size of the input volume to each layer is $W_i \times H_i \times C_i$ where W_i, H_i and C_i represent the width, height and number of channels, respectively.

Convolution Convolution has the following hyper-parameters: number of filters K, kernel size F, stride S and padding P. Convolution has also learned parameters, weights and biases. The total number of weights is $F \cdot F \cdot C_i \cdot K$ and the total number of biases is K. Convolution produces an output volume of size $W_o \times H_o \times C_o$ where $W_o = \lfloor \frac{W_i - F + 2 \cdot P}{S} + 1 \rfloor$, $H_o = \lfloor \frac{H_i - F + 2 \cdot P}{S} + 1 \rfloor$ and $C_o = K$. To compute one output element for one channel, $N_{mul_{elem}} = F \cdot F$ and $N_{add_{elem}} = F \cdot F - 1$. To compute one output element for all channels, $N_{mul_{chans}} = C_i \cdot N_{mul_{elem}} = C_i \cdot F \cdot F$ and $N_{add_{chans}} = C_i \cdot N_{add_{elem}} + C_i - 1 + 1 = C \cdot F \cdot F$ where the extra addition is for adding the bias value. To compute all output elements for one filter, $N_{mul_{filter}} = W_o \times H_o \times N_{mul_{chans}} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2}{S^2}$ and $N_{add_{filter}} = W_o \times H_o \times N_{add_{chans}} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2}{S^2}$. To compute all output elements for all K filters, $N_{mul} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2 \cdot K}{S^2}$ and $N_{add} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2 \cdot K}{S^2}$.

Rectifier Linear Unit (ReLU) The ReLU is an activation function defined as the positive part of its argument ($max(0, x)$). This unit is implemented by a comparator that can be simply modeled as an adder. Therefore, $N_{add} = W_i \cdot H_i \cdot C_i$.

Pooling To reduce the spatial size of volumes in the network, pooling down-samples the input volume by choosing the maximum element inside the kernel. This unit has two hyper-parameters: kernel size (F) and stride (S). Pooling produces an output volume of size $W_o \times H_o \times C_o$ where $W_o = \lfloor \frac{W_i - F}{S} + 1 \rfloor$, $H_o = \lfloor \frac{H_i - F}{S} + 1 \rfloor$ and $C_o = C_i$. To find the maximum element inside a kernel, it requires $N_{add_{elem}} = F \cdot F$. To compute the output for one channel, $N_{add_{chan}} = W_o \times H_o \times N_{add_{elem}} \approx \frac{W_i \cdot H_i \cdot F^2}{S^2}$. The total number of additions to compute the output for all channels is $N_{add} \approx \frac{W_i \cdot H_i \cdot Ci \cdot F^2}{S^2}$.

Concat A concat layer concatenates two or more volumes and does not perform any computation on the inputs. Hence, its computational complexity is zero.

Since the majority of layer types in our example has been considered as shown in Table 1.1, the computational complexities of the remaining layers are deferred for now. Furthermore, the peak computational capacity available in each layer is also configurable.

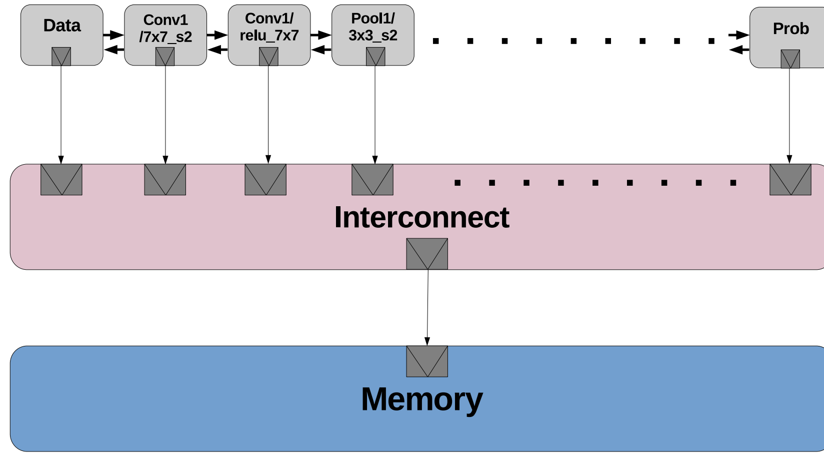
4.2.3 Interconnect

To refine a dedicated point-to-point communication between a layer and a memory, we design a generic TLM-2.0 LT interconnect module. The interconnect module arbitrates and forwards existing transaction objects from initiator layers to a target memory. The interconnect can also model the latency to accept transaction objects before forwarding them to the memory.

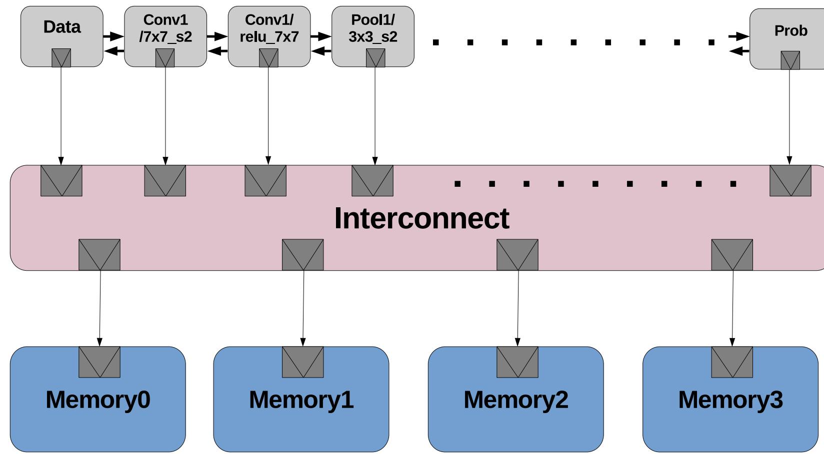
As an example depicted in Figure 4.4a, the interconnect is placed between the 142 initiator layers and a single target memory. In this architecture, the interconnect has 142 target sockets and one initiator socket. The thick double arrows represent the inter-module communication mechanism supporting the timing in the model. Furthermore, the interconnect supports the modeling of multiple memories. As another example illustrated in Figure 4.4b, the interconnect is placed between four separate memories with segmented address space.

The main functionality of the interconnect is to route transactions from an incoming target socket to an outgoing initiator socket. Since each memory has a dedicated address space, the interconnect routes transactions to the correct memory depending on the address embedded in the transaction. After address translation, the interconnect forwards the transaction via the corresponding initiator socket connected to the memory.

To allow maximum flexibility in the interconnect, we design a programmable memory map.



(a)



(b)

Figure 4.4: (a) TLM-2.0 LT model with an interconnect (b) TLM-2.0 LT model with an interconnect and multiple memories

Each address region has an entry in this table which contains the start address, size and the index of the initiator socket to forward the transaction. To decode an address, the interconnect inspects the address attribute in the generic payload and looks it up in the memory map to determine which outgoing initiator socket to forward the transaction to. If the address is found in the table, the interconnect overwrites the address attribute with the decoded local address in the memory and forwards the transaction to the correct target memory. Otherwise, it aborts the simulation with an error message.

According to TLM-2.0 guidelines, an interconnect module is not allowed to change the data length attribute of the generic payload. This means that if the data in a transaction is split into two separate memories, the interconnect must act as the end point for that particular transaction. Here, the interconnect forms two separate transactions to the memories. In that respect, the role of the interconnect is dynamic. It functions as an interconnect component for some transactions and as a target for other transactions.

Our interconnect module can model a wide range of on-chip interconnect topologies and performance metrics. The choice of the interconnect architecture depends on its cost in terms of area, power, bandwidth utilization, Quality-of-Service (QoS), etc. In this work, our interconnect can be considered as a shared bus matrix in contrast to network-on-chip (NoC) or mesh. However, with the interoperability mechanism offered by TLM-2.0, different models of interconnect architecture can be easily instantiated and evaluated.

4.3 TLM-2.0 Approximately-timed (AT) Model

As described in Section 4.2, each transaction in a LT model has two timing points, the start and the end of the transaction. It is possible to increase the number of timing points for each transaction to have a higher degree of timing accuracy. More accurate timing can help to better model basic communication aspects such as throughput, latency, bandwidth utilization and transaction pipelining. However, it is more likely that processes run in lock step with the SystemC scheduler with more timing points. Hence, it is generally expected that AT models simulate significantly slower than the LT counterparts.

The AT coding style uses a non-blocking transport interface which in addition to the timing annotation supports multiple phases within the lifetime of a transaction. The base protocol for AT modeling defines four phases to represent four timing points for each transaction: the

start and end of the request and the start and the end of the response. These four phases of the base protocol can model three timing parameters: (1) the request accept delay, (2) the latency of the target, and (3) the response accept delay [27, 30].

To build an AT model for a DNN, we first refine the communication part of the LT style initiator. The blocking transport interface is replaced with an implementation of the `nb_transport_fw` and `nb_transport_bw` functions. Following TLM-2.0 guidelines with regards to the usage of the generic payload in non-blocking interfaces, we instantiate a memory manager to `acquire` a generic payload transaction from a pool of transaction objects and `release` it to return to the same pool once the transaction is no longer in use. Furthermore, the logic for handling the base protocol call sequence is also added with the help of a payload event queue (PEQ) and its callback method.

Next, the LT interconnect is replaced with an interconnect that supports the handling of non-blocking interfaces and the AT base protocol. The AT interconnect, unlike its LT counterpart, can queue incoming requests if there is already a request in progress and the interconnect has not completed the `END_REQ` phase for that request. AT interconnect can also queue up incoming responses from the memory to forward transactions later on the backward path to initiators. The sequence of phase transitions for each transaction helps to model the contention. The arbitration strategy implemented in our AT interconnect is First Come First Serve (FCFS) policy. Finally, we reuse the logic for address mapping and address translation from the LT interconnect.

As the final refinement step, we replace the LT memory with an AT counterpart. Our AT memory model implements the base protocol with four phases to provide the proper timing granularity for the AT coding style and accurate contention modeling. The bus width, size, request and response delays for the AT memory are all configurable. To provide accurate timing for comparative analysis, we devise an estimation for read and write request and

response delays for each transaction as follows:

$$\begin{aligned} request\ accept\ delay &= memory\ latency \\ response\ accept\ delay &= \frac{generic\ payload\ length}{memory\ bus\ width} \cdot memory\ latency \end{aligned} \tag{4.1}$$

Given TLM-2.0 AT compliance, we connect the `multi_passthrough_initiator_socket` of the AT interconnect to the `target_socket` of the AT memory module.

4.4 TLM-2.0 Loosely-timed Contention-aware (LT-CA) Model

Transaction modeling using loosely-timed (LT) coding style simulates fast because transactions complete in a single blocking transport method call, namely a `b_transport` call. For the same reason, LT models are usually not used for contention analysis which requires a detailed sequence of interactions between the initiator and the target within the life of a transaction. On the contrary, AT coding style uses a non-blocking transport interface which supports multiple phases within the lifetime of a transaction. This enables resource contention modeling to find performance bottlenecks in the design. However, an AT model simulates slower than its LT counterpart because it can contain up to four function calls to complete a transaction.

AT modeling is one of the more complex aspects of TLM-2.0 which makes AT model development a non-trivial task. Therefore, the development of AT models is often postponed to later stages of the design flow, typically only after the LT model is in place. Moreover, when AT model development is disregarded as result of a tight project schedule, RTL simulations are most likely used to find performance bottlenecks. However, chip-level RTL simulations suffer from orders of magnitude slower simulation speed compared to system-level AT models.

A cycle-accurate model is the closest abstraction to the final hardware and can exhibit the most accurate estimation of memory contention in the design. However, as mentioned earlier, simulation performance of cycle-accurate and even AT models are typically an order of magnitude slower than a LT model. Importantly, techniques to tackle memory contention at the lower levels of abstraction usually have a sub-optimal impact on the overall performance. Hence, it is crucial that memory contention becomes visible already at the early stages of the design flow. This enables system designers and chip architects to codesign both hardware and software to optimally mitigate memory bottlenecks.

To have full visibility of memory contention early on with fast simulation, we propose to use the timing annotation in the blocking transport interface to keep track of memory congestion. By storing the memory-busy status in a state variable inside the interconnect, we can schedule transactions to occur at the correct simulation time without the need to store pending transactions in a PEQ.

As listed in Algorithm 2, we store a timestamp marking the end of memory occupation in a state variable (`busy_until`). Since the memory is not busy at the start of the simulation, we initialize `busy_until` to zero. Once a new transaction arrives at the interconnect, we calculate the remaining time left until the memory becomes available again (`busy`). If `busy` is less than zero, this indicates that the transaction has arrived after the point that the memory was busy. Hence, the memory is available and `busy_until` is reset to the current timestamp. Before forwarding the transaction to the memory, the timing annotation of the transaction is updated with the sum of `busy` and the interconnect latency. Once the transaction is forwarded to the memory, the LT memory in turn updates the timing annotation with its read or write latency and returns immediately. Before returning the transaction to the initiator, the `busy_until` variable is then incremented with the recorded time for the memory latency (`memory_delay`).

Note that this minimal change to the LT model of the interconnect is easy to add and does

Algorithm 2: Maintaining busy state in b_transport inside interconnect

```
initialization: busy_until = 0;
busy = busy_until − current timestamp;
if busy < 0 then
    busy_until = current timestamp;
    busy = 0;
end
delay = interconnect delay + busy;
d1 = delay;
socket−>b_transport(transaction, delay);
d2 = delay;
memory delay = d2 − d1;
busy_until += memory delay;
```

not require any knowledge of AT modeling. Furthermore, any LT-memory type with an arbitrary memory delay can be connected to the interconnect. We also devise an estimation for the delay of each generic payload as follows:

$$payload\ delay = \frac{generic\ payload\ length}{memory\ bus\ width} \cdot memory\ latency \tag{4.2}$$

Given an interconnect that stores the busy state of the memory and the memory delay of each payload, the model is contention-aware and reflects accurate simulation time. Most importantly, our loosely-timed contention-aware (LT-CA) modeling uses the blocking transport interface which simulates fast.

4.5 Transaction Level Model Generator

A model generator should be parameterizable, customizable and extensible so that it can be flexibly utilized for wide design space exploration. The automatic generation of the aforementioned TLM models has two benefits: (a) it saves time for model development and manual optimization, and (b) due to the absence of manual coding, it allows for easy verification

and can minimize human errors. Based on the model generator initially developed in [1] and [2], we design a significantly improved generator framework, called `netspec` (Figure 4.2), to automatically generate customized SystemC TLM-1 and TLM-2.0 models with different timing accuracy (untimed, LT, AT, and LT-CA) from an abstract DNN specification.

Figure 4.5 illustrates the internal structure of our automatic generator. First, `netspec` is instrumented by a set of modeling parameters for design space exploration which describe a wide range of modeling features, such as the desired TLM standard, coding style, inter-module communication, and buffer architecture. Second, `netspec` extracts the network architecture and network learned parameters by parsing the DNN textual protocol buffer file (`.prototxt`) and a DNN binary protocol buffer file (`.caffemodel`). Third, `netspec` constructs an internal graph data structure that stores each node's inputs and outputs, the input and output buffer shapes of each node, and the shapes of weights/biases for those nodes with learned parameters. Fourth, based on the TLM parameters and network hyper-parameters, `netspec` constructs a custom generator for each SystemC module. Each module generator captures the attributes for a customized constructor, the specific method for TLM communication, the support for temporal decoupling and the buffer addressing. Finally, `netspec` generates SystemC code for all the modules in the network, as well as the top level network module with all its connections.

`Netspec` is written in Python 3 and uses the Python interface to the Caffe library, `pyCaffe`, in order to read the input files and construct its internal data representation of the DNN. Following good practices of SystemC coding, `netspec` generates separate SystemC header and implementation files to enable a modular file structure and build flow.

`Netspec` can generate both TLM-1 and TLM-2.0 untimed, LT, AT and LT-CA models based on modeling type, coding style and contention configuration. In the case of TLM-2.0, `netspec` automatically generates an address map file based on the buffer architecture and supports memory address generation for multiple buffers for any layer in the network.

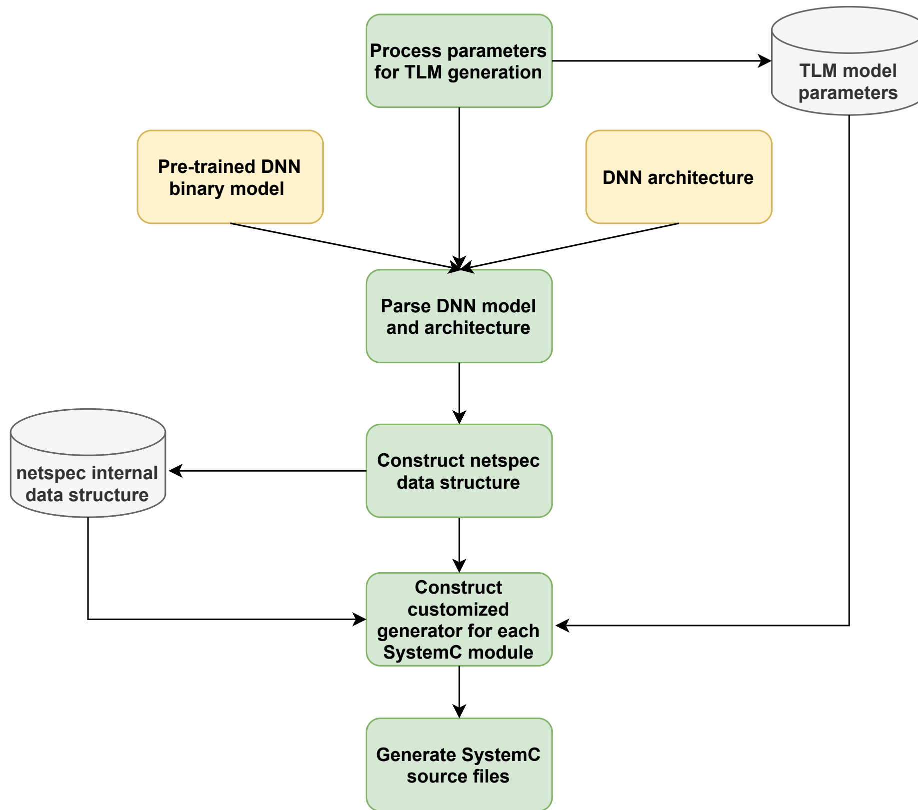


Figure 4.5: Internal structure of the `netspec` TLM generator for transaction-level design space exploration

For DSE, the latency of memory modules, the accept latency of the interconnect and the peak computational performance available to each layer are all configurable. The global interconnect supports multiple memories with arbitrary sizes. Table 4.1 summarizes the features of `netspec` which provides a DNN system synthesis framework that automatically generates SystemC models from an abstract specification.

4.6 Experiments and Results

Using our TLM model generator we have generated a set of TLM-1 and TLM-2.0 models of GoogLeNet. In this section, we describe our experiments, obtained simulation results, and some insights gained from analyzing the models.

Table 4.1: Table of parameterized features for `netspec` model generation and component customization

Feature	Possible values	Description
TLM standard	TLM1/TLM2	Specify TLM standard
Coding style	UT/LT/AT	Specify model's timing points: untimed, loosely-timed and approximately-timed
Channel type	BLK/NBLK/SC	In case of TLM-1, user-defined blocking FIFO, user-defined non-blocking FIFO and SystemC FIFO
Inter-module communication	FF/BP	In case of TLM-2.0, feed forward and back-pressure
Buffer architecture	1..N	In case of TLM-1, number of buffers inside arbitrary channels. In case of TLM-2.0, number of buffers allocated for each module inside memory
Interconnect architecture	1..N	In case of TLM-2.0, number of initiator sockets connected to memory(ies) for multiple memories support
Global memory architecture	(NumxSize) (in MiB)	In case of TLM-2.0, number and size of memories connected to the interconnect
Computational capacity	X (GFLOPS)	In case of TLM-2.0, peak computational capacity available to each layer (default 1 GFLOPS)
Memory latency	X (ps)	In case of TLM-2.0, word latency for read-/write memory accesses (default 1ps)
Contention	True/False	In case of TLM-2.0, disable/enable modeling of interconnect/memory contention in loosely-timed model

4.6.1 Simulation setup

We use SystemC 2.3.1 and OpenCV 3.4.1 built in the default release mode settings for simulation. For benchmarking, we measure the simulator run-time using Linux */usr/bin/time* under CentOS 6.10. To have reproducible experiments, the Linux CPU scaling governor is set to 'performance' mode to run all cores at the maximum frequency and file I/O operations are minimized. An Intel Xeon E5-2680 CPU running at 2.7 GHz with 8 physical cores and 2 threads per core is used as our simulation platform[1]. Lastly, the stimulus module is config-

[1]We have also measured our simulator run-time using another simulation platform with only 4 physical cores and 2 threads per core. The obtained simulation results confirm identical pattern in measured total simulator run-time for the host with fewer number of available cores.

ured to feed 100 images with size of 224x224 pixels to the model which results in reasonable simulator run-times for our experiments.

4.6.2 Memory load estimation

For an estimation of the load on the memory from the DNN, we generate an untimed TLM-2.0 model. The layers' input and output data are stored in a single global memory and each layer uses its own local storage to process its data. We use double-buffering in the memory so that the producer layer can write to the front buffer and the consumer layer can simultaneously read the data from the back buffer, and vice versa. Since the untimed model runs on a delta-cycle basis, we devise a specific buffer architecture for the concat layers at the bottom of the inception modules to compensate for the unbalanced graph topology. The concat layers require 4, 2, 2, and 3 buffers in the tracks to synchronize correctly in the double-buffering scheme. Figure 4.6 shows the total read and write memory accesses to the single memory architecture using the model running the classification of 100 images. As the figure shows, the memory accesses peak at over 90 million bytes per delta-cycle once all layers are active and processing data. This confirms that GoogLeNet is a very memory intensive application.

The TLM-2.0 untimed model also measures the memory usage of the DNN. For example, Table 4.2 lists the memory requirements for each layer in the GoogLeNet. The total required memory for each layer is the combination of its input buffers, output buffer and, in case of learned parameters, its weights and biases. To reduce the memory footprint of the DNN, adjacent modules can share their input/output buffers. Hence, a consumer module points to the output of the corresponding producer module.

As described in Section 3.3.1, to implement a double-buffering scheme, extra filler buffers are required to balance the graph architecture. Therefore, the total memory requirement

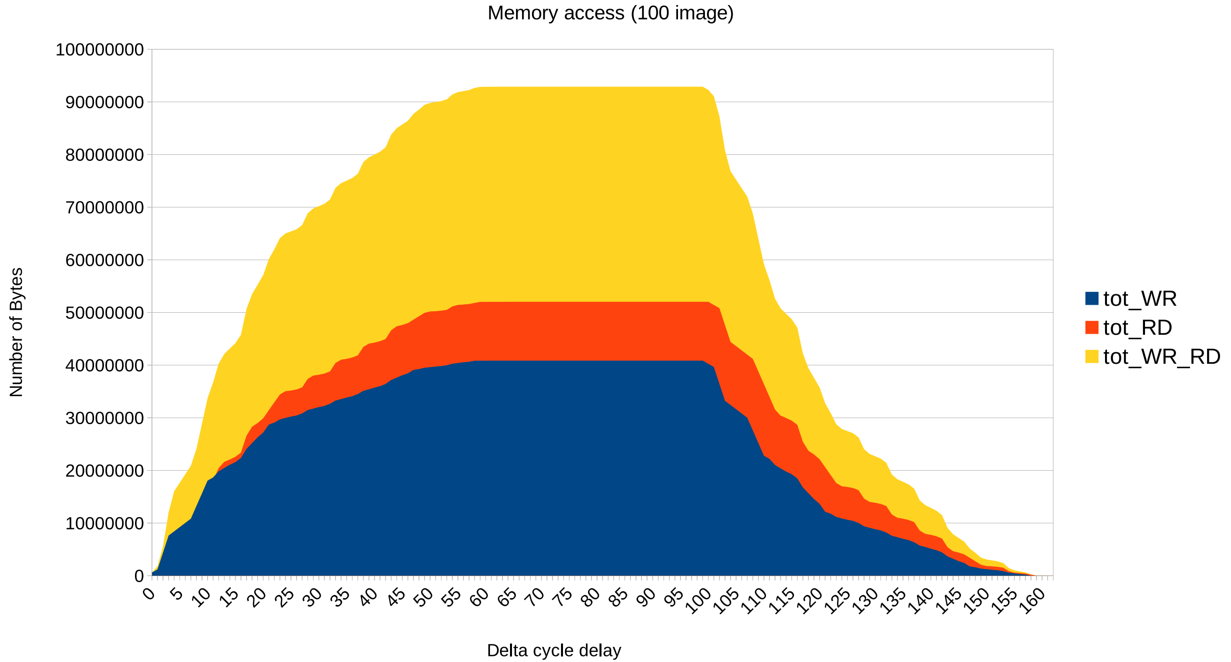


Figure 4.6: Total memory accesses for pass of 100 images in GoogLeNet

of GoogLeNet in a double-buffering mode is double the size of the total output buffers: $40.02 \times 2 = 80.04\ MiB$. It is also worth mentioning that with a smart addressing scheme, the memory footprint can be further reduced. For example, the consumers of the concat layers can simply point to the output buffers of the concat producers.

4.6.3 Comparison of LT, AT, and LT-CA models

Our proposed `netspec` can automatically generate TLM-2.0 models for early software performance analysis and virtual prototyping. The generated LT models carry sufficient timing information to provide a coarse-grain estimation of the application execution time. However, the generic LT does not take into account any bus contention. In contrast, our proposed LT-CA counterpart is developed to analyze the effect of interconnect and memory contention. Additionally, AT models provide even better timing accuracy for memory contention analysis. Therefore, we also generate AT models of our application for comparison.

Table 4.2: GoogLeNet total memory footprint

Layer type	Input [MiB]	Output [MiB]	Weights [MiB]	Bias [MiB]	Total memory [MiB]
Input	0.000	0.574	0.000	0.000	0.574
Convolution	17.78	12.30	22.75	0.027	52.87
ReLU	12.30	12.30	0.000	0.000	24.61
Pooling	11.16	5.411	0.000	0.000	16.57
LRN	3.062	3.062	0.000	0.000	6.125
Concat	4.713	4.713	0.000	0.000	9.426
Dropout	0.003	0.003	0.000	0.000	0.007
InnerProduct	0.003	0.003	3.906	0.003	3.917
Softmax	0.003	0.003	0.000	0.000	0.007
Extra Fillers	0.000	1.638	0.000	0.000	1.638
Total	49.04	40.02	26.66	0.030	115.76

By choosing four different memory latencies[2] (1ns, 10ns, 100ns, 1000ns) and four different computational capacities[3] (1 GFLOPS, 10 GFLOPS, 100 GFLOPS, 1000 GFLOPS), `netspec` automatically generates 16 different LT models. We also instruct `netspec` to generate 16 LT-CA models and 16 AT models across the same parameters. The LT memory bus is configured to have a 64-bit width and the interconnect is configured to avoid extra acceptance latency. Furthermore, we set the burst length to 8 and the size of the generic payload for each transaction is configured to be the burst length multiplied by the memory data width (8*8B = 64B). Since the AT memory can model both request and response accept latencies, we configure the AT memory request accept delay equal to memory latency and the response accept delay identical to LT-CA and LT memory delays. Finally, we set the size of the generic payload for each transaction identical to LT-CA and LT models (64B). Table 4.3 summarizes the total simulated time for all 48 models.

[2]Sweeping memory latency and computational capacity values are based on today's technological candidates to fulfill those requirements. For example, current memory technology to realize aforementioned access delays includes static random-access memory (SRAM), high bandwidth on-chip memory, dynamic random access memory (DRAM), and flash memory.

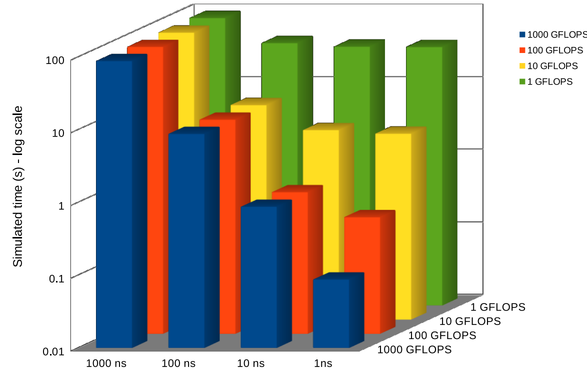
[3]A viable candidate to realize such computational capacities could be a massively parallel processor array. For example, chips such as Epiphany-V [48] (2016) and Manticore [70] (2020) have already demonstrated high performance and energy-efficient many-core architectures. Considering their competitive efficacy in performance and energy metrics such as GFLOPS/mm^2, GFLOPS/Watt and Watt/mm^2, there are good indications that with the increase in transistor density in the future, such computing power may be available.

Table 4.3: Total simulated time of GoogLeNet for different computational capacities and memory latencies (in seconds)

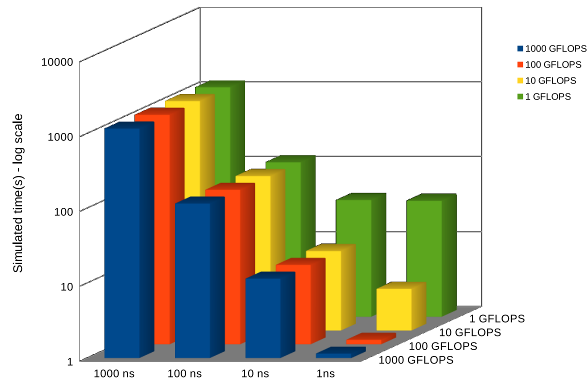
	Loosely-timed				Loosely-timed contention-aware				Approximately-timed			
memory / comp	1ns	10ns	100ns	1000ns	1ns	10ns	100ns	1000ns	1n	10ns	100ns	1000ns
1000 GFLOPS	0.088	0.877	8.763	87.62	1.161	11.61	116.1	1161	1.161	11.61	116.1	1161
100 GFLOPS	0.403	0.888	8.773	87.63	1.164	11.61	116.1	1161	1.164	11.61	116.1	1161
10 GFLOPS	3.603	4.034	8.888	87.73	3.618	11.64	116.1	1161	3.623	11.64	116.1	1161
1 GFLOPS	35.60	36.03	40.34	88.88	35.61	36.18	116.4	1161	35.62	36.23	116.4	1161

As shown in Table 4.3, the total simulated times of LT models (left box) are significantly less than their LT-CA and AT counterparts. In generic LT modeling, transactions that simultaneously access the shared memory, complete their accesses at the same simulated time point. This lack of contention modeling incorrectly makes the total simulated time shorter than with the other modeling styles that reflect contention. As shown in the middle box of Table 4.3, the total simulated times of the LT-CA models show significant increase compared to LT models by taking the effect of contention into account. Finally, the right box shows the total simulated times of AT models that accurately model contention of both memory requests and memory responses. Comparing LT-CA and AT simulated times shows the high accuracy and high fidelity of our proposed LT-CA modeling.

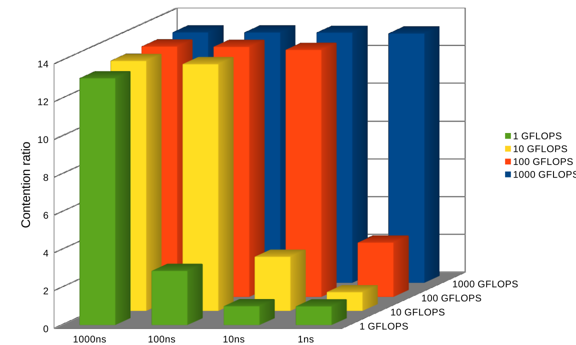
Figures 4.7a and 4.7b visualize the total simulated time of LT and LT-CA models reported in Table 4.3. As expected, both lower computational capacities and higher memory latencies increase simulated time. However, the impact of memory with higher latencies on the performance is more significant. For example, the memory with 1000ns latency performs constantly over all computational capacities, indicating an increase in computational power leaves no significant effect on simulated time. Moreover, the impact of memory with higher latencies on the performance becomes more significant in higher computational powers. For example, having 1000 GFLOPS computational capacity available, 10x increase in the memory latency from 10ns to 100ns leads to 10x decrease in performance. On the contrary, at the same interval in 1 GFLOPS, performance only decreases 3x. This clearly shows the



(a)



(b)



(c)

Figure 4.7: Simulated time of GoogLeNet for a pass of 100 images across 4 computational capacities and 4 memory latencies (a) Loosely-timed (b) Loosely-timed contention-aware (c) Contention ratio

application is heavily memory-bound, rather than compute-bound.

Figure 4.7c illustrates the impact of memory contention by showing the ratios of simulated times for LT-CA models over simulated times for LT models. The negative effect of memory

contention is visible in every computation/memory configuration. However, the contention is more noticeable in higher computational capacities. For example for 1000 GFLOPS, the contention reduces the performance by 13x. Meanwhile, the contention has a lower impact on the lower computational capacities. For 1 GFLOPS, the decreasing memory latency by 10x from 10ns to 1ns (which is quite an expensive design decision), has only little effect on the performance.

In contrast to the simulated execution times, Table 4.4 lists the total simulator run-time for all LT, LT-CA and AT models. As shown, LT models simulate faster than their LT-CA and AT counterparts, because they use only a single function call to complete a transaction. LT-CA models simulate slightly slower than LT models (1.2x) by storing memory congestion status inside the interconnect. AT models have much longer simulator run-times as each transaction can have multiple phases and can use up to four function calls to complete a transaction. In particular, the simulator speed of the LT-CA models is an order of magnitude higher than their AT counterparts. Notably, the LT-CA models show an impressive total speedup of 46x in simulation while providing the same accuracy.

Table 4.4: Total simulator run-time of GoogLeNet for different computational capacities and memory latencies (in seconds) on a 32-core host

	Loosely-timed				Loosely-timed contention-aware				Approximately-timed			
comp \ memory	1ns	10ns	100ns	1000ns	1ns	10ns	100ns	1000ns	1n	10ns	100ns	1000ns
1000 GFLOPS	124.4	121.6	120.4	119.8	141.3	140.0	139.0	137.9	6496	6594	6520	6473
100 GFLOPS	106.9	123.6	123.0	124.0	145.8	145.0	144.9	141.4	6476	6504	6569	6434
10 GFLOPS	105.0	108.5	123.4	131.8	126.5	146.2	142.9	142.7	6310	6669	6544	6529
1 GFLOPS	98.85	104.3	108.9	127.6	124.9	124.6	143.8	141.7	6493	6360	6621	6473

4.6.4 Contention visualization

By having the fast and accurate LT-CA models available, we can further analyze the memory access patterns and the effect of the interconnect/memory contention in the network. We

simulate the application with 1000 GFLOPS computational capacity and 1ns memory latency using the same setup in LT-CA modeling and the buffer size as the generic payload data length. Figure 4.8 shows the transaction-level timing diagram of the first inception module, `inception_3a`, in GoogLeNet. The left diagram shows the timing without contention (LT model) and the right one with contention (LT-CA). The x- and y-axes represent the simulated time and the names of the parallel tracks in the inception module, respectively. The elapsed times for memory write, memory read, computation and contention are colored in blue, light green, dark green and red, respectively.

Since the LT does not model contention, the layers in parallel tracks access memory without blocking each other. However, when a layer accesses the memory in the LT-CA model, the other layers are blocked and wait until access becomes available again (red areas). For instance at the beginning of the contention diagram in Figure 4.8, once the first layer in the 1x1 track issues a read transaction, all layers in the other tracks are blocked until the read transaction completes.

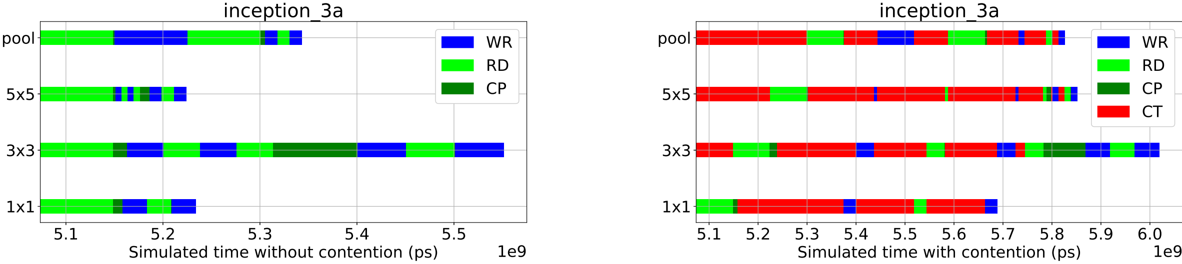


Figure 4.8: Simulated time of the inception_3a module without (left) and with (right) contention (1000 GFLOPS, 1ns memory latency)

It is clear from the visual charts in Figure 4.8 that the LT-CA model accurately reflects the red idle waiting periods for the modules blocked by memory contention. For example, the first layers in all four tracks of `inception_3a` simultaneously initiate read accesses to the memory at 5.07ms with a payload size of 602,112 bytes. The layer in the `1x1` track grants access and blocks the memory for approximately 0.07ms (total delay to access 602,112 bytes

of data). Therefore, the read transaction of the layer in `3x3` track experiences a delay of 0.07ms. Subsequently, layers in `5x5` and `pool` tracks face contention of 0.15ms, and 0.23ms, respectively. Later, as soon as the layer in the `1x1` track completes its computation at 5.15ms, it sends a transaction to write its result to the memory. Since there are already pending transactions, the layer in the `1x1` track must wait until the memory becomes available again at 5.37ms once the layer in the `pool` track completes its read transaction. Hence, the timing annotation of the write transaction for the layer in the `1x1` track is updated with the waiting time for the next memory availability (5.37ms - 5.15ms = 0.22ms). This schedules the write operation to start at 5.37ms precisely after the memory has become available (start of the blue bar in the `1x1` track in Figure 4.8).

Figures 4.9 and 4.10 show the transaction-level timing diagrams for all inception modules in the GoogLeNet for one pass of an image without and with contention, respectively. As seen in Figure 4.10, layers in parallel tracks block each other and contention is high. Furthermore, the track of 3x3 has the highest elapsed time in all inceptions, making it the critical path of the execution. That is valuable feedback to system architects on how to allocate computation resources.

The effect of contention becomes more significant once the DNN pipeline is full with images to process and layers frequently access the shared memory. We simulate the LT-CA model of GoogLeNet by feeding in 100 images to classify. Figure 4.11 shows the timing diagram for the 75th image. As clearly seen by the almost all red coloring in the figure, the layers are mostly blocked due to contention. The effect of back-pressure is also quite visible especially in the first inception module where all its layers are mostly blocked for the subsequent inceptions to process.

The high amount of contention on the shared memory suggests new processor architectures that place private local memories close to computing units. Memory architectures that rely on fine-grained data localities of the given application, reduce the average bandwidth usage

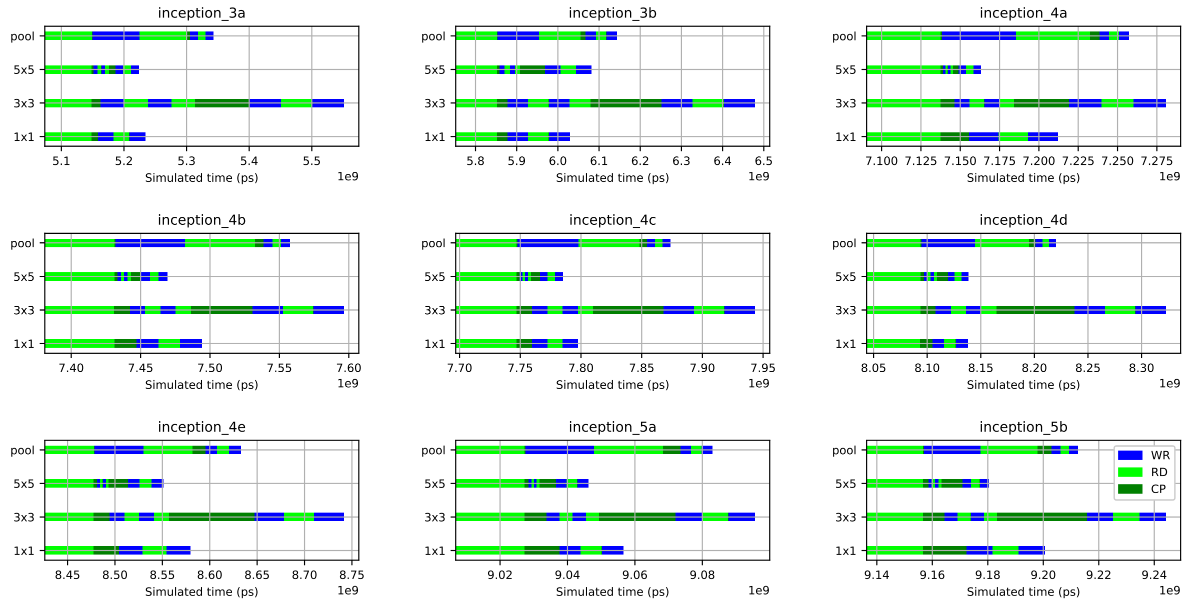


Figure 4.9: Simulated time of all 9 inception modules without contention (1000 GFLOPS, 1ns memory latency)

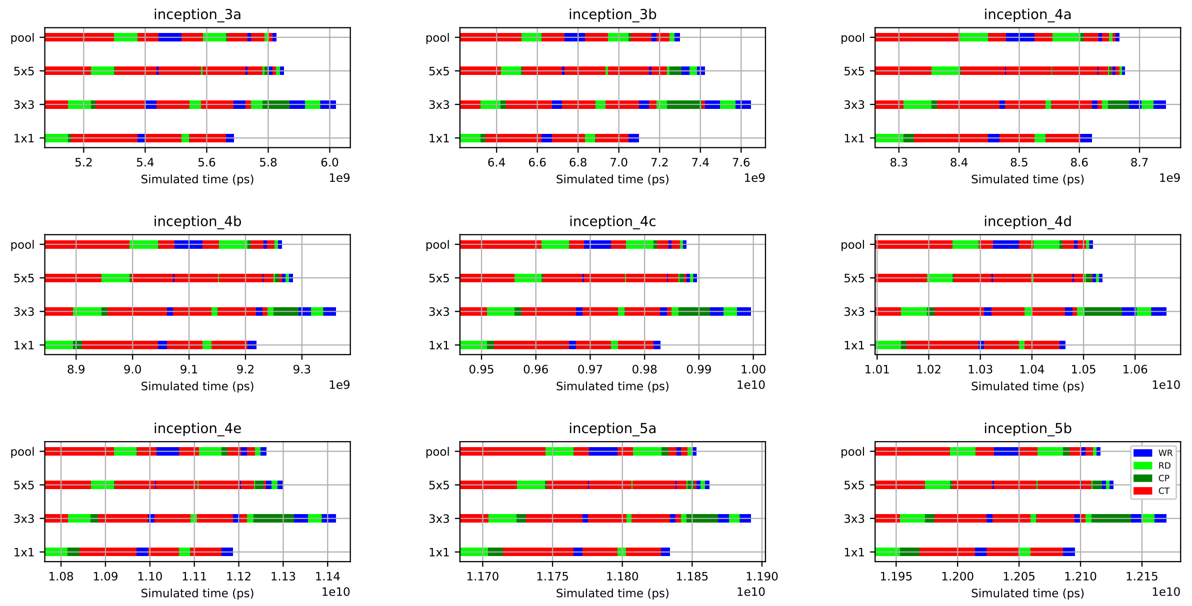


Figure 4.10: Simulated time of all 9 inception modules with contention for image #0 (1000 GFLOPS, 1ns memory latency)

on the global shared memory. Such a memory architecture eliminates the performance overhead of the memory contention and also complex cache coherency. However, new processor architectures with private local memories close to the computing units require us to rethink

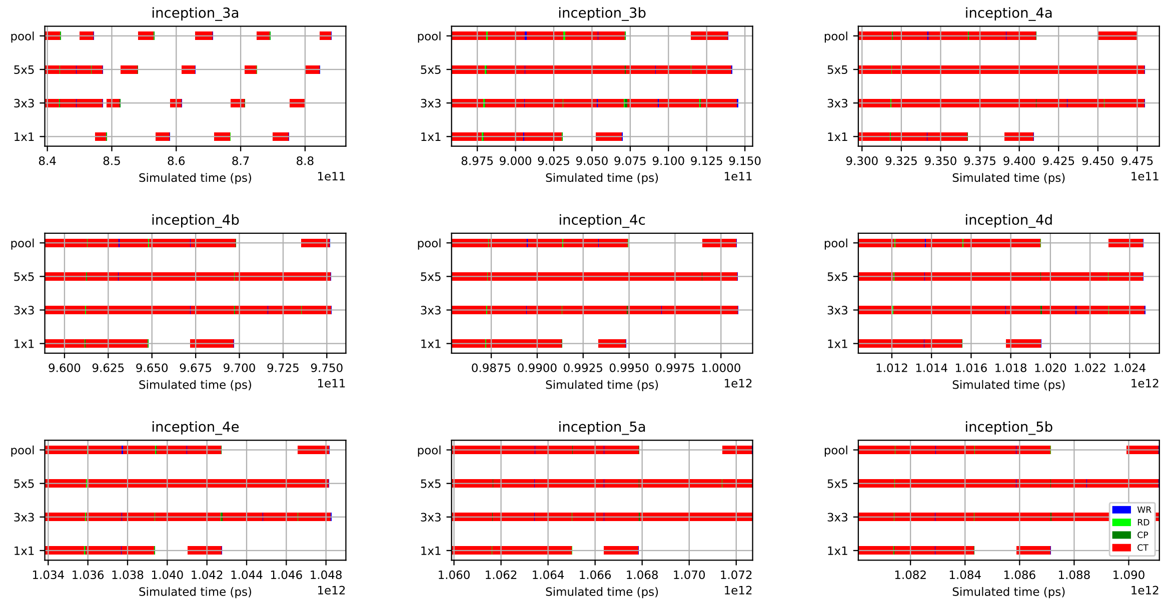


Figure 4.11: Simulated time of all 9 inception modules with contention for image #75 (1000 GFLOPS, 1ns memory latency)

the conventional programming model, compilation flow and run-time system support. For the exact same reason, TLM of architectures with local memories will also require fast yet accurate estimation of the contention and our approach becomes attractive for enabling the efficient codesign of both hardware and software solutions.

4.7 Conclusion

Interconnect and memory contention is a critical aspect in system-level models that requires attention already at the early design stages. In this chapter, we have presented a SystemC TLM framework that automatically generates a configurable set of TLM-1 and TLM-2.0 models from a high-level DNN specification. For efficient design space exploration and performance optimization, our novel loosely-timed contention-aware (LT-CA) modeling breaks the speed/accuracy trade-off and offers high simulation speed with accurate observation of memory contention. We have demonstrated the effectiveness of this approach for represen-

tative large DNN graph structures such as GoogLeNet. Our LT-CA modeling is an order of magnitude faster than its equivalent approximately-timed model (46x) while maintaining the same timing accuracy. Furthermore, we have been able to visualize memory contention to a greater extent using transaction-level timing diagrams, enabling effortless detection of excessive concurrent memory accesses.

Chapter 5

Toward Cycle Accuracy in System-Level Memory Modeling

A cycle-accurate model is the closest abstraction to the final hardware implementation and can exhibit the most accurate estimation of memory contention in the design. However, simulation performance of cycle-accurate models is typically orders of magnitude slower than system-level models. Importantly, techniques to tackle memory contention at the lower levels of abstraction usually have a sub-optimal impact on the overall performance. Therefore, it is crucial that memory contention becomes visible already at the very early stages of the design cycle. This enables system designers and chip architects to truly codesign both hardware and software to optimally mitigate memory contention.

5.1 Introduction

To find the final optimal implementation candidate, TLM models are further refined down to cycle-accurate models of the platform architecture. These cycle-accurate models can

incorporate instruction set simulators (ISS), custom hardware components, and interface components [20]. The ISS represents general-purpose processors that can execute a stream of instructions. Custom hardware components are models of accelerator cores or intellectual property cores (IP cores). Interface components are refinement of interconnection modules which can consist of arbiters, interrupt controllers, bridges and transducers. In later stages of refinement, ISS can be replaced with processor's RTL and in case of a custom hardware and interface components, synthesizable cycle-accurate models can be fed to high-level synthesis (HLS) tools to generate RTL descriptions of the IP core and interface modules.

In this chapter, we primarily aim at the cycle-accuracy of the memory model which has a significant impact on the performance of ISS, IP cores and interface components. First, we further refine the TLM-2.0 approximately-timed model by adding a cycle-accurate model of a shared memory subsystem. The cycle-accurate memory model provides a higher timing accuracy for contention analysis. Hence it gives a more accurate estimation of the performance of the model in terms of simulated time. However, the high temporal accuracy sacrifices the simulation speed. Therefore, we revise our loosely-time contention aware (LT-CA) memory delay modeling to estimate further accuracy comparable to the cycle-accurate TLM model of the memory subsystem.

Observing the high amount of contention on the shared memory suggests new processor architectures. To eliminate the performance overhead of the memory contention and also complex cache coherency, we propose an alternative memory architecture that relies on fine-grained data localities of the given application, hence reducing the average bandwidth usage on the global shared memory.

To further evaluate our system-level modeling framework, we feed the specification of another state-of-the-art DNN, Single Shot MultiBox Detector (SSD) network introduced in 1.4.2, to our automatic SystemC model generator. Using the TLM-2.0 LT-CA of the SSD network, we are able to find memory contention in a fast and accurate fashion.

Our key contributions in this chapter are as follows:

(1) A proposal of an enhanced memory delay model for loosely-timed contention-aware modeling to improve timing accuracy comparable to a cycle-accurate memory model

(2) A local memory architecture as an alternative to conventional approaches (i.e., globally shared memory) to mitigate memory contention in DNNs

(3) Extensive experimental results with cycle-accurate memory models and accuracy analysis

The rest of this chapter is organized as follows: Section 5.2 describes the TLM-2.0 AT model with a cycle-accurate memory subsystem. Section 5.3 presents our enhanced memory delay model to show the accuracy and fidelity of the LT-CA modeling compared to a cycle-accurate memory model. To mitigate memory contention, we introduce a new processor architecture model with local memories in Section 5.4. We present our extensive results and analysis in Section 5.5. Furthermore, we demonstrate the effectiveness of our modeling framework by modeling the SSD network using its abstract DNN specification in Section 5.5.4. Finally, Section 5.6 concludes and summarizes this chapter.

5.2 TLM-2.0 AT Model with Cycle-Accurate Memory

To provide accurate estimation of the performance, we refine the approximately-timed memory model with a cycle-accurate memory subsystem model. Towards this end, we make use of the DRAMSys [60] framework that contains a TLM-based model of an accurately modeled Dynamic Random Access Memory (DRAM) subsystem. Specifically, we rely on DRAMSys4.0 [32] [31], a cycle-accurate DRAM subsystem design exploration framework based on SystemC TLM-2.0 with AT coding style. Since the AT base protocol is not sufficient to model the communication between memory controller and device with full accuracy, DRAMSys4.0

relies on its own custom TLM AT protocol, called DRAM-AT, to model DRAM commands. Moreover, DRAMSys reduces the number of simulated clock cycles by only simulating the clock cycles in which DRAM device state changes occur. This approach results in high simulation speedup in comparison to RTL memory model simulations which require simulating all clock cycles to model DRAM commands.

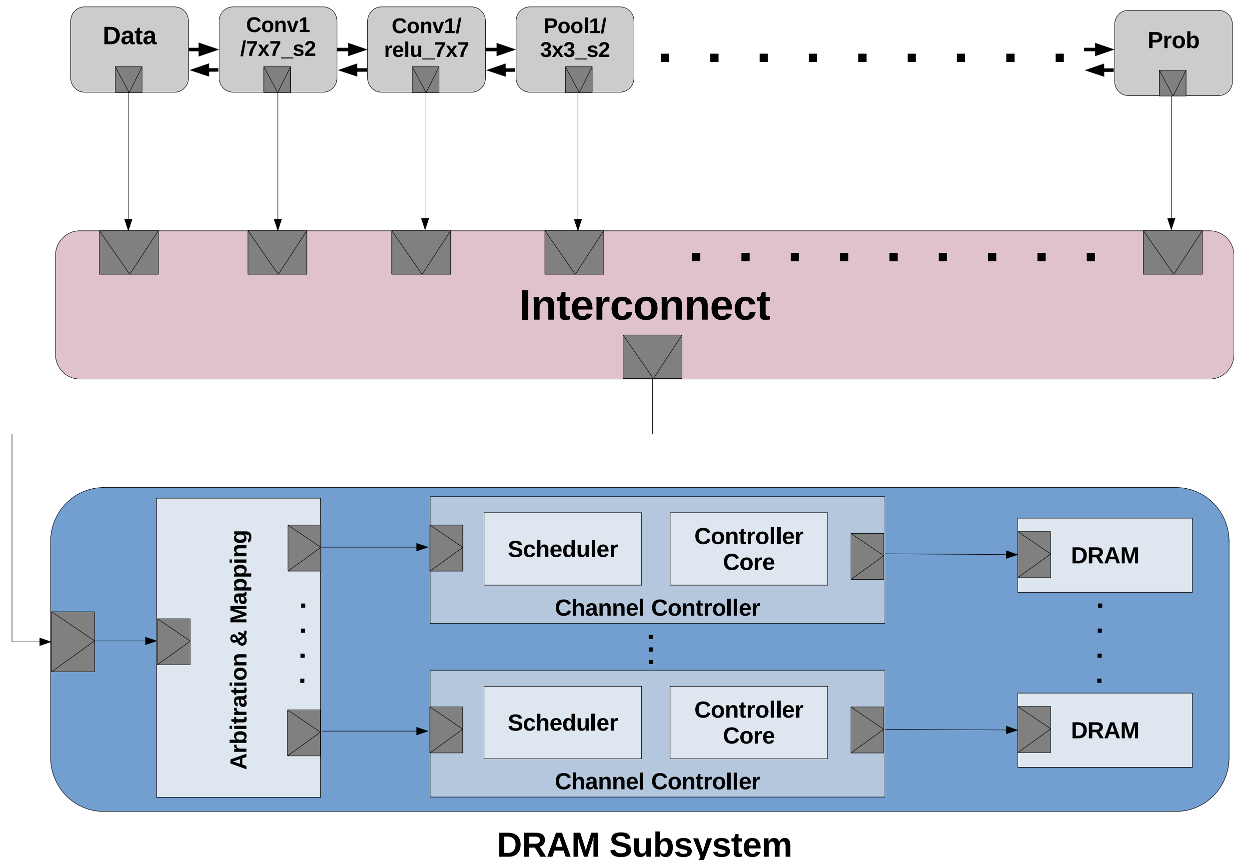


Figure 5.1: TLM-2.0 AT model with an interconnect and DRAM subsystem implemented by DRAMSys4.0 [32] framework

To use DRAMSys memory models, we include the DRAMSys library header in our source code and dynamically instantiate the DRAMSys module inside our AT model. Given TLM-2.0 AT compliance, we connect the `multi_passthrough_initiator_socket` of the AT interconnect to the `multi_passthrough_target_socket` of the DRAMSys module. The AT models of computation and interconnect modules remain unchanged. Figure 5.1 shows the TLM-2.0 AT model diagram of GoogLeNet with DRAMSys memory subsystem module con-

nected to the AT interconnect.

As illustrated in Figure 5.1, DRAMSys module contains detailed models of memory controller, channel controller and DRAM device. The frontend part of DRAMSys arbiters, maps and forwards incoming transactions to different channel schedulers based on specific mapping configuration and priority scheme. Since the single channel in the subsystem is independent, each channel has its own scheduler and controller core. Once the scheduler collects and reorders transactions, it forwards them to the backend with the channel controller that handles the correct communication with the DRAM device.

DRAMSys4.0 framework supports different memory standards, addressing schemes, and scheduling policies. These features allow us to experiment with different memory organizations and memory timings for design space exploration. Furthermore, the DRAMSys4.0 framework is written in SystemC, which makes the integration of a TLM of a given DNN application with DRAM models quite seamless. Finally, co-simulation of full SystemC TLM models also offers the advanced analysis and debugging capabilities of the TLM standard.

The TLM-2.0 AT model with a cycle-accurate memory can accurately reflect the effect of the memory contention. This means the performance of such a model would be the closest estimation to the performance of the final implementation compared to pure AT, LT-CA or LT models. However, the simulation speed of cycle-accurate models would be orders of magnitude slower than models in higher levels of abstraction. In the next section, we revise our delay estimation in LT-CA modeling to provide better accuracy in terms of model performance without majorly sacrificing the simulation speed.

5.3 Enhanced Delay Estimation in TLM-2.0 LT-CA Model

As discussed in Section 4.4, TLM-2.0 LT-CA modeling approach makes contention an inseparable part of modeling even for designs at high levels of abstraction. In Section 4.6.3, we also conducted a systematic and comparative analysis between three modeling approaches i.e. LT, LT-CA and AT with respect to total simulated time. We showed the high accuracy and high fidelity of LT-CA modeling compared to the AT model. In this section, we enhance the LT-CA delay modeling to provide high accuracy and high fidelity of simulated time estimates comparable to cycle-accurate memory.

We are aware that simplified estimation models yield fast simulation but generally result in lower accuracy. However, a high level of accuracy may not always be required as long as the estimated value of the design quality metric allows the system designer to make proper tradeoff decisions [21]. Therefore, we would revise our LT-CA delay estimation model that yields estimates with high *fidelity* [1].

For reference, we repeat the original delay estimation equation used in LT-CA models here (Equation 4.2):

$$payload\ delay = \frac{generic\ payload\ length}{memory\ bus\ width} \cdot memory\ latency \tag{5.1}$$

For brevity, we rewrite the above equation as follows:

$$delay = \frac{gp_{\,len}}{bus\ width} \cdot memory\ latency \tag{5.2}$$

[1]Fidelity of an estimation method is defined as the percentage of correctly predicted comparisons between design implementations [36].

We factor in the memory clock frequency and its data rate in the delay estimation and revise the memory latency:

$$memory\ latency = \frac{memory\ clock\ period}{data\ rate} \tag{5.3}$$

Aside from the memory latency incurred by the data transfer, we also add an average access time overhead factor $T_{overhead_{gp}}$ for each memory byte access in our delay model. Hence, we revise the original delay model in Equation 5.2 as follows:

$$delay = \frac{gp_{\,len}}{bus\ width} \cdot memory\ latency + gp_{\,len} \cdot T_{overhead_{gp}} \tag{5.4}$$

The $T_{overhead_{gp}}$ can be estimated by the difference between the inverse of average memory bandwidth and the maximum memory bandwidth. The overhead factor would be simply the extra time required for each byte accessed in the memory and has the unit of seconds:

$$T_{overhead_{gp}} = \left(\frac{1}{BW_{\,avg}} - \frac{1}{BW_{\,max}}\right) \tag{5.5}$$

The maximum bandwidth can be calculated based on memory clock frequency, its data rate, and memory bus width as follows:

$$BW_{\,max} = \frac{data\ rate}{memory\ clock\ period} \cdot bus\ width \tag{5.6}$$

The average memory bandwidth is a fraction of the maximum memory bandwidth. Therefore, we introduce a memory bandwidth utilization factor β to estimate the average memory

bandwidth,

$$BW_{avg} = \beta \cdot BW_{max} \;\; where \; 0 < \beta <= 1 \tag{5.7}$$

The accurate value of β can be only calculated using cycle-accurate model simulation. However, a nominal value of β can be selected for system-level modeling simulation and be calibrated in later stages of refinements. Moreover, since LT-CA simulation is orders of magnitude faster than AT or cycle-accurate simulation, different intervals of β values can be selected and simulated for experimentation.

Finally, we simplify the delay estimation for each memory transaction as follows:

$$\begin{aligned} delay &= \frac{gp_{len}}{bus\ width} \cdot \frac{memory\ clock\ period}{data\ rate} + gp_{len} \cdot \frac{1}{BW_{max}} \cdot (\frac{1}{\beta} - 1) \\ delay &= \frac{gp_{len}}{BW_{max}} + \frac{gp_{len}}{BW_{max} \cdot \beta} - \frac{gp_{len}}{BW_{max}} \\ delay &= \frac{gp_{len}}{BW_{max} \cdot \beta} \end{aligned} \tag{5.8}$$

This refined memory delay estimation provides an average delay for each memory access based on the length of payload, maximum memory bandwidth, and bandwidth utilization. Given the enhanced delay estimation and contention modeling inside the interconnect, LT-CA provides more accurate total simulated time. This makes contention a visible part of a model in early system development, benefiting system designers and system architects to incorporate memory contention in loosely-timed models with minimum effort, acceptable accuracy and high simulation speed.

5.4 Processing-in-Memory (PIM) with Local Memories

LT-CA modeling enables exploration of alternative memory structures fast yet accurately. To minimize contention and improve the locality of data, one proposal can be an architecture with private local memories and interconnects adjacent to computing units. As illustrated in Figure 5.2, we model each layer with two initiator sockets connected to two separate private local memories which store layers' input and output. The global interconnect is broken into local interconnects which exclusively service two adjacent layers. In other words, the global shared memory is transformed into many private local memories which store the intermediate results.

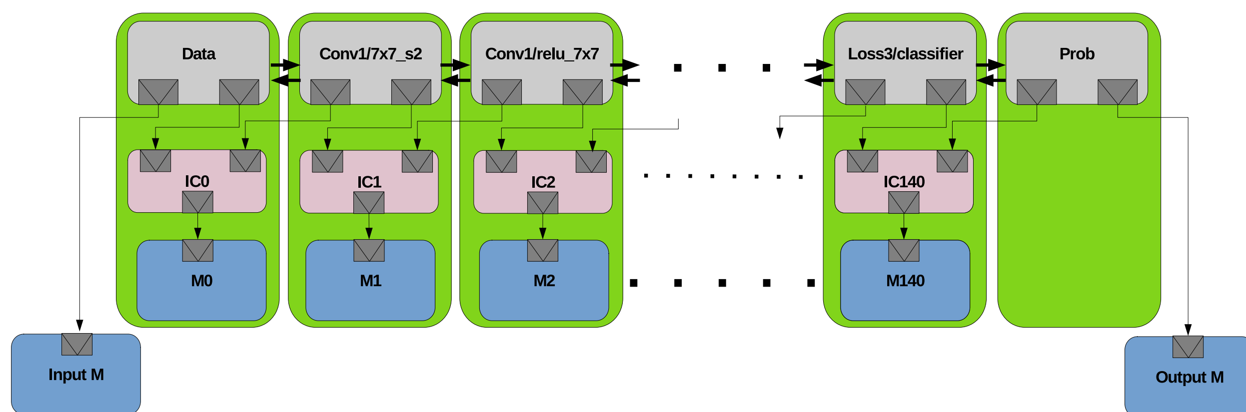


Figure 5.2: TLM-2.0 LT-CA model with local interconnects and private memories

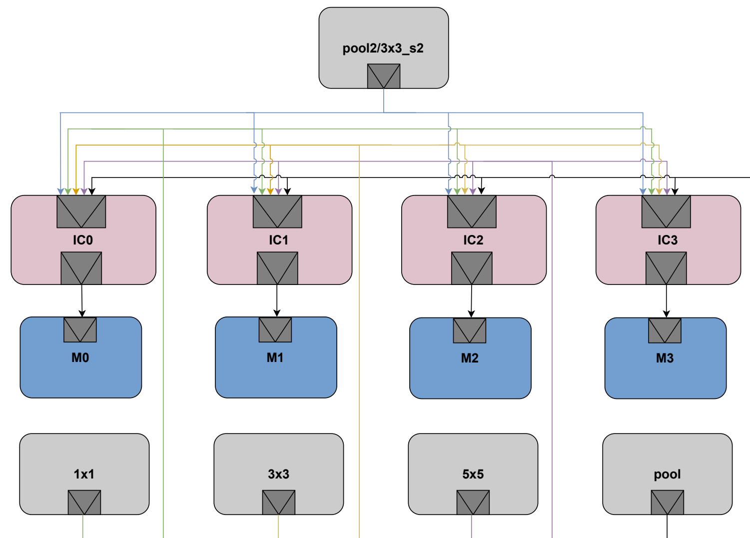
Since data is stored and processed locally in private memories, there is no need for any extra logic to implement cache coherence protocols between multiple computing units. This also eliminates the performance penalty for cache coherency, high congestion in the interconnect and high contention in a global shared memory. Moreover, such a local memory architecture makes programs data-race free as the data is only shared between two adjacent modules and not concurrently accessed by other threads.

As mentioned earlier, the data in the local memory architecture is stored and shared only between adjacent layers. Therefore, layers with more than one input or output require multiple private local memories for each of their input or output layers. Consequently, multi-producer and multi-consumer layers impose an extra design challenge on the local memory architecture. In the case of a multi-consumer layer, we partition the output data into multiple smaller local memories and devise a static scheduling scheme between the consumer layers to avoid contention in simultaneous read accesses. The sizes of these local memories are equal to the output buffer size divided by the number of consumers. This leaves the total memory requirement of the application unchanged.

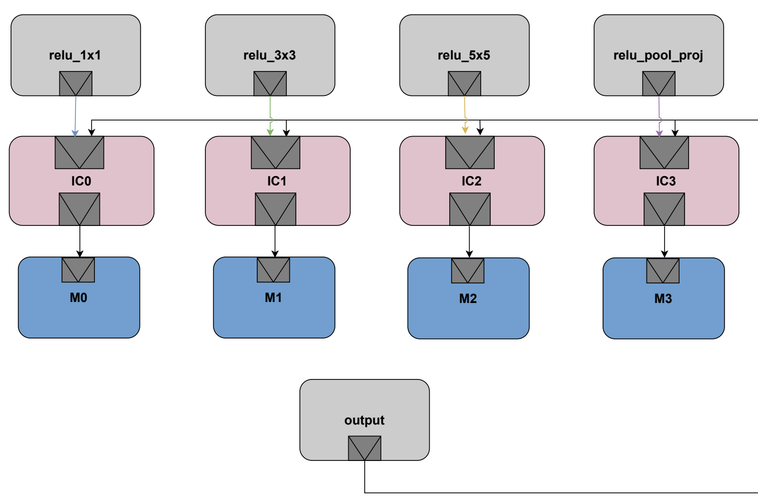
Furthermore, the interconnect `simple_target_socket` which supports only one-to-one connections is replaced with a `multi_passthrough_target_socket` to support the one-to-many connections. As an example illustrated in Figure 5.3a, the input data to the inception module is partitioned into four segments and placed into four local memories. Each consumer layer dispatches four sequential non-overlapping read transactions to read its input. Since the accesses are non-overlapping, the input data for each track is fetched in parallel without any contention.

In the case of a multi-producer layer, each producer owns a private local interconnect and a private memory. The dedicated local memory and local interconnect for each input layer prevents the possible competition between producers to access the single local memory placed between all the producers. In this structure, the producers output their data to their own local memory as soon as they complete their processing without any threat of contention induced by the other producers. For example, such a structure is shown in Figure 5.3b for the inception module output.

We build the LT-CA model of the proposed local memory architecture. Since all data is stored close to the compute cores and the possibility for contention is carefully looked after, memory contention is completely resolved. Therefore, the simulated time of the network



(a)



(b)

Figure 5.3: (a) Inception module input local memory structure (multi-consumer) (b) Inception module output local memory structure (multi-producer)

matches exactly with the LT model with global shared memory that does not model contention. The benefit of the local memory architecture becomes more significant when the network is filled with images (Figure 4.11).

Last but not least, since von Neumann computing is implicitly based on accesses to globally shared memory [46], von Neumann programming styles inherently concern themselves with

the enormous traffic through the memory bottleneck [4]. Therefore, new processor architectures with private local memories close to the computing units require us to rethink the conventional programming model, compilation flow and run-time system support. For the exact same reason, TLM of the local memories architecture with fast yet accurate estimation of the contention is an attractive approach that enables codesign of both hardware and software solutions.

5.5 Experiments and Results

In this section, we describe our extensive cycle-accurate simulation results using DRAMSys4.0. To show the accuracy of our enhanced LT-CA delay modeling, we compare simulation results of DRAMSys with both LT and enhanced LT-CA models. Finally, we show how the LT-CA model of SSD can be used also to effectively visualize memory contention.

5.5.1 Simulation setup

We use SystemC 2.3.4 with the DRAMSys4.0 framework for a cycle-accurate memory modeling simulation. OpenCV 3.4.1 is built in the default release mode setting. We simulate the GoogLeNet AT model with two different memory standards, DDR3 and DDR4. Table 5.1 shows the two DRAM memory specifications used for simulation. To benchmark simulation execution performance, we measure simulator run-time using Linux */usr/bin/time* under CentOS 7. An Intel Xeon E3-1240 CPU running at 3.4 GHz with 4 physical cores and 2 threads per core is used as our simulation platform.

To generate the DNN model with cycle-accurate memory, we extend `netspec` to support model generation with DRAMSys. Furthermore, we also add support for configurable bandwidth utilization factors required for enhanced LT-CA modeling.

Table 5.1: DRAM memory specifications

Parameter	DDR3-1600	DDR4-1866
Address mapping	8x1Gbx8, 1KB page (row, bank, column)	8x4Gbx8, 1KB page (row, bank, column)
Scheduling policy	FR-FCFS	FR-FCFS
Specification	MICRON DDR3-1600	JEDEC DDR4-1866
#Ranks	1	1
#Bank Groups	1	4
#Banks	8	16
#Rows	16384	32768
#Columns	1024	1024
#Devices on DIMM	8	8
#Channels	1	1
Chip data bus width	8	8
Burst length	8	8
Data rate	2	2
Clock frequency	800 MHz	933 MHz

5.5.2 DRAMSys

We build and simulate successfully the cycle-accurate model of the GoogLeNet with DRAMSys4.0 using `netspec`. In order to show the memory access rates with different computational capacities, we run multiple simulations and list the reported bandwidth figures from the DRAMSys framework. Table 5.2 shows the average and maximum bandwidth figures for DDR3-1600 running at 800 MHz and DDR4-1866 running at 933 MHz with a burst length of 8 and a total bus width of 8B. As shown in the first two rows, the average bandwidths do not improve by increasing the computational capacity. This clearly shows the application is heavily memory-bound, rather than compute-bound.

Figure 5.4 shows the strip plot of memory latencies for the AT model reported by DRAMSys DDR3 for all 145 million transactions. Due to the complex and pipelined access behavior of DRAM, it is not possible to predict an exact distribution of the delays. This phenomenon is better highlighted by presenting memory latency histograms in different delay intervals as

Table 5.2: GoogLeNet memory bandwidth figures on DDR3-1600 and DDR4-1866 (burst length 8, memory bus width 8B)

BW (Gb/s) / computation	DDR3-1600		DDR4-1866	
	Avg BW	Max BW	Avg BW	Max BW
1000 GFLOPS	65.72	102.4	94.02	119.4
100 GFLOPS	64.91	102.4	93.24	119.4
10 GFLOPS	20.29	102.4	20.22	119.4
1 GFLOPS	2.08	102.4	2.08	119.4

depicted in Figure 5.5.

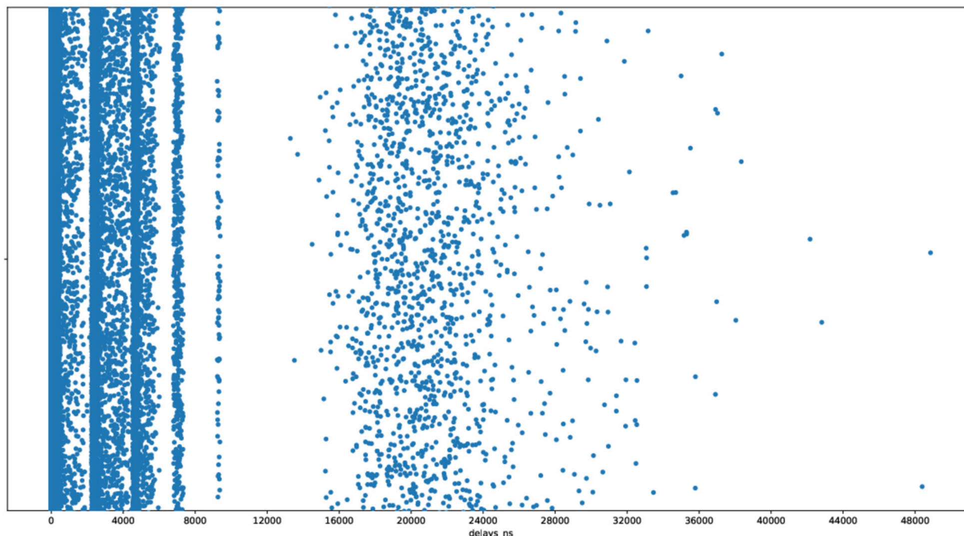


Figure 5.4: Strip plot of memory transaction delays for GoogLeNet AT model with DRAMSys DDR3-1600 (1000 GFLOPS)

Given the underlying distribution of delay is unknown but has finite mean and variance, the sampling distribution of the average delay is approximately Gaussian by the central limit theorem [40]. This allows us to compute the probability involving the average delay even though we do not know the exact distribution of the delay. We will highlight this quality by comparing the delay distributions generated by the cycle-accurate DRAMSys model and the LT-CA model that estimates the average delay in the next section.

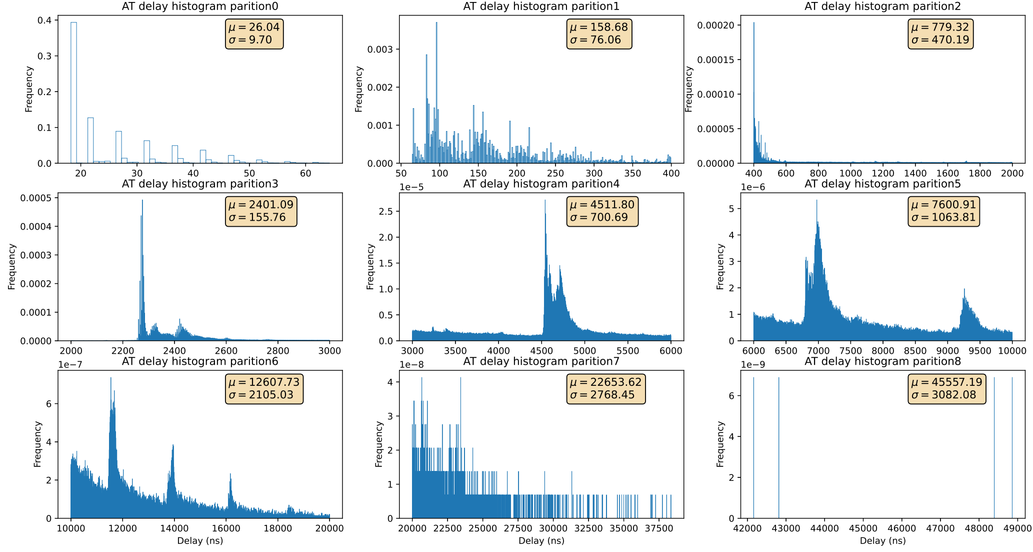


Figure 5.5: Histograms of memory transaction delays for GoogLeNet AT model with DRAMSys DDR3-1600 in partitions (1000 GFLOPS)

5.5.3 LT, LT-CA and DDR

To compare the accuracy of LT-CA modeling between the contention unaware LT model and the cycle-accurate DRAM model, we instrument `netspec` to generate three sets of GoogLeNet model as follows:

1. AT model of GoogLeNet with the cycle-accurate DRAMSys DDR3 model with specification outlined in Table 5.1. To provide accurate timing, the size of generic payload for each transaction is configured to be the burst length multiplied by the memory data width (8*8B = 64B)

2. LT-CA model of GoogLeNet with the LT interconnect with contention modeling and the LT memory with the enhanced delay estimation matching DDR3 memory specification, namely memory latency of 0.625ns (1/(2*800MHz)), a nominal bandwidth utilization factor of 66%, and generic payload size of 64B

3. LT model of GoogLeNet with LT interconnect and LT memory with the enhanced delay estimation matching DDR3 memory specification, namely memory latency of 0.625ns (1/(2*800MHz)), a nominal bandwidth utilization factor of 66%, and generic payload size of 64B

Table 5.3 shows total simulated times of GoogLeNet for LT, LT-CA and DDR3-1600 models. As shown in Table 5.3, the performance of the LT-CA model with enhanced delay estimation provides results comparable to cycle-accurate DDR3 model. However, the LT model without any contention modeling estimates the performance of the design with a high error margin e.g. over 12x less in 1000 GFLOPS.

Table 5.3: Total simulated time of GoogLeNet for LT, LT-CA, and DDR3-1600 models (in milliseconds)

computation \ model	LT	LT-CA	DDR3
1000 GFLOPS	84.067	1097.2	1130.7
100 GFLOPS	404.77	1100.3	1144.8
10 GFLOPS	3600.8	3614.8	3662.7
1 GFLOPS	35601	35612	35660

We repeat the same experiment with the DDR4 memory model. We again instrument `netspec` to generate three sets of GoogLeNet model: AT model with cycle-accurate DRAMSys DDR4-1866 specified in Table 5.1 with generic payload size of 64B (*burst length* $\times$ *memory data width*), LT-CA and LT models matching DDR4 memory specification, namely memory latency of 0.536ns (1/(2*933MHz)), a nominal bandwidth utilization factor of 66%, and generic payload size of 64B. Table 5.4 shows total simulated times of GoogLeNet for the above models. It is evident that LT-CA models estimate total simulated times with acceptable accuracy and fidelity compared to LT models. As expected, LT-CA models are not able to estimate the accurate total simulated times compared to the cycle-accurate DRAM model. However, LT-CA models still provide better estimations compared to contention

unaware LT models used broadly in early system design and modeling.

Table 5.4: Total simulated time of GoogLeNet for LT, LT-CA, and DDR4-1866 models (in milliseconds)

computation \ model	LT	LT-CA	DDR4
1000 GFLOPS	74.214	938.15	790.28
100 GFLOPS	394.22	941.49	796.90
10 GFLOPS	3594.2	3632.3	3675.2
1 GFLOPS	35942	35604	35672

We also measure the simulator run-time of GoogLeNet for each of the modeling approaches to evaluate their simulation execution performances. Table 5.5 summarizes simulator run-time for DRAMsys DDR models together with their corresponding LT and LT-CA models. LT contention unaware models have the fastest simulation execution time. LT-CA models simulate slightly slower due to contention modeling overhead (1.2x). As expected, DDR models have the slowest simulation execution time compared to LT (90x-100x) and LT-CA models (75x-85x). Overall, LT-CA modeling shows a significant simulation speedup with an impressive timing accuracy.

Table 5.5: Total simulator run-time of GoogLeNet for LT, LT-CA and DDR models (in seconds) on a 8-core host

	DDR3-1600			DDR4-1866		
computation \ model	LT	LT-CA	DDR3	LT	LT-CA	DDR4
1000 GFLOPS	72.38	84.36	7228	79.27	85.27	8310
100 GFLOPS	65.98	85.62	7168	66.02	84.98	8618
10 GFLOPS	64.33	76.56	5810	64.29	76.79	6495
1 GFLOPS	64.90	75.75	5724	64.72	78.15	6194

To signify the effect of memory bandwidth utilization factor, we instrument `netspec` to generate GoogLeNet LT and LT-CA models by only sweeping β values for different computational capacities. Table 5.6 shows total simulated times of GoogLeNet LT and LT-CA

models for β values of 0.1 to 1 with an increasing step of 0.1. As seen in column 1 and 2, LT-CA models estimate the total simulated times much higher than the corresponding LT models. This stems from the contention modeling in LT-CA models and preventing concurrent memory accesses to finish early. Furthermore, comparing the simulated times for β values of 0.6 and 0.7 for LT and LT-CA models, it confirms LT-CA estimations are closer to the corresponding cycle-accurate DDR3 models.

Table 5.6: Total simulated time of GoogLeNet for different computational capacities and bandwidth utilization factors β sweeping (0.1, 1) with a step of 0.1 (in milliseconds) for memory running on 800MHz, burst length of 8 and data width of 8B

(a) LT

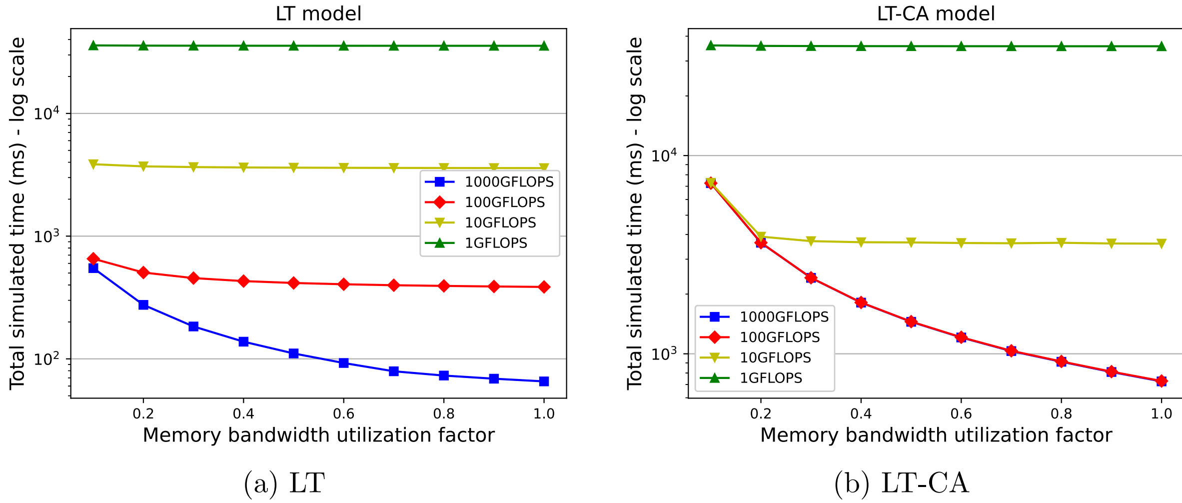
β	1000G	100G	10G	1G
0.1	548.7	654.5	3854.6	35855.4
0.2	275.4	505.3	3705.3	35706.1
0.3	183.6	455.1	3655.2	35655.9
0.4	138.0	430.2	3630.3	35631.1
0.5	110.7	415.3	3615.4	35616.1
0.6	92.5	405.4	3605.4	35606.2
0.7	79.2	398.1	3598.2	35598.9
0.8	73.1	393.1	3593.2	35593.9
0.9	68.9	388.9	3589.0	35589.7
1.0	65.5	385.5	3585.5	35586.3

(b) LT-CA

β	1000G	100G	10G	1G
0.1	7255.3	7255.9	7290.2	35950.9
0.2	3632.9	3634.3	3899.4	35746.8
0.3	2416.1	2418.3	3701.6	35681.4
0.4	1812.4	1814.9	3654.2	35650.4
0.5	1450.2	1453.0	3645.9	35631.6
0.6	1208.7	1211.7	3621.7	35618.8
0.7	1032.2	1035.5	3611.2	35609.7
0.8	911.5	914.8	3630.2	35603.5
0.9	809.3	812.7	3601.9	35598.1
1.0	725.7	729.1	3595.1	35593.8

Figure 5.6 illustrates the total simulated time of GoogLeNet LT and LT-CA models for different β values extracted from Table 5.6 in logarithmic scale. As evident in both diagrams, the change of memory bandwidth utilization factor has no impact on low computational capacities. However, the performance of models is highly dependent on memory utilization factors in high computational capacities. Furthermore, the rate of change in LT-CA models is greater than LT models in higher computational capacities. This indicates that the effect of memory contention is more visible in LT-CA models that consider contention.

Figure 5.7 shows the contention ratio of the TLM model of GoogLeNet for different computational capacities. The contention ratio for each β is calculated based on the simulated



(a) LT

(b) LT-CA

Figure 5.6: Simulated time of GoogLeNet for a pass of 100 images across 4 computational capacities and bandwidth utilization factors β sweeping (0.1, 1) with a step of 0.1 (a) LT (b) LT-CA (in milliseconds) for memory running on 800MHz, burst length of 8 and data width of 8B

time of the LT-CA model that allows contention modeling over the simulated time of the LT model that ignores contention. As expected, there is a negligible level of contention in the lower computational capacities. However, the effect of contention becomes significant in higher computational capacities. For example, the effect of contention on model performance is remarkable in 1000 GFLOPS. For example, the 10x increase in memory bandwidth utilization factor has little to no effect on the contention. In contrast, the effect of contention is flat in lower computational capacities. In specific, the improvement of the bandwidth utilization factor has no impact on the contention in 1 GFLOPS or 10 GFLOPS. These useful insights provide directions to system designers and architects on the allocations of memory and computational resources for the next level of refinements.

Figure 5.8 illustrates the total simulated times for LT and LT-CA models of GoogLeNet for a pass of 100 images across a range of bandwidth utilization factors in each computational capacity extracted from Table 5.6 in log scale. As illustrated in Figures 5.8a and 5.8b, the difference for reported simulated times between LT and LT-CA is significant for high computational capacities. This clearly indicates that contention modeling has a considerable

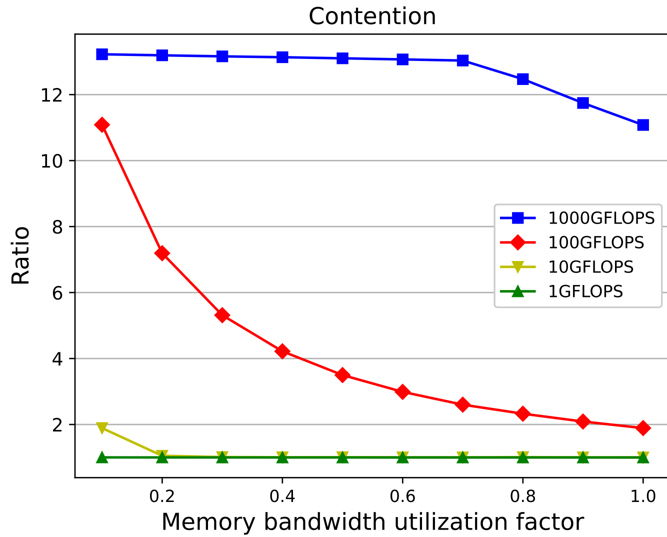
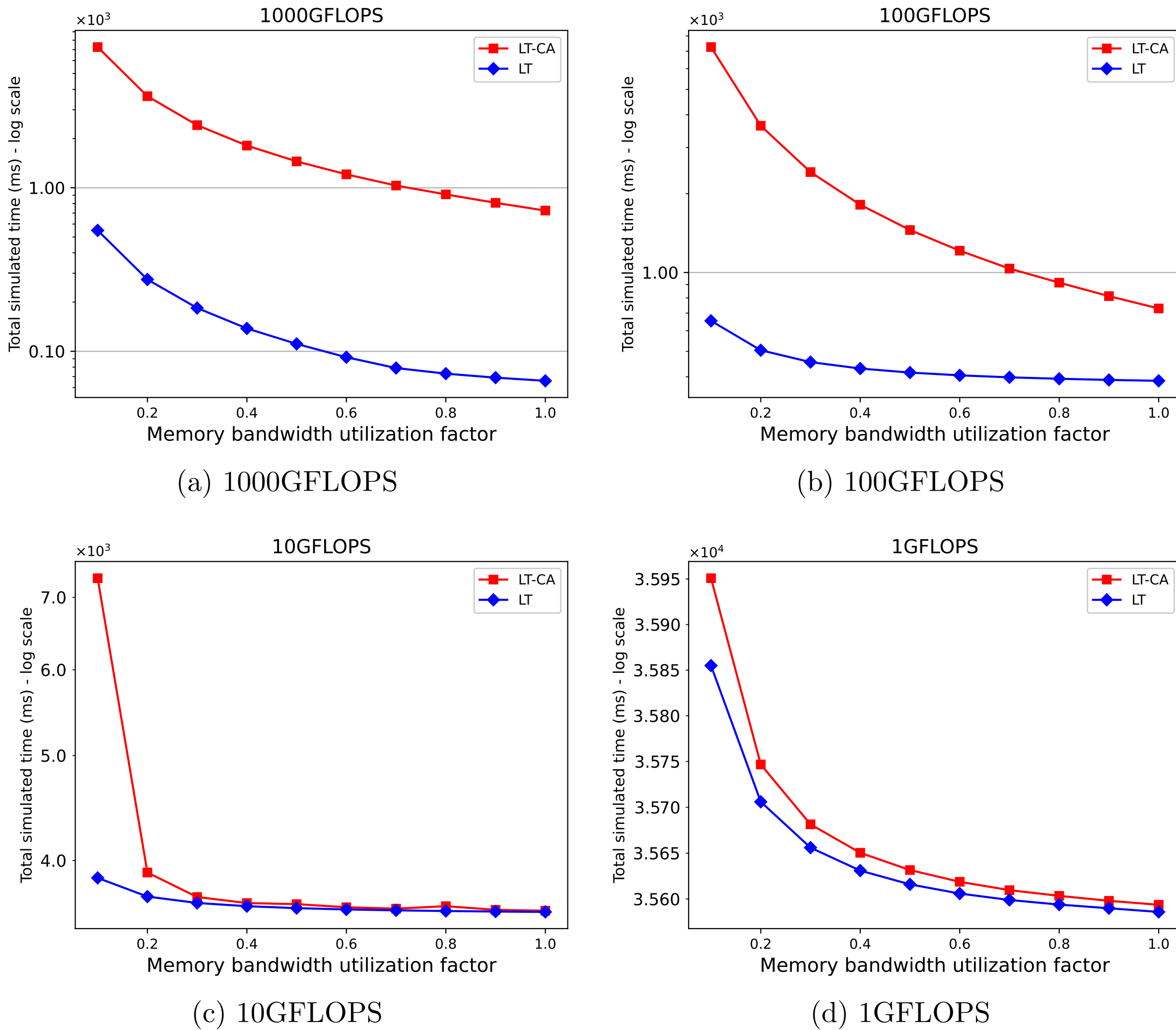


Figure 5.7: Contention ratio of GoogLeNet for a pass of 100 images with memory bandwidth utilization factor of 10%-100% across 4 computational capacities

impact on the quality of models used for early design space exploration.

Finally, we also compare the latency histograms of both AT DDR and LT-CA models at 1000 GFLOPS to further demonstrate our accurate estimation of total simulated time. Figure 5.9a (left) shows the latency histogram for the AT model reported by DRAMSys DDR3 for all 145 million transactions. Due to the complex and pipelined access behavior of DRAM, it is not possible to predict an exact distribution of the delays. Figure 5.9a (center) shows the latency histogram measured by the LT-CA model. Since the LT-CA delay histogram shows the average delay distribution, its probability density is approximately a normal distribution even if the original delays themselves are not normally distributed according to the central limit theorem as described in 5.5.2. The accuracy of the average delay estimation is evident from the close mean delay values in the AT and LT-CA models (107.99ns vs. 118.04ns). Figure 5.9a (right) compares the kernel density estimation (KDE) of the LT-CA delays and a normal distribution with the mean and variance of the LT-CA delays. It is clear that the KDE of the LT-CA and the reference normal distribution have similar shapes. The exact same patterns can also be seen on DDR4-1866 delay histograms (5.9b).



(a) 1000GFLOPS (b) 100GFLOPS

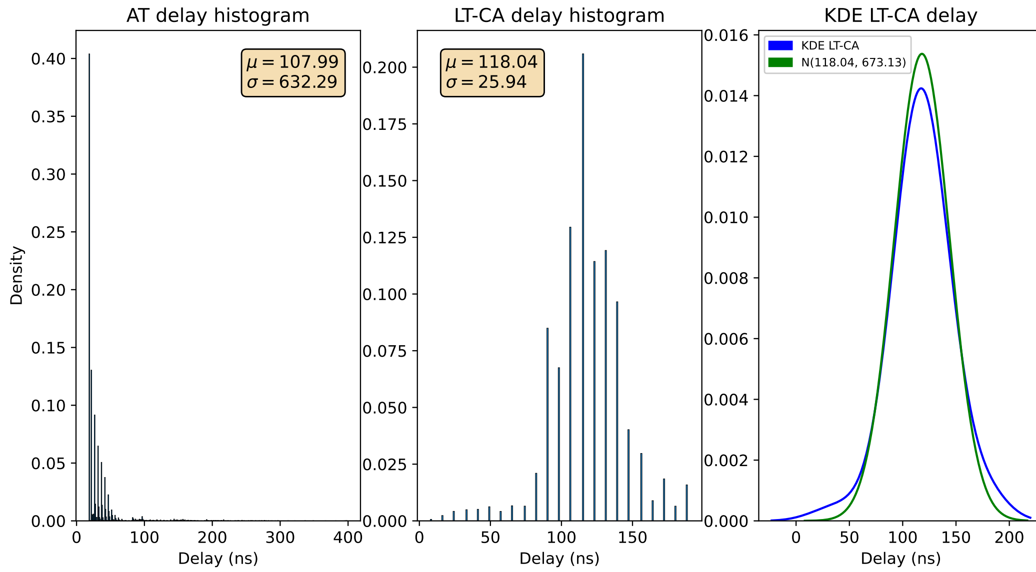
(c) 10GFLOPS (d) 1GFLOPS

Figure 5.8: Simulated time of LT and LT-CA models of GoogLeNet for a pass of 100 images across memory bandwidth utilization factor and categorized for each computational capacity (a) 1000GFLOPS (b) 100GFLOPS (c) 10GFLOPS (d) 1GFLOPS (in milliseconds)

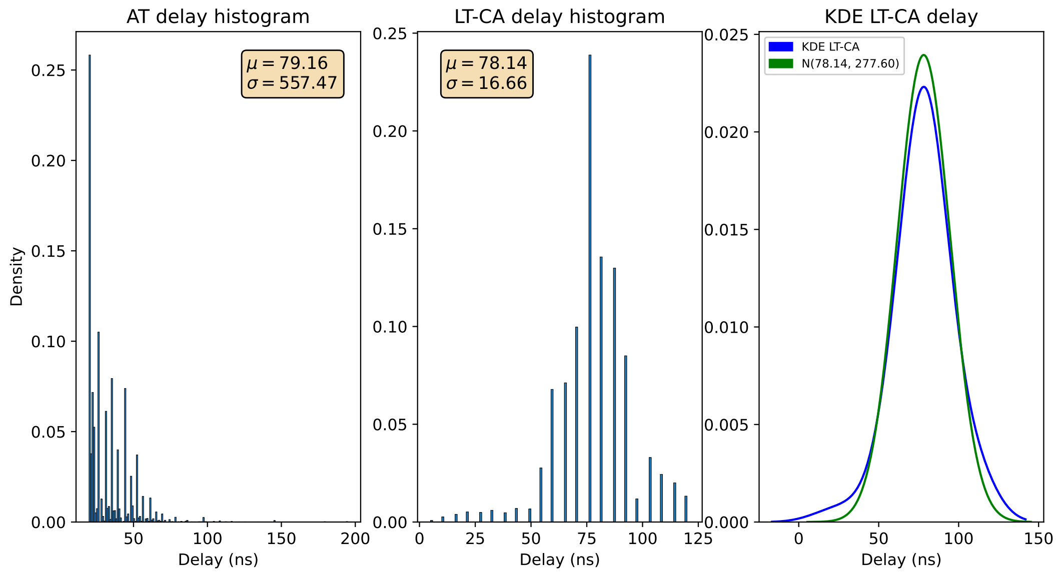
5.5.4 Single Shot MultiBox Detector (SSD)

We have primarily focused on GoogLeNet for modeling and simulation until now. In this section, we demonstrate the generality and effectiveness of our modeling framework on a second application, Single Shot Detector (SSD) network introduced in Section 1.4.2.

Given our automatic model-based design flow, the SSD TLM models can be quickly generated by simply feeding the abstract SSD specification to `netspec`. Then we simulate the



(a)



(b)

Figure 5.9: From left: AT delay histogram, LT-CA delay histogram and kernel density estimation of LT-CA delays versus a normal distribution N(LT-CA mean, LT-CA variance) for (a) DDR3-1600 and (b) DDR4-1866

LT-CA model with similar simulation configurations as used for the GoogLeNet, namely 1000 GFLOPS computational capacity, memory latency of 0.625ns (1/(2*800MHz)), and a

nominal bandwidth utilization factor of 66%. For contention visualization, we set the generic payload data length equal to the buffer size.

The LT-CA model of the SSD can easily demonstrate the memory bottlenecks in the network. For example, Figure 5.10 shows the transaction-level timing diagram for the `fc7` branch of the SSD. The `fc7` branch is composed of four parallel tracks, namely `conv6`, `mbox_loc`, `mbox_conf` and `mbox_priorbox`. As shown in Figure 5.10 (left), the four first layers start a read transaction at 84.7ms with a payload size of 1,478,656 bytes. In the LT model, all four layers are granted access to the memory and experience a memory delay of 0.189ms (total delay of 1,478,656 bytes data). As illustrated in Figure 5.10 (right), the first layer in `mbox_loc` experiences a contention delay of 0.189ms. Subsequently, the first layers in `mbox_conf` and `mbox_priorbox` tracks experience a delay of 0.378ms and 0.576ms respectively. It is evident from the visual representations in Figure 5.10 that the LT-CA model accurately reflects the red idle waiting periods for the modules blocked by memory contention.

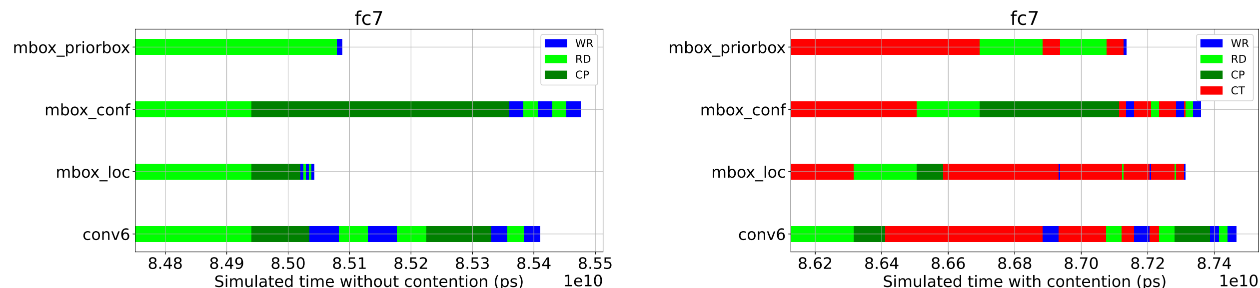


Figure 5.10: Simulated time of fc7 branch without (left) and with (right) contention (1000 GFLOPS, 0.625ns memory latency)

5.6 Conclusion

Cycle-accurate models provide higher timing accuracy at the expense of simulation speed. In this chapter, we presented a TLM-2.0 approximately-timed model with a cycle-accurate shared memory subsystem for accurate memory contention analysis. To have fast yet ac-

curate estimation of contention with high fidelity, we proposed an enhanced memory delay estimation for our loosely-timed contention-aware modeling approach. We also proposed an alternative local memory architecture to globally shared memory in order to eliminate the performance overhead of memory contention. Finally, we performed a systematic analysis of loosely-timed, loosely-timed contention-aware, and approximately-timed with cycle-accurate memory models using GoogLeNet, and demonstrated the effectiveness of our modeling framework for memory contention analysis on the SSD network.

Chapter 6

Conclusion

In this chapter, we briefly summarize the contributions of this dissertation. Additionally, we discuss potential research topics for future work.

6.1 Contributions

In this dissertation, we made a number of key contributions:

1. Design of exploratory modeling framework for hardware-software codesign based on IEEE SystemC TLM for DNN applications with focus on improving parallelism and mitigating memory contention [1, 2, 3, 45]

2. Exploration of parallelism opportunities in TLM models and improvement of parallelism by proposing less restrictive communication mechanisms and transaction types for enhanced parallelism with out-of-order parallel simulation of TLM-1 and TLM-2.0 models [2, 3]

3. Design of a loosely-timed modeling approach for early, fast and accurate memory contention modeling [45]

4. Enhancement of memory delay estimates for loosely-timed modeling to improve timing accuracy comparable to cycle-accurate memory model

6.1.1 DNN System Level Modeling with Exposed Parallelism

In Chapter 2, we proposed an untimed system-level model of a state-of-art deep neural network, GoogLeNet, based on IEEE SystemC. To expose the system-level parallelism, we separated the communication from computation in the design. We successfully validated the functionality of the model using both sequential and parallel SystemC simulators. We presented extensive experimental results to demonstrate the effect of multi-threaded and thread-level parallelism on the simulation speedup using sequential Accellera SystemC 2.3.1 and parallel RISC V0.5.1. simulators.

6.1.2 Improvement of System Level Parallelism

In Chapter 3, we proposed a set of standard-compliant modeling techniques to increase parallelism in IEEE SystemC TLM-1 and TLM-2.0 models. In particular, we demonstrated the impact of varying synchronization mechanisms on the exposed parallelism using six modeling styles of GoogLeNet. Later, we quantified the available parallelism in these six improved SystemC models by measuring the performance of out-of-order PDES in RISC. As a result, we suggested that the design with the highest amount of parallelism exposed is suited best for further refinement in the system design flow.

6.1.3 Fast Loosely-Timed System Models with Accurate Memory Contention

In Chapter 4, we proposed a TLM-2.0 loosely-timed contention-aware modeling approach which offers accurate observation of memory contention with high simulation speed. Fast yet accurate contention modeling enables efficient early design space exploration which both reduce the design time and cost of embedded systems. We also presented our TLM framework based on the IEEE SystemC methodology to explore both parallelism and memory contention early in the design cycle. Our automatic model generation dramatically reduces the burden of constructing and debugging simulation models. We demonstrated the effectiveness of our modeling approach for GoogLeNet which simulates 46x faster than equivalent approximately-timed models with an average error of less than 1% in simulated time.

6.1.4 Cycle Accuracy in System-Level Memory Modeling

In Chapter 5, we extended our SystemC TLM framework to include cycle-accurate memory models of DRAM subsystems for accurate analysis of memory contention. We proposed an enhanced memory delay estimation for LT-CA modeling to improve both accuracy and fidelity of the design models. The extensive amount of contention on the globally shared memory hugely degrades the performance of data-intensive DNN applications. Therefore, we presented an alternative processor architecture with private local memories to reduce the average bandwidth on the shared memory. Finally, we analyzed the performance of LT, LT-CA, and AT with cycle-accurate memory models using GoogLeNet and SSD to provide early feedback to system designers for hardware implementation.

6.2 Future Work

We are facing the outset of three significant advances in the semiconductor industry, (1) readily fabrication of thousands of processing cores in a single chip (2) rapid innovations in compute-memory integration (3) availability of open-source integrated circuit (IC) design tools. We are also experiencing a boom in data-intensive applications ranging from artificial intelligence (AI) and machine learning, through virtual reality and 5G/6G wireless communications to genome sequencing and drug discovery. These technological breakthroughs and application trends require rethinking the conventional von Neumann computing paradigm. Research opportunities in design, modeling and simulation of post von Neumann computer systems that are capable of handling computationally expensive applications efficiently are numerous.

Here, we list a few research areas that we think TLM simulation framework would be an attractive approach to reduce both design time and cost of such future computer systems.

6.2.1 Extend the TLM Modeling Framework

We intend to apply our modeling framework to more DNNs and other memory-intensive applications. We plan to examine different scheduling policies and investigate its effect on simulation performance. Hence, we can enhance accuracy of LT-CA modeling in terms of total simulated time.

6.2.2 Explore Local Memory Topologies to Minimize Cost and Contention

We plan to conduct a trade-off analysis with respect to the costs incurred by local interconnects and local memories in terms of area, bandwidth and latency. Therefore, we can explore different local memory topologies to minimize both contention and chip cost.

6.2.3 Programming Models for Processing-In-Memory

Massively parallel processor arrays with tight compute-memory integration are a promising computer architecture to tackle the von Neumann memory bottleneck. However, the software stack and the system support for such new emerging computer systems are immature. Investigation of programming models, compilation flows and run-time system based on TLM simulation framework would help to increase the ease of programmability and reduce development costs for such novel computer systems.

6.2.4 New Memories for Compute-Memory Integration

We plan to investigate potential candidates for local on-chip memory integration. Nonvolatile memory devices such as Resistive switching Random Access Memory (RRAM), Ferroelectric RAM (FeRAM), and Phase Change Memory (PCM) have shown promising results as future new memories for compute-memory integration.

Bibliography

[1] E. M. Arasteh and R. Dömer. An Untimed SystemC Model of GoogLeNet. *Proceedings of the International Embedded Systems Symposium*, 2019.

[2] E. M. Arasteh and R. Dömer. Improving parallelism in system level models by assessing pdes performance. *Proceedings of Forum on Specification and Design Languages (FDL)*, sep 2021.

[3] E. M. Arasteh and R. Dömer. Systematic evaluation of six models of googlenet using pdes. Technical Report CECS-TR-21-03, CECS, UCI, september 2021.

[4] J. W. Backus. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM*, 21(8):613–641, 1978.

[5] D. Bertsekas and R. Gallager. *Data Networks (2nd Ed.)*. Prentice-Hall, Inc., USA, 1992.

[6] D. C. Black, J. Donovan, B. Bunton, and A. Keist. *SystemC: From the Ground Up, Second Edition*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[7] A. Bobrek, J. M. Paul, and D. E. Thomas. Shared resource access attributes for high-level contention models. In *2007 44th ACM/IEEE Design Automation Conference*, pages 720–725, 2007.

[8] L. Cai and D. Gajski. Transaction Level Modeling: An Overview. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, October 2003.

[9] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5:440–452, 1979.

[10] C. Chen and F. Lin. An easy-to-use approach for practical bus-based system design. *IEEE Trans. Computers*, 48(8):780–793, 1999.

[11] W. Chen, X. Han, C.-W. Chang, G. Liu, and R. Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(12):1859–1872, Dec. 2014.

[12] Z. Cheng and R. Dömer. Analyzing variable entanglement for parallel simulation of systemc tlm-2.0 models. *ACM Trans. Embed. Comput. Syst.*, 18(5s), Oct. 2019.

[13] B. Chopard, P. Combes, and J. Zory. A Conservative Approach to SystemC Parallelization. In *International Conference on Computational Science (4)*, pages 653–660, 2006.

[14] F. Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24, 2019.

[15] R. Dömer. Seven obstacles in the way of standard-compliant parallel SystemC simulation. *IEEE Embededded Systems Letters*, 8(4):81–84, Dec. 2016.

[16] R. Dömer, Z. Cheng, D. Mendoza, and E. M. Arasteh. Pushing the limits of parallel discrete event simulation for systemc. In *A Journey of Embedded and Cyber-Physical Systems*, 2021.

[17] W. Ecker, W. Müller, and R. Dömer. *Hardware-dependent Software – Principles and Practice*. Springer, Boston, MA, 2009.

[18] M. I. Frank, A. Agarwal, and M. K. Vernon. Lopc: Modeling contention in parallel algorithms. page 276–287, 1997.

[19] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, Oct 1990.

[20] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[21] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., USA, 1994.

[22] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

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

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

[25] P. Grun, N. D. Dutt, and A. Nicolau. *Memory architecture exploration for programmable embedded systems*. Kluwer, 2003.

[26] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, pages 2261–2269. IEEE Computer Society, 2017.

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

[28] Performance Analysis for Applications and Systems. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html`. Accessed: 2021-06-26.

[29] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[30] John Aynsley. TLM-2.0 base protocol checker. `https://www.doulos.com/knowhow/systemc/tlm2/at_example`, 2009. Accessed: 2020-02-02.

[31] M. Jung, C. Weis, and N. Wehn. Dramsys: A flexible dram subsystem design space exploration framework. *IPSJ Trans. Syst. LSI Des. Methodol.*, 8:63–74, 2015.

[32] M. Jung, C. Weis, N. Wehn, and K. Chandrasekar. Tlm modelling of 3d stacked wide i/o dram subsystems: A virtual platform for memory controller design space exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '13, New York, NY, USA, 2013. Association for Computing Machinery.

[33] A. Kaushik and H. D. Patel. Systemc-clang: An open-source framework for analyzing mixed-abstraction systemc models. In *Proceedings of the 2013 Forum on specification and Design Languages (FDL)*, pages 1–8, 2013.

[34] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[35] S. Kunzli, F. Poletti, L. Benini, and L. Thiele. Combining simulation and formal methods for system-level performance analysis. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, 2006.

[36] F. J. Kurdahi, D. D. Gajski, C. Ramachandran, and V. Chaiyakul. Linking register transfer and physical levels of design. In *IEICE Transsaction on Information and Systems*, volume E76-D, Sept. 1993.

[37] G. Larsson, M. Maire, and G. Shakhnarovich. Fractalnet: Ultra-deep neural networks without residuals. *CoRR*, abs/1605.07648, 2016.

[38] Y. Le Cun, L. D. Jackel, B. Boser, J. S. Denker, H. P. Graf, I. Guyon, D. Henderson, R. E. Howard, and W. Hubbard. Handwritten Digit Recognition: Applications of Neural Network Chips and Automatic Learning. *Comm. Mag.*, 27(11):41–46, Nov. 1989.

[39] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object Recognition with Gradient-Based Learning. In *Shape, Contour and Grouping in Computer Vision*, page 319, 1999.

[40] A. Leon-Garcia. *Probability, Statistics, and Random Processes for Electrical Engineering*. Pearson/Prentice Hall, third edition, 2008.

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

[42] G. Liu, T. Schmidt, Z. Cheng, D. Mendoza, and R. Dömer. RISC Compiler and Simulator, Release V0.6.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-TR-19-04, Center for Embedded and Cyber-physical Systems, University of California, Irvine, Sept. 2019.

[43] G. Liu, T. Schmidt, and R. Dömer. RISC Compiler and Simulator, Alpha Release V0.2.1: Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-TR-15-02, Center for Embedded and Cyber-physical Systems, University of California, Irvine, Oct. 2015.

[44] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. *Lecture Notes in Computer Science*, page 21–37, 2016.

[45] E. Malekzadeh Arasteh and R. Dömer. Fast loosely-timed system models with accurate memory contention. *ACM Transactions on Embedded Computing Systems*, under review.

[46] P. Marwedel. *Embedded Systems Design*. Springer International Publishing, 4th edition, 2021.

[47] M. Moy. Parallel programming with systemc for loosely timed models: A non-intrusive approach. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, page 9–14, San Jose, CA, USA, 2013. EDA Consortium.

[48] A. Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip, 2016.

[49] Accellera Systems Initiative, Core SystemC Language and Examples. `http://accellera.org/downloads/standards/systemc`. Accessed: 2022-07-22.

[50] OpenCV Tutorials, Load Caffe framework models. `https://docs.opencv.org/3.4/d5/de7/tutorial_dnn_googlenet.html`, 2018. Accessed: 2021-04-08.

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

[52] P. Panda, N. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings European Design and Test Conference. ED & TC 97*, pages 7–11, 1997.

[53] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[54] L. Roeder. Netron, Visualizer for neural network, deep learning, and machine learning models. `https://github.com/lutzroeder/netron`, 2017. Accessed: 2022-06-09.

[55] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker. Adaptive algorithm and tool flow for accelerating systemc on many-core architectures. In *2014 17th Euromicro Conference on Digital System Design*, pages 137–145, 2014.

[56] M. M. Sabry Aly, T. F. Wu, A. Bartolo, Y. H. Malviya, W. Hwang, G. Hills, I. Markov, M. Wootters, M. M. Shulaker, H. . Philip Wong, and S. Mitra. The n3xt approach to energy-efficient abundant-data computing. *Proceedings of the IEEE*, 107(1):19–48, 2019.

[57] G. Schirner and R. Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *ACM Trans. Embed. Comput. Syst.*, 8(1), Jan. 2009.

[58] G. Schirner, A. Gerstlauer, and R. Dömer. Fast and accurate processor models for efficient MPSoC design. *ACM Transactions on Design Automation of Electronic Systems*, 15(2):10:1–10:26, Mar. 2010.

[59] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A.-J. Boonstra. Near-memory computing: Past, present, and future, 2019.

[60] L. Steiner, M. Jung, F. S. Prado, K. Bykov, and N. Wehn. Dramsys4.0: A fast and cycle-accurate systemc/tlm-based DRAM simulator. In A. Orailoglu, M. Jung, and M. Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation - 20th International Conference, SAMOS 2020, Samos, Greece, July 5-9, 2020, Proceedings*, volume 12471 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2020.

[61] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[62] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pages 1–9, 2015.

[63] N. Ventroux and T. Sassolas. A new parallel systemc kernel leveraging manycore architectures. *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 487–492, 2016.

[64] E. Wandeler. Modular performance analysis and interface based design for embedded real time systems. 2006.

[65] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann. Systemc-link: Parallel systemc simulation using time-decoupled segments. In *Proceedings of the 2016 Conference on Design, Automation and Test in Europe*, DATE '16, page 493–498, San Jose, CA, USA, 2016. EDA Consortium.

[66] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto. Time-decoupled parallel systemc simulation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, 2014.

[67] H.-S. P. Wong, R. Willard, and I. K. Bell. Ic technology – what will the next node offer us? In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–52, 2019.

[68] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016.

[69] Yufei Ma, N. Suda, Yu Cao, J. Seo, and S. Vrudhula. Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Aug 2016.

[70] F. Zaruba, F. Schuiki, and L. Benini. Manticore: A 4096-core risc-v chiplet architecture for ultra-efficient floating-point computing, 2020.