Center for Embedded and Cyber-Physical Systems
University of California, Irvine

Systematic Evaluation of Six Models of GoogLeNet using PDES

Emad Malekzadeh Arasteh, Rainer Dömer

Technical Report CECS-21-03
September 27, 2021

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

emalekza,doemer@uci.edu
http://www.cecs.uci.edu

Systematic Evaluation of Six Models of GoogLeNet using PDES

Emad Malekzadeh Arasteh, Rainer Dömer

Technical Report CECS-21-03
September 27, 2021

Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
(949) 824-8919

emalekza,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

Convolutional neural networks (CNN) are a class of artificial neural networks, commonly used to solve the image classification problem. Exploring parallelism available in a CNN model deepens our understanding of its behavior and enables simulation speedup. In this report, we describe six untimed TLM-1.0 and TLM-2.0 SystemC models of GoogLeNet, a state of the art deep CNN using OpenCV library. The models are designed to examine different opportunities for parallelism. Towards this end, we use Recoding Infrastructure for SystemC (RISC) to exploit the introduced parallelism and provide extensive experimental results for all six models on four different hardware platforms over five RISC versions. The results confirm four hypotheses, (H1)

more aggressive simulation modes exploit more parallelism, (H2) newer RISC versions show higher simulation speedup, (H3) less restrictive transaction types enable higher parallelism and (H4) abstract TLM-1.0 models carry less workload than memory accurate TLM-2.0 models.

Contents

List of Figures

List of Tables

List of Listings

Systematic Evaluation of Six Models of GoogLeNet using PDES

E. M. Arasteh, R. Dömer
Center for Embedded and Cyber-Physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
emalekza,doemer@uci.edu
http://www.cecs.uci.edu

Abstract

Convolutional neural networks (CNN) are a class of artificial neural networks, commonly used to solve the image classification problem. Exploring parallelism available in a CNN model deepens our understanding of its behavior and enables simulation speedup. In this report, we describe six untimed TLM-1.0 and TLM-2.0 SystemC models of GoogLeNet, a state of the art deep CNN using OpenCV library. The models are designed to examine different opportunities for parallelism. Towards this end, we use Recoding Infrastructure for SystemC (RISC) to exploit the introduced parallelism and provide extensive experimental results for all six models on four different hardware platforms over five RISC versions. The results confirm four hypotheses, (H1) more aggressive simulation modes exploit more parallelism, (H2) newer RISC versions show higher simulation speedup, (H3) less restrictive transaction types enable higher parallelism and (H4) abstract TLM-1.0 models carry less workload than memory accurate TLM-2.0 models.

1 Introduction

Computer vision (CV) as a scientific field aims to gain understanding of images and video. CV covers a wide range of tasks, such as object recognition, scene understanding, human motion recognition, etc. One of the core problems in visual recognition is image classification.

1.1 Image Classification using CNN

Image classification is the problem of assigning a descriptive label to an input image from a fixed set of categories. Deep learning and convolutional neural network (CNN) have been shown to solve this hard image classification problem fast and with acceptable precision.

Early work on CNN dates back to 1989 with the LeNet network for handwritten digit recognition [7]. However, the early 2010s started a new era for CNN applications by the introduction of AlexNet [6] 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 [13].

A CNN mainly consists of alternating con-

volution 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. For example, LeNet-5, a CNN for digit recognition, as depicted in Figure 1, contains three convolution layers, two sub-sampling layers, and one fully connected layer [8].

In this report, we describe details of untimed TLM-1.0 and TLM-2.0 SystemC models of GoogLeNet [14], a state of the art deep CNN. We extend the original model initially described in SystemC [1] to six variants to explore higher level of parallelism available in the model.

The rest of this paper is organized as follows: Subsection 1.2 describes high level structure of GoogLeNet. Section 2 outlines four hypotheses regarding models behavior and RISC improvements. Section 3 describes SystemC modeling details of each layer and the overall GoogLeNet models. Section presents simulation results of all six models with analysis of each hypothesis. At last, Section 5 concludes this case study.

1.2 GoogLeNet Structure

GoogLeNet is a deep CNN for image classification and detection that was the winner of the ImageNet Large Scale Recognition Competition (ILSVRC) in 2014 with only 6.67% top-5 error [14]. GoogLeNet was proposed and designed with computational efficiency and deployability in mind. The two main features of GoogLeNet are (1) using 1x1 convolution layer for dimension reduction and (2) applying Network-in-Network architecture to increase representational power of the neural network [14].

GoogLeNet is 22 layers deep when counting only layers with parameters. The overall number of layers (independent building blocks) is 142 distinct layers. The main constituent layers are convolution, pooling, concatenation and classifier. GoogLeNet includes two auxiliary classifiers that are used during training to combat the vanishing gradient problem. The detailed types of layers inside GoogLeNet and the number of each type of layers are summarized in Table 1.

Table 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

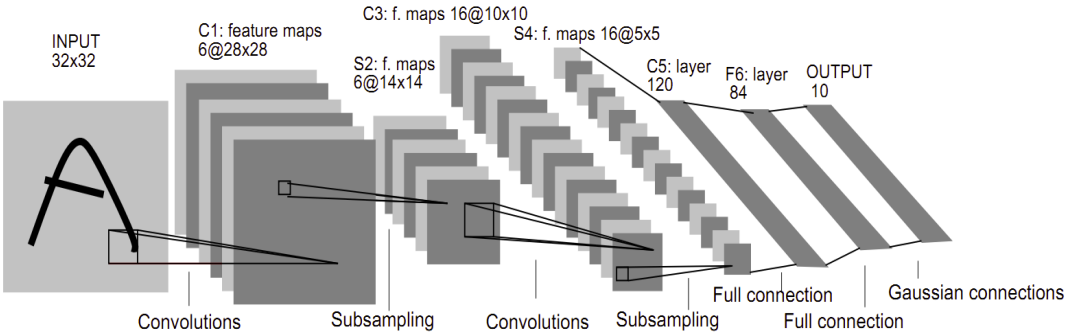

Figure 1: Architecture of LeNet-5, a CNN for digits recognition [8]

Our focus for now is on inference by using the proposed neural network architecture and not training for fine-tuning network parameters or suggesting improved network architecture. Therefore, our model does not include the two auxiliary classifier layers.

A schematic view of GoogLeNet is depicted in Figure 2. An image is fed in from the bottom, and processed by all layers. Then, a vector with probabilities for the set of categories comes out on the top. 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".

To get 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 [5]. The GoogLeNet Caffe model comes with (1) a binary file `.caffemodel` that contains network parameters, and (2) a text file `.prototxt` that specifies network architecture. Including weights and bias values, there are a total of 5.97 million learned parameters in GoogLeNet.

We also use another text file listing 1000 labels used in ILSVRC 2012 challenge that includes a synonym ring or synset of those labels.

2 Hypotheses

Based on the initial TLM-1.0 model of GoogLeNet [1], we start by designing various SystemC models to expose the inherent parallelism in GoogLeNet. Having RISC infrastructure available, we are equipped with a SystemC aware compiler and parallel simulator to exploit this introduced parallelism. This is beneficial for model exploration and faster simulation. Having said that, we have devised four initial hypotheses considering the models and RISC versions as follows:

2.1 H1: More aggressive simulation modes exploit more parallelism

The SystemC Accellera proof-of-concept simulator is based on co-routine semantics [4], hence it schedules only a single thread at each simulation step. In contrast, Parallel Discrete Event Simulation (PDES) allows paral-

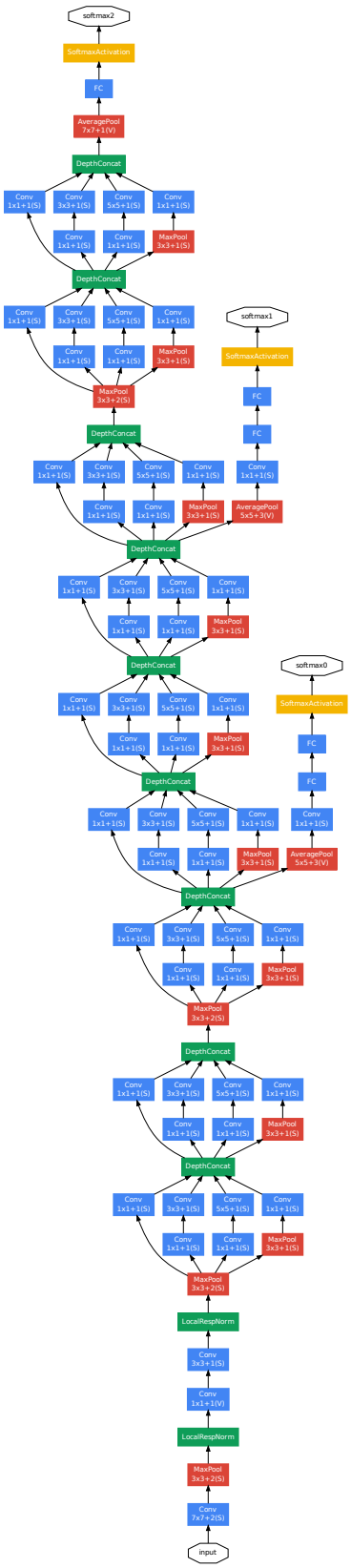

Figure 2: GoogLeNet network with all the bells and whistles [14]

lel simulation on multi-core processors. The PDES approach, by imposing a total order on event delivery and time advance, makes delta- and time-cycles absolute barriers for thread execution. Instead, by analyzing potential data dependencies in the model, RISC introduces out-of-order PDES by breaking the simulation-cycle barrier and as well as letting data-independent threads run out-of-order and in parallel [3].

Therefore, we expect that simulation time using the Accellera reference simulator will be longer than using PDES, and that simulation time using PDES will be longer than that using OoO PDES on multi-core processors.

2.2 H2: Newer RISC versions show higher speedup

RISC has been under continuous development since its introduction in 2014 at the Center for Embedded and Cyber-physical Systems at UCI, and each release adds various new features and makes the infrastructure more efficient and stable. The latest RISC versions available for this work are: V0.5.1, V0.5.2, V0.5.3 [9], V0.6.0 and V0.6.1 [10]. Since TLM-2.0 support is added from V0.5.3, TLM-2.0 models are only built and simulated using the three latest RISC versions since older versions do not support TLM-2.0 modeling style..

We expect that the performance of every RISC version continuously improves and that therefore, the latest version shall deliver the best performance in terms of simulator run time.

2.3 H3: Less restrictive transaction types enable higher parallelism

TLM-1.0 transactions are modeled with buffers inside a channel. SystemC offers a predefined primitive channel `sc_fifo` that implements read and write access functionality. Write function in turn calls `request_update()` function that causes the scheduler to queue an update request for the current primitive channel. Calls to `request_update()` run sequentially in the scheduler and degrade simulation performance. Therefore, designing a customized channel without calls to `request_update()` can improve the simulation performance. Furthermore, using channels with customized buffer sizes can increase the parallelism available in the model. Lastly, channels that do not induce any wait statement for write access can increase the simulation performance even further.

2.4 H4: Abstract TLM-1.0 carries less workload than memory accurate TLM-2.0

As previously known, the higher abstraction level of a model, the faster its simulation will be. We also know that TLM-2.0 standard provides the facilities such as core interfaces, sockets, generic payload and base protocol for modeling memory-mapped buses with explicit support for timing annotation. The memory accuracy exhibited in the TLM-2.0 standard incurs more overhead to simulation compared to the abstract TLM-1.0 model. Adding these extra implementation details to a model may slow down simulation speed for TLM-2.0 models. Hence, we expect simulation time for TLM-2.0 models to be longer than for the TLM-1.0 models.

3 SystemC TLM modeling of GoogLeNet

We now describe how we design SystemC TLM-1.0 and TLM-2.0 models of GoogLeNet.

3.1 Reference Model using OpenCV

Our SystemC models of GoogLeNet are implemented based on an original model using OpenCV 3.4.1, a library of computer vision functions mainly aimed for real-time applications written in C/C++ [12]. 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 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 paper. OpenCV also provides utility functions to load an image and read a Caffe model from `.prototxt` and `.caffemodel` files.

3.2 TLM Modeling Goals

Given the OpenCV primitives, we set three design goals in the early stage of model development [1] as follows:

1. *Generic layers*: Since GoogLeNet is composed of only 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 case 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. *Reuseable and modular code*: 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 autogenerated using only the network model defined by Caffe model files.

Note that these goals will allow us to easily generate SystemC models 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 [9][10], as described in Section 3.8 below.

Furthermore, to have models with practical significance, we set three extra goals for TLM modeling:

4. *Maximum throughput*: Model shall process as many images as possible in a shortest possible amount of time.

5. *Maximum parallelism*: Model shall demonstrate maximum parallelism.

6. *Minimum buffer*: Model shall use minimum number of buffers.

3.3 TLM-1.0 Layer Implementation

Each layer in the CNN is defined as a `sc_module` with one input port and one output port. Ports are defined as `sc_port` and are parameterized either by `sc_fifo_in_if` and `sc_fifo_out_if` primitive interface classes or our own defined interface classes, `mat_in_if` and `mat_out_if`. These 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 a granularity of port parameterization simplifies the design by focusing on the proper level of abstraction at this stage of modeling. Depending on the communication mechanism, the appropriate interface class is derived and plugged in channel declaration. As an example, the module definition of the first convolution layer `conv1_7x7_s2` is shown in Listing 1.

As shown in lines 38-50 of Listing 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 pretrained Caffe model files and stores weight and bias values in the `weights` and `bias` member variables.

Listing 2 shows the definition of the main method for `conv1_7x7_s2` in TLM-1.0 modeling style. Main method contains an endless loop that continuously reads the input port, processes the received data and writes the results to the output port.

Each module has also a main thread that continuously reads its input port, computes results, and writes those to its output port. Data processing is handled by the `run` method. Here, we rely on OpenCV to perform the computations. 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 3 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.

3.3.1 Channel variants

We develop multiple channels to explore parallelism available in the model. These channels differ in (1) their way of interacting with the scheduler and (2) their buffer sizes. We classify the channels based on their interaction mechanism with the scheduler into three categories as follows:

1. *Blocking channel*: In a blocking channel, read access suspends once the buffer is empty and write access suspends once the buffer is full.

2. *Non-blocking channel*: In a non-blocking channel, write access does not suspend and continuously accepts new elements to place in its buffer. If the buffer is already full, there is a risk that the buffer will be overwritten with a new data. On the other hand, read access suspends once there is no element in the buffer to read.

3. *SystemC FIFO channel*: This channel is built on the predefined primitive channel `sc_fifo` with default read and write member functions to access elements in the buffer.

As a primitive channel, `sc_fifo`, calls the `request_update()` function to queue an update request in the scheduler. In contrast, the user-defined channels, blocking and non-blocking channels, do not have any call to `request_update()`. Hence, they do not impose any sequential execution on the scheduler. Furthermore, a non-blocking channel does not induce any wait statement in its write access function, so it increases the possibility that an out-of-order PDES simulator schedules more aggressively.

Buffer size in a channel is another factor that affects the level of parallelism available in the model. The more buffers that exist in the channels, the more possibilities there are for pipelining of images in the network. However, increasing buffer sizes of all channels without thorough inspection of the model does not have any practical significance. In that regard, GoogLeNet can also be seen as stacks

```
const int conv1_7x7_s2_t::weight_sz[4] = {64, 3, 7, 7};
const int conv1_7x7_s2_t::bias_sz[4]   = {1, 1, 1, 64};

class conv1_7x7_s2_t : sc_module
{
public:
  sc_port<mat_in_if>  blob_in0;
  sc_port<mat_out_if> blob_out0;

  SC_HAS_PROCESS(conv1_7x7_s2_t);

  conv1_7x7_s2_t(sc_module_name n_,
     String name_,
     unsigned int num_output_,
     unsigned int pad_,
     unsigned int kernel_size_,
     unsigned int stride_,
     unsigned int dilation_,
     unsigned int group_) :
    sc_module(n_),
    name(name_),
    num_output(num_output_),
    pad(pad_),
    kernel_size(kernel_size_),
    stride(stride_),
    dilation(dilation_),
    group(group_),
    weights(4, weight_sz, CV_32F, weight_data),
    bias(4, bias_sz, CV_32F, bias_data)
  {
    load();
    SC_THREAD(main)
  }

  void load();
  void main();;
  void run(std::vector<Mat> &inpVec,
     std::vector<Mat> &outVec);

private:
  String            name;
  unsigned int      num_output;
  unsigned int      pad;
  unsigned int      kernel_size;
  unsigned int      stride;
  unsigned int      dilation;
  unsigned int      group;
  static const int  weight_sz[4];
  unsigned int      weight_data[64*3*7*7];
  static const int  bias_sz[4];
  unsigned int      bias_data[64];
  Mat               weights;
  Mat               bias;
};
```

Listing 1: TLM-1.0 conv1_7x7_s2 module definition

```
void conv1_7x7_s2_t::main()
{
  std::vector<Mat> inpVec(1), outVec(1);

  while (1)
  {
    inpVec[0] = blob_in0->read();
    run(inpVec, outVec);
    blob_out0->write(outVec[0]);
  }
}
```

Listing 2: TLM-1.0 conv1_7x7_s2 main method definition

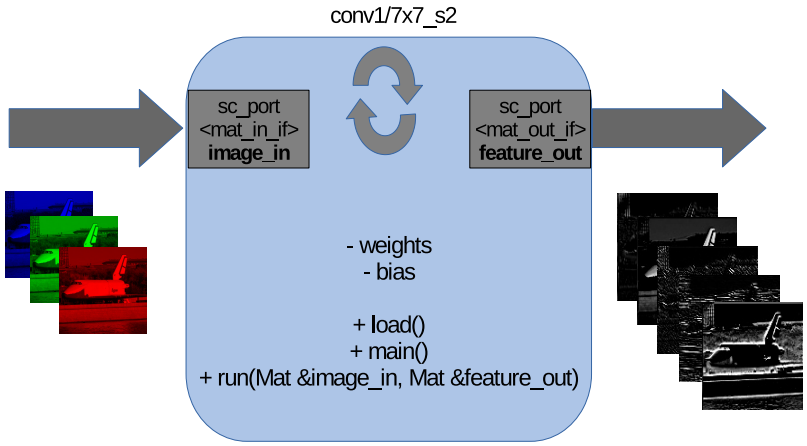

Figure 3: Schematic view of a SystemC convolution module

of layers group together under a so called "inception module". Each inception module contains multiple convolution, ReLU and pooling layers with different attributes. For example, the first inception module in GoogLeNet is depicted in Figure 4. As shown with down arrows, each inception module has four parallel tracks with various workloads.

Given the TLM modeling goals, channels with double buffers increase the parallelism in the model. While the producer layer writes to the front buffer, the consumer layer simultaneously reads the data from the back buffer, and vice versa. To maintain a continuous stream of images in every delta cycle, the channels connected to the output layer of the inception module in branch 0 and branch 3 require extra buffers. Therefore, by a adopting double buffering scheme, the channels connected to the output layer require 4, 2, 2 and 3 buffers in tracks 0, 1, 2 and 3 respectively.

In case of modeling with non-blocking

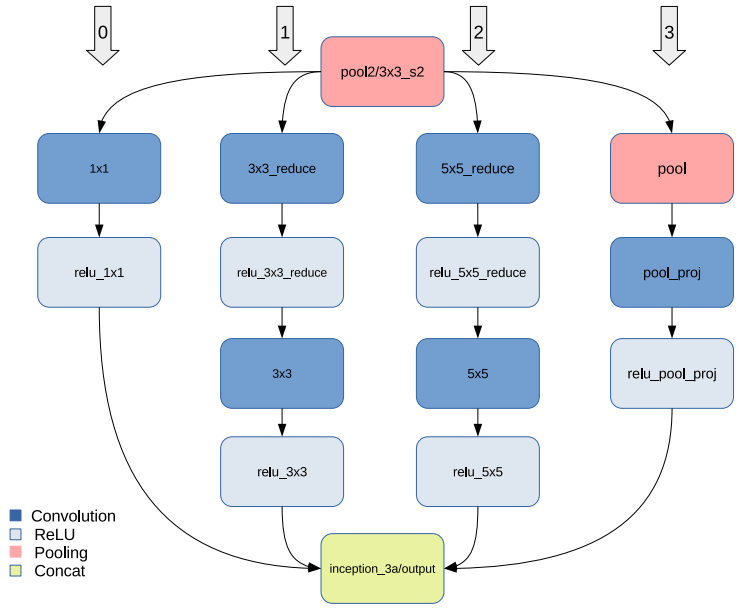

Figure 4: Inception module in GoogLeNet

channels, channels should have enough free slots to avoid any buffer overflow. Since the main threads in layers run indeterministicly, it can happen that the producer layer writes to the channel before the consumer layer starts to read the content of the channel from the previous delta cycle. This requires the channel to have enough free slots for reading the content of the current delta cycle and also the previous delta cycles. In the worst case scenario, all producer layers write to the channel before consumer layers read the channel. To dimension the channel sizes for this worst case scenario, non-blocking channels should have space for the maximum total number of producer layers in the entire model plus one, namely 63 elements.

Given the channel types and channel sizes, we develop four variants of TLM-1.0 models listed in Table 2. First, we start with a TLM-1.0 model using `sc_fifo` with buffer size of one. Due to its simplicity, measurements for this model haven't been taken. Second, we identify that double buffers and extra buffers inside the inception layers can increase the available parallelism. Hence, we develop `tlm1_sc_mul`. Third, we use our own user-defined channels (blocking channel) with one single element to avoid calls to `request_update()` in the write access function (`tlm1_blk_min`). Fourth, we increase the number of buffers in blocking channels and instantiate those channels in the model (`tlm1_blk_mul`). Fifth, we replace blocking channels with non-blocking channels with buffer size of 63 elements to remove any induced wait statements in write access function (`tlm1_nb_max`).

Table 2: TLM-1.0 models summary

Model name	Description
tlm1_blk_min	Blocking channels with buffer size of 1
tlm1_blk_mul	Blocking channels with double buffers and (4, 2, 2, 3) buffers in the output layer of inceptions
tlm1_sc_mul	SystemC FIFO channels with double buffers and (4, 2, 2, 3) buffers in the output layer of inceptions
tlm1_nb_max	Non-blocking channels with buffer size of 63

3.4 TLM-2.0 Layer Implementation

In TLM-2.0 modeling of GoogLeNet, input and output ports are replaced with initiator sockets connected to target sockets on a shared memory. The communication is realized through memory-mapped modules and each module has a dedicated address space inside the memory to read and write buffers. Figure 5 shows the connections of initiator sockets of the first convolution and ReLU layers in GoogLeNet to target sockets of shared memory.

To notify the consumer layer when the producer fills the shared buffer, two handshake protocols are devised: (1) feed forward (2) double. In feed forward, the producer simply notifies the consumer via an event once the buffer is filled. In double handshake, the consumer reads the buffer when the producer has sent a notification for a new buffer as well as when the consumer is ready to process that new buffer. Figure 6 illustrates the event connections between conv1/7x7_s and conv1/relu_7x7 layers in both feed forward and double handshakes. Feed forward handshake is fragile and only works with Accellera simulator and the synchronous mode of a parallel simulator but double handshake works with all simulation modes including non-prediction and out-of-order.

As event connection is depicted in Figure 6, to read a new buffer, conv1/7x7_s2 module waits for notification from the previous layer via a strobe event (in0_stb) and ensures conv1/relu_7x7 is also ready to accept the processed data via its ready event (out0_ready). Once conv1/7x7_s2 receives both notifications, it reads the data, processes, and writes to a buffer for conv1/relu_7x7 to fetch. Finally, conv1/7x7_s2 notifies conv1/relu_7x7 that the new data is ready to process via strobe event (out0_stb) and also notifies the previous layer via ready event (in0_ready) to signal that it is ready to accept a new buffer to read. As an example, the module definition of the first convolution layer `conv1_7x7_s2` in TLM-2.0 modeling style is shown in Listing 3. Table 3 summarizes the properties of two developed TLM-2.0 models.

Listing 4 shows the definition of the main method for `conv1_7x7_s2` in TLM-2.0 modeling style. Main method contains an endless loop that continuously reads the input data from a buffer in the memory, processes and writes the result back to the memory. First, the module waits for a start event from the previ-

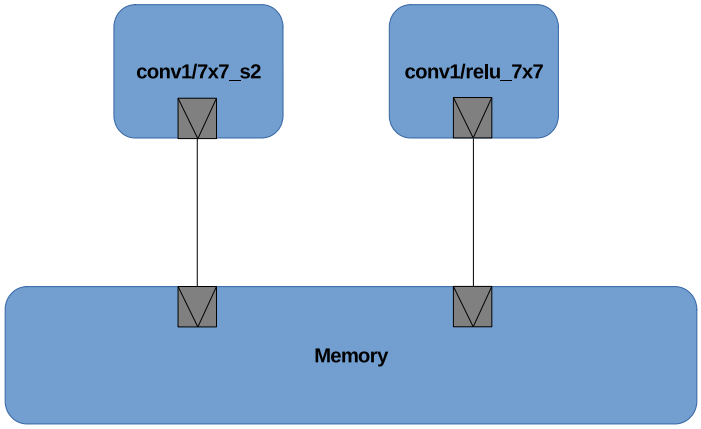

Figure 5: TLM-2.0 model connections

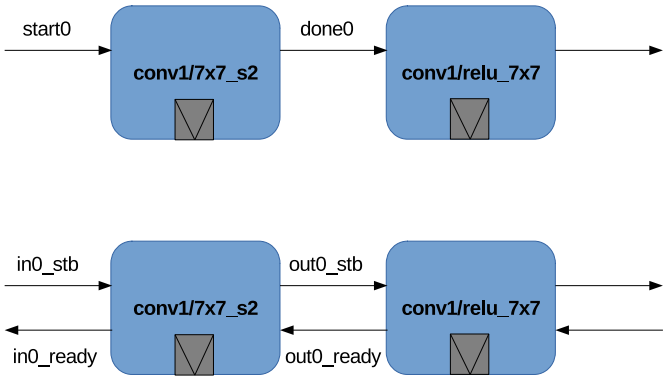

Figure 6: (Top) feed forward (bottom) double handshake mechanism in TLM-2.0 model

ous layer to ensure that the input data is in the memory (line 18). Then, it generates a read transaction using a generic payload to read the input buffer from the memory (lines 20-28). After processing the input data, it generates a write transaction using the same generic payload to write the result to the output buffer in memory (lines 36-44). Finally, it notifies the next layer via a done event that its input data is ready to fetch (line 50).

3.5 Netspec Generator

Each SystemC module has specific attributes based on its layer type and its corresponding TLM model. Writing module declara-

```
const int conv1_7x7_s2_t::weight_sz[4] = {64, 3, 7, 7};
const int conv1_7x7_s2_t::bias_sz[4]   = {1, 1, 1, 64};

class conv1_7x7_s2_t : sc_module
{
public:
  simple_initiator_socket<conv1_7x7_s2_t> port;
  sc_event &in0_stb;
  sc_event &in0_ready;
  sc_event &out0_stb;
  sc_event &out0_ready;

  SC_HAS_PROCESS(conv1_7x7_s2_t);

  conv1_7x7_s2_t(sc_module_name n_,
     String name_,
     unsigned int num_output_,
     unsigned int pad_,
     unsigned int kernel_size_,
     unsigned int stride_,
     unsigned int dilation_,
     unsigned int group_,
     unsigned int num_in0_buf_,
     unsigned int num_out_buf_,
     sc_event &in0_stb_,
     sc_event &in0_ready_,
     sc_event &out0_stb_,
     sc_event &out0_ready_) :
    sc_module(n_),
    name(name_),
    num_output(num_output_),
    pad(pad_),
    kernel_size(kernel_size_),
    stride(stride_),
    dilation(dilation_),
    group(group_),
    num_in0_buf(num_in0_buf_),
    num_out_buf(num_out_buf_),
    in0_stb(in0_stb_),
    in0_ready(in0_ready_),
    out0_stb(out0_stb_),
    out0_ready(out0_ready_),
    weights(4, weight_sz, CV_32F, weight_data),
    bias(4, bias_sz, CV_32F, bias_data)
  {
    load();
```

Listing 2: TLM-2.0 conv1_7x7_s2 module definition

```
47      SC_THREAD(main)
48    }
49
50    void main();
51    void run(std::vector<Mat> &inpVec,
52       std::vector<Mat> &outVec);
53    void load();
54
55  private:
56    String name;
57    unsigned int num_output;
58    unsigned int pad;
59    unsigned int kernel_size;
60    unsigned int stride;
61    unsigned int dilation;
62    unsigned int group;
63    static const int  weight_sz[4];
64    unsigned int      weight_data[64*3*7*7];
65    static const int  bias_sz[4];
66    unsigned int      bias_data[64];
67    Mat               weights;
68    Mat               bias;
69    unsigned int      num_in0_buf;
70    unsigned int      num_out_buf;
71    static const int  in0_sz[4];
72    unsigned int      in0_data[1*3*224*224];
73    static const int  out_sz[4];
74    unsigned int      out_data[1*64*112*112];
75  };
```

Listing 3: TLM-2.0 conv1_7x7_s2 module definition (cont)

Table 3: TLM-2.0 models summary

Model name	Description
tlm2_nil_mul	Double buffers and (4, 2, 2, 3) buffers in the output layer of inception with feed forward handshake
tlm2_db_mul	Double buffers and (4, 2, 2, 3) buffers in the output layer of inception with double handshake

tions by hand for 142 SystemC modules for each of the six TLM models is a tremendously error-prone and tedious task. Furthermore, declaring all modules and interconnections in the top level GoogLeNet module, instantiating them with the correct parameters,

```
const int conv1_7x7_s2_t::in0_sz[4] = {1, 3, 224, 224};
const int conv1_7x7_s2_t::out_sz[4] = {1, 64, 112, 112};

void conv1_7x7_s2_t::main()
{
  std::vector<Mat>          inpVec(1), outVec(1);
  tlm::tlm_generic_payload  trans;
  sc_core::sc_time          delay = sc_core::SC_ZERO_TIME;
  unsigned int              i0 = 0;
  unsigned int              j = 0;
  unsigned int              in0_addr;
  unsigned int              out_addr;
  inpVec[0] = Mat(4, in0_sz, CV_32F, in0_data);
  outVec[0] = Mat(4, out_sz, CV_32F, out_data);

  while (1)
  {
    wait(start0);
    in0_addr = CONV1_7X7_S2_IN0_BUF0_ADDR + i0 * CONV1_7X7_S2_IN0_BUF_SIZE;
    trans.set_command(tlm::TLM_READ_COMMAND);
    trans.set_address(in0_addr);
    trans.set_data_ptr(inpVec[0].data);
    trans.set_data_length(sizeof(in0_data));
    trans.set_streaming_width(sizeof(in0_data));
    trans.set_byte_enable_ptr(0);
    trans.set_dmi_allowed(false);
    trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
    port->b_transport(trans, delay);
    if (trans.get_response_status() != tlm::TLM_OK_RESPONSE)
    {
      SC_REPORT_FATAL(name(), trans.get_response_string().c_str());
    }
    i0 = (i0 + 1) % num_in0_buf;
    run(inpVec, outVec);
    out_addr = CONV1_7X7_S2_OUT_BUF0_ADDR + j * CONV1_7X7_S2_OUT_BUF_SIZE ;
    trans.set_command(tlm::TLM_WRITE_COMMAND);
    trans.set_address(out_addr);
    trans.set_data_ptr(outVec[0].data);
    trans.set_data_length(sizeof(out_data));
    trans.set_streaming_width(sizeof(out_data));
    trans.set_byte_enable_ptr(0);
    trans.set_dmi_allowed(false);
    trans.set_response_status(tlm::TLM_INCOMPLETE_RESPONSE);
    port->b_transport(trans, delay);
    if (trans.get_response_status() != tlm::TLM_OK_RESPONSE)
    {
      SC_REPORT_FATAL(name(), trans.get_response_string().c_str());
    }
    j  = (j + 1)  % num_out_buf;
    done0.notify(sc_core::SC_ZERO_TIME);
  }
}
```


Listing 4: TLM-2.0 conv1_7x7_s2 main method definition with feed forward handshake

binding either queues or sockets to the right modules, and in the case of TLM-2.0 models, connecting events between neighboring modules, are all laborious tasks. Therefore, we develop a generator tool to automatically extract the network architecture from a textual protocol buffer `.prototxt` and the network learned parameters from a binary protocol buffer `.caffemodel`. The generator, called `netspec`, is written in Python and uses Python interface to Caffe library, `pyCaffe`, in order to read `.caffemodel` and `.prototxt` files to construct its internal data representation of the neural network. `Netspec` then uses this data structure to generate SystemC codes for all the modules, as well as the top level GoogLeNet module with all its interconnection.

`Netspec` generates both TLM-1.0 and TLM-2.0 models based on modeling type, buffer architecture and channel type. In the case of TLM-2.0 models, `netspec` generates an extensible memory module with an arbitrary number of target sockets. It also automatically generates an address map file based on buffer architecture and supports memory address generation for multiple buffers for any layer in the network.

3.6 Validation by Simulation

A top level test bench validates our GoogLeNet SystemC model against the reference OpenCV implementation. The top level test bench instantiates our SystemC GoogLeNet module which contains all modules inside the network with all the interconnection 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 7 (a) shows TLM-1.0 top-level test bench that stimulus and monitor are connected using FIFO with granularity of `Mat`. And Figure 7 (b) shows TLM-2.0 top-level test bench that stimulus and monitor are instead connected via sockets to shared memory inside DUT.

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 class and its probability against the reference model.

3.7 Parallelism

Stimulus module aims to feed the models every delta cycle to achieve a maximum throughput. The modules inside `tlm1_blk_min` and `tlm1_blk_mul` models can only process data every other delta cycle. That simply means every module is idle every other cycle and this reduces the throughput to half of the theoretical maximum throughput. The `tlm1_nb_max` model accepts a new image every delta cycle but this comes with a high price of 63 buffers inside all channels. The `tlm1_sc_mul` modules can process data every delta cycle with minimum number of buffers in channels.

The modules inside `tlm2_nil_mul` and `tlm2_db_mul` models can accept new input every delta cycle. These models show maximum parallelism with minimum possible buffers inside the memory. Therefore, our TLM-2 models can achieve the maximum theoretical throughput and have maximum parallelism with minimum number of buffers.

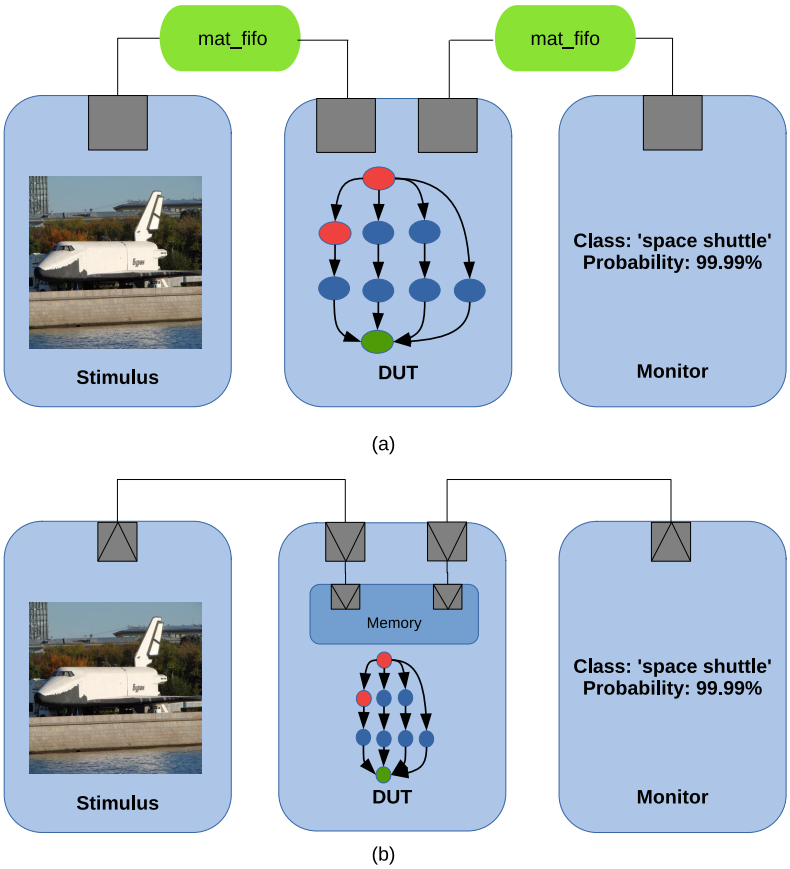

Figure 7: (a) TLM-1.0 (b) TLM-2.0 top-level test bench

3.8 Modular Source File Structure and Build Flow

Following good practices of SystemC coding, we place each module definition in a header file `.hpp` and the corresponding module implementation in a `.cpp` file. Also, to explore parallelism existing in the GoogLeNet system level model using RISC, we decide to split the implementation into two separate `.cpp` files. One `.cpp` file contains only methods that directly call OpenCV APIs ⟨*module_name_cv.cpp*⟩ and the other only contains the `main` method implementation that does not directly interact with OpenCV APIs ⟨*module_name.cpp*⟩. This prevents RISC from unnecessarily analyzing and instrumenting the code inside the OpenCV library, by only feeding object files generated from CV parts and not including OpenCV library source code.

First `.caffemodel` and `.protoxt` files are fed to the `netspec` tool to generate code for convolution modules and the overall GoogLeNet module. Once these modules are generated, all ⟨*module_name.cpp*⟩ and ⟨*module_name_cv.cpp*⟩ files are passed to

the GNU compiler to generate the object files. Then, the object files are passed all together to the GNU linker with OpenCV and SystemC libraries to obtain the final executable. Running the executable requires the Caffe model files to load convolution modules with weights and bias values and also a synset file to read the class names.

The build flow specifically for RISC requires minimum effort due to our early decision to split the OpenCV source code from the model source code. Since RISC prefers all the source code in a single file, all header files and implementation files are merged into one file. This flattened source code, with object files generated from the OpenCV part of the modules, is then fed to RISC which then generates a multithreaded parallel executable. Figure 9 depicts the build flow for RISC compilation and execution.

4 Results

Our untimed TLM-1.0 and TLM-2.0 SystemC models of GoogLeNet compile and simulate successfully with Accellera SystemC 2.3.1. For parallel simulation, we also compile and simulate the models using the five latest RISC versions. All simulation results match the OpenCV reference model output.

4.1 Performance Setup

We use two different computer platforms to benchmark the simulations. The specifications of each platform are shown in Table 4. 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 only on a single thread. Lastly, the stimulus module is configured to feed 500 images with size of 224x224 pixels to the model.

4.2 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 simulations are performed using RISC simulators V0.5.1, V0.5.2, V0.5.3, V0.6.0 and V0.6.1 in three simulation modes: synchronous (SYN), non-prediction (NPD) and out-of-order (OOO).

For reliability of the results, each measurement is performed three times. Later, if the distance of each recorded value (user time, system time and elapsed time) from its median is greater than $\pm 2\%$, that entire measurement is ignored. Among the remaining measurements, the first one is selected for further analysis.

Tables 11 to 14 in Appendix A show detailed results of measurements for four TLM-

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

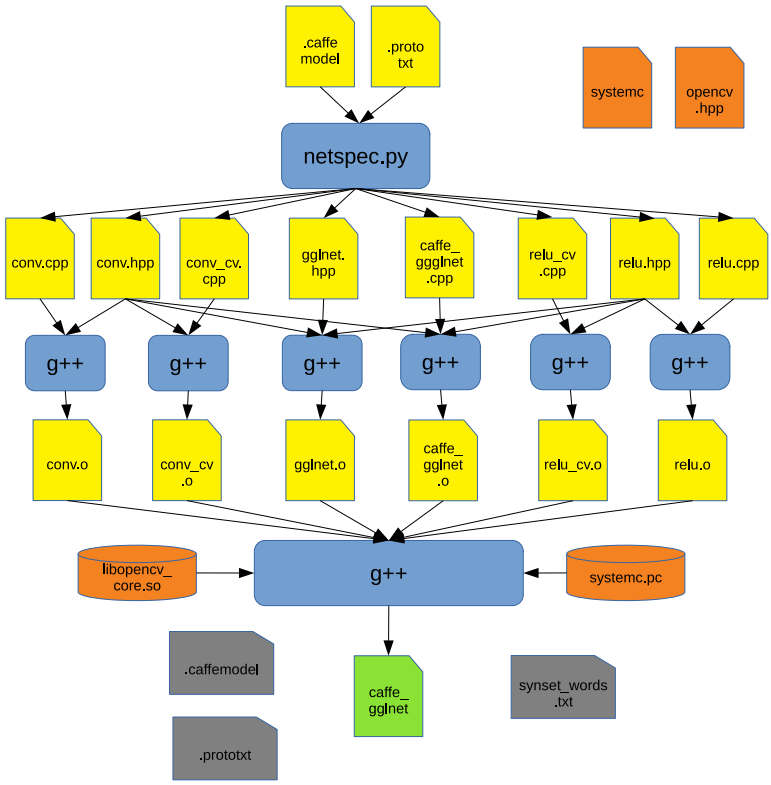

Figure 8: Build flow with Accellera SystemC

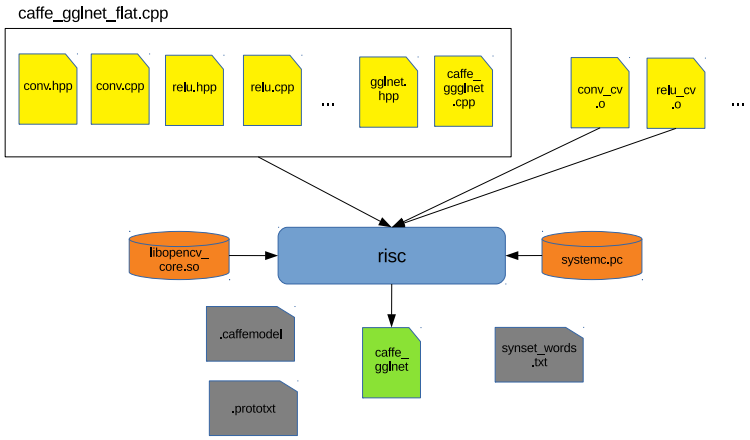

Figure 9: Build flow with RISC [10]

Table 4: Platform specification

Platform name	4-core (Omicron)	8-core (Omicron HT)	16-core (Phi)	32-core (Phi HT)
OS	CentOS 7.6	CentOS 7.6	CentOS 6.10	CentOS 6.10
CPU Model name	Intel Xeon E3-1240	Intel Xeon E3-1240	Intel Xeon E5-2680	Intel Xeon 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 cores	1	2	1	2

1.0 and two TLM-2.0 models for each simulation mode on four different platforms. 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. Furthermore, RISC support for TLM-2.0 was added from RISC V0.5.3.

4.3 Analysis

We analyze the measurement results obtained from the simulations of six models using five RISC versions on four hardware platforms. We create various heat map tables to identify the relevant results regarding each hypothesis. The results confirm the initial hypotheses described in Section 2 regarding simulation modes, RISC versions, transaction types and transaction level modeling.

4.3.1 H1: Simulation Modes

As discussed in Subsection 2.1, we expect more aggressive simulation modes to exploit more parallelism. Table 5 shows the heat map table for gained speedup in each simulation mode compared to sequential simulation. Each box in the table shows the simulation speedup using different simulation modes on a specific RISC version. Red color is used for minimum speedup, green for maximum and a linear gradient from red through yellow to green for values in between. For example, the top right box shows the speedup for all six models using RISC V0.6.1. As shown, switching from SEQ simulations to SYN simulations increases the speedup by 2.5x-2.8x and switching from SYN to NPD and OOO increases speedup even further to 2.8x-3.5x.

As mentioned earlier, TLM-2.0 support is added to RISC from V0.5.3. Hence, no speedup values are reported for TLM-2.0 models with earlier versions and those cells are colored gray. Furthermore, the `tlm2_nil_mul` model is not safe for out-of-order parallel simulation, so no speedup values for NPD and OOO simulations are reported for this model. It is worth mentioning that in the case of the `tlm1_nb_max` model, RISC V0.5.1 runs the model in SEQ mode even for parallel simulations which is fixed from V0.5.3 and later versions.

As seen in all the boxes in Table 5, in almost all simulation models, the speedup improves from sequential to synchronous, from synchronous to non-prediction and from non-prediction to out-of-order mode. The maximum speedup gained on the 4-core machine (omicron) is 3.52x which is very close to the optimal speedup of 4x. Comparing speedup results between 4/8-core and 16/32-core machines shows hyper-threading technology is largely ineffective for this application.

4.3.2 H2: RISC Versions

As explained in Subsection 2.2, we expect that newer RISC versions show higher speedup. Table 6 shows the heat map table for speedup of out-of-order simulations compared to sequential simulations using the five latest RISC versions. Each of the four main boxes is dedicated to speedups of a specific platform. Looking across the models in each box, the leftmost column shows the speedup for the oldest RISC version (RISC V0.5.1) and the rightmost column shows the speedup for the latest RISC version (RISC V0.6.1). Although support for TLM-2.0 models was added in RISC V0.5.3, that specific version has a bug and is unable to handle NPD and OOO simulations. Hence, no speedup numbers are reported for these two simulation modes under RISC V0.5.3.

As shown in all the platforms, the latest RISC version delivers the absolute best speedup for TLM-2.0 models. For TLM-1.0 models, the latest RISC version also delivers high speedup with a few exceptions. For example, RISC V0.5.2 shows slightly better values than RISC V0.6.1 and the reason is unclear for us at this time. Overall, continuous development of the RISC project proves to contribute to higher speedup for our models.

4.3.3 H3: Transaction Types

As pointed out in Subsection 2.3, less restrictive transaction types enable higher parallelism. Table 7 shows the elapsed time of the models in SYN, NPD and OOO simulation modes using RISC V0.6.1. The models are ordered based on time of development during the project, with the earliest developed model listed first.

Considering the 4-core machine (omicron), the first model, `tlm1_sc_mul` uses SystemC FIFOs to implement channels. SystemC FIFO forces synchronous simulation. Hence, the elapsed time of `tlm1_sc_mul` for SYN, NPD and OOO are almost identical as reflected in the first row. The three other TLM-1.0 models each use transactions that have more freedom to run in parallel. The `tlm1_blk_min` model removes calls to `request_update()` function. The `tlm1_blk_mul` model uses multiple buffers to increase the possibility for pipelining in addition to removing calls to `request_update()`. The `tlm1_nb_max` model removes wait statements in the write function to let the OOO scheduler schedule multiple threads together. As can be seen in the second, third and the fourth rows, the elapsed time for SYN simulation mode increases. However, the OOO simulation exploits the introduced parallelism and reports slightly better elapsed time than `tlm1_sc_mul`.

Table 8 shows the number of data conflicts and event notifications in all four TLM 1.0 models generated by the RISC compiler. As illustrated, the `tlm1_sc_mul` model, that uses `sc_fifo` and has calls to `request_update()`, does not have any event notifications. In contrast, the other

Omicron	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
Speedup	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	1.00	2.12	2.46	2.13	1.00	2.44	2.92	2.92	1.00	2.42	2.94	2.72	1.00	2.38	2.96	2.95	1.00	2.44	2.83	2.85
tlm1_blk_mul	1.00	2.19	2.54	2.27	1.00	2.45	3.11	3.16	1.00	2.56	2.96	3.00	1.00	2.41	3.12	3.16	1.00	2.64	2.82	2.84
tlm1_sc_mul	1.00	2.87	2.86	2.88	1.00	3.46	3.43	3.44	1.00	2.60	2.61	2.55	1.00	2.96	2.92	3.15	1.00	2.83	2.83	2.83
tlm1_nb_max	1.00	1.00	1.00	0.98	1.00	0.97	0.92	0.91	1.00	2.40	2.85	2.94	1.00	2.37	2.81	2.97	1.00	2.69	2.75	2.87
tlm2_nil_mul									1.00	2.51			1.00	2.30			1.00	2.80		
tlm2_db_mul									1.00	2.51			1.00	2.22	2.39	2.74	1.00	2.81	2.79	3.52

Omicron HT	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
Speedup	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	1.00	2.09	2.45	2.14	1.00	2.52	2.90	2.80	1.00	2.45	2.94	3.02	1.00	2.47	2.91	2.89	1.00	2.44	2.93	2.94
tlm1_blk_mul	1.00	2.12	2.55	2.17	1.00	2.54	2.92	2.91	1.00	2.51	2.92	3.01	1.00	2.51	2.90	2.94	1.00	2.69	2.91	2.94
tlm1_sc_mul	1.00	3.04	3.03	2.99	1.00	3.33	3.33	3.32	1.00	3.32	3.34	3.33	1.00	3.33	3.08	3.00	1.00	2.92	2.93	2.92
tlm1_nb_max	1.00	1.00	1.00	0.98	1.00	0.97	0.92	0.91	1.00	2.34	2.75	2.93	1.00	2.44	2.71	2.97	1.00	2.69	2.82	2.95
tlm2_nil_mul									1.00	2.85			1.00	2.47			1.00	2.91		
tlm2_db_mul									1.00	2.61			1.00	2.27	2.51	2.87	1.00	2.90	2.89	3.53

Phi	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
Speedup	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	1.00	2.60	3.46	2.84	1.00	3.57	4.52	4.51	1.00	3.38	4.54	4.70	1.00	3.51	4.58	4.57	1.00	3.56	4.87	4.89
tlm1_blk_mul	1.00	2.72	3.49	2.87	1.00	3.71	4.73	4.79	1.00	3.68	4.78	4.86	1.00	3.79	4.81	4.83	1.00	4.10	4.86	4.87
tlm1_sc_mul	1.00	4.96	4.99	4.94	1.00	5.29	5.29	5.29	1.00	5.27	5.27	5.26	1.00	4.63	5.15	4.68	1.00	4.80	4.78	4.85
tlm1_nb_max	1.00	0.99	0.99	0.98	1.00	0.97	0.93	0.91	1.00	3.26	4.08	4.54	1.00	3.41	4.18	4.65	1.00	4.06	4.28	4.70
tlm2_nil_mul									1.00	4.08			1.00	3.26			1.00	4.82		
tlm2_db_mul									1.00	3.13			1.00	2.80	3.01	4.14	1.00	4.81	4.80	5.61

Phi HT	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
Speedup	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	1.00	2.68	3.44	2.64	1.00	3.44	4.41	4.32	1.00	3.32	4.30	4.51	1.00	3.42	4.37	4.41	1.00	3.44	4.79	4.82
tlm1_blk_mul	1.00	2.81	3.44	2.79	1.00	3.73	4.57	4.58	1.00	3.64	4.55	4.66	1.00	3.72	4.58	4.61	1.00	3.95	4.78	4.76
tlm1_sc_mul	1.00	4.78	4.80	4.78	1.00	4.88	4.84	4.89	1.00	4.85	4.86	4.86	1.00	4.78	4.62	4.90	1.00	4.69	4.83	4.83
tlm1_nb_max	1.00	0.99	0.99	0.98	1.00	0.97	0.93	0.91	1.00	3.31	3.98	4.22	1.00	3.60	4.17	4.56	1.00	4.00	4.19	4.64
tlm2_nil_mul									1.00	3.88			1.00	3.29			1.00	4.70		
tlm2_db_mul									1.00	3.06			1.00	2.83	2.96	4.01	1.00	4.71	4.67	5.60

Table 5: Speedup heat map table for validating hypothesis H1

three models that use user-defined channels and omit the `request_update()` function, reports zero event notification.

As previously stated, the `tlm2_nil_mul` model is not safe for out-of-order parallel simulation, so elapsed time for NPD and OOO simulations are not reported for this model. While both `tlm2_nil_mul` and `tlm2_db_mul` models have similar elapsed time in SYN simulation mode, OOO simulation can again exploit a higher level of parallelism introduced in `tlm2_db_mul` model and reports the shortest simulation time. The exact same pattern applies to the other TLM-1.0 and TLM-2.0 models on machines with a higher number of cores.

Having models that use transaction types with less restrictions enables out-of-order parallel scheduler to exploit the opportunities for parallelism.

4.3.4 H4: TLM-1.0 vs TLM-2.0

As noted in Subsection 2.4, abstract TLM-1.0 models are expected to carry less workload than memory accurate TLM-2.0 models. Table 9 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 slightly longer elapsed time than the first

Omicron	risc_v0.5.1	risc_v0.5.2	risc_v0.5.3	risc_v0.6.0	risc_v0.6.1
Speedup	OOO	OOO	OOO	OOO	OOO
tlm1_blk_min	2.13	2.92	2.72	2.95	2.85
tlm1_blk_mul	2.27	3.16	3.00	3.16	2.84
tlm1_sc_mul	2.88	3.44	2.55	3.15	2.83
tlm1_nb_max	0.98	0.91	2.94	2.97	2.87
tlm2_nil_mul					
tlm2_db_mul				2.74	3.52

Omicron HT	risc_v0.5.1	risc_v0.5.2	risc_v0.5.3	risc_v0.6.0	risc_v0.6.1
Speedup	OOO	OOO	OOO	OOO	OOO
tlm1_blk_min	2.14	2.80	3.02	2.89	2.94
tlm1_blk_mul	2.17	2.91	3.01	2.94	2.94
tlm1_sc_mul	2.99	3.32	3.33	3.00	2.92
tlm1_nb_max	0.98	0.91	2.93	2.97	2.95
tlm2_nil_mul					
tlm2_db_mul				2.87	3.53

Phi	risc_v0.5.1	risc_v0.5.2	risc_v0.5.3	risc_v0.6.0	risc_v0.6.1
Speedup	OOO	OOO	OOO	OOO	OOO
tlm1_blk_min	2.84	4.51	4.70	4.57	4.89
tlm1_blk_mul	2.87	4.79	4.86	4.83	4.87
tlm1_sc_mul	4.94	5.29	5.26	4.68	4.85
tlm1_nb_max	0.98	0.91	4.54	4.65	4.70
tlm2_nil_mul					
tlm2_db_mul				4.14	5.61

Phi HT	risc_v0.5.1	risc_v0.5.2	risc_v0.5.3	risc_v0.6.0	risc_v0.6.1
Speedup	OOO	OOO	OOO	OOO	OOO
tlm1_blk_min	2.64	4.32	4.51	4.41	4.82
tlm1_blk_mul	2.79	4.58	4.66	4.61	4.76
tlm1_sc_mul	4.78	4.89	4.86	4.90	4.83
tlm1_nb_max	0.98	0.91	4.22	4.56	4.64
tlm2_nil_mul					
tlm2_db_mul				4.01	5.60

Table 6: Measurement results for validating hypothesis H2

four rows for TLM-1.0 models. This could come from the difference in number of memory copies in TLM-1.0 and TLM-2.0 models. TLM-1.0 models use shallow copy for assigning `Mat` objects in reading and writing to channels. However, TLM-2.0 models use two memory copies to read and write from/to the memory module. This can increase the workload for TLM-2.0 models in comparison with TLM-1.0 models.

The user time in out-of-order parallel simulation can also be interpreted as another indication for this hypothesis. Table 10 shows the heat map table for the user time in OOO

Omicron	risc_v0.6.1		
Elapsed time	SYN	NPD	OOO
tlm1_sc_mul	219.86	219.61	219.92
tlm1_blk_min	254.92	220.14	218.53
tlm1_blk_mul	234.35	219.24	217.33
tlm1_nb_max	231.45	226.15	217.14
tlm2_nil_mul	223.63		
tlm2_db_mul	223.81	225.52	178.33

Omicron HT	risc_v0.6.1		
Elapsed time	SYN	NPD	OOO
tlm1_sc_mul	213.09	212.43	212.81
tlm1_blk_min	255.23	212.18	211.9
tlm1_blk_mul	230.06	212.43	210.49
tlm1_nb_max	231.37	220.9	211.02
tlm2_nil_mul	215.28		
tlm2_db_mul	217.06	217.3	178.14

Phi	risc_v0.6.1		
Elapsed time	SYN	NPD	OOO
tlm1_sc_mul	197.02	197.93	194.9
tlm1_blk_min	266.66	194.77	193.94
tlm1_blk_mul	229.04	193.63	193.08
tlm1_nb_max	231.79	220.32	200.29
tlm2_nil_mul	198.31		
tlm2_db_mul	198.66	199.31	170.44

Phi HT	risc_v0.6.1		
Elapsed time	SYN	NPD	OOO
tlm1_sc_mul	213.09	212.43	212.81
tlm1_blk_min	276.04	198.4	197.17
tlm1_blk_mul	237.69	196.73	197.47
tlm1_nb_max	235.12	224.39	202.9
tlm2_nil_mul	203.29		
tlm2_db_mul	203.17	204.89	170.68

Table 7: Measurement results for validating hypothesis H3

simulation mode. Again, TLM-2.0 models have higher workload compared to TLM-1.0

Table 8: Data conflicts and event notifications in TLM-1.0 models

Model	Data	Event
(sc, mul)	1948	0
(blk, min)	9202	1403
(blk, mul)	9202	1403
(nb, max)	4145	423

	Omicron	Omicron HT	Phi	Phi HT
Elapsed time	SEQ	SEQ	SEQ	SEQ
tlm1_sc_mul	622.2	622.02	945.24	943.98
tlm1_blk_min	622.39	622.5	949.02	949.56
tlm1_blk_mul	618.11	618.36	940.12	939.61
tlm1_nb_max	623.04	622.93	941.93	940.93
tlm2_nil_mul	627.7	628	956.28	955.81
tlm2_db_mul	628.15	628.55	956.28	956.01

Table 9: Measurement results for validating hypothesis H4 (SEQ)

models for 4-core (omicron) and 16-core (phi) machines while hyper-threading technology brings some irregularities to our observation in 8-core and 32-core machines.

In summary, Figure 10 shows a 3D diagram of elapsed time for all six TLM models in OOO simulation mode on a 4-core machine. As illustrated, RISC releases generally keep improving the simulation speedup for each model. For example, the earlier RISC versions aren't able to exploit the parallelism available in in `tlm1_nb_max` but the later RISC versions exploit the parallelism and reduce the elapsed time drastically. Of all the models using the latest RISC version (front row), the `tlm2_db_mul` model has the highest level of parallelism and reports the shortest simulation elapsed time.

5 Conclusion

In this report, we described six untimed TLM-1.0 and TLM-2.0 SystemC models of GoogLeNet using OpenCV 3.4.1 library. We also developed a tool to automatically generate SystemC codes for all the TLM models from Caffe model files. We successfully simulated the generated TLM models using Accellera SystemC 2.3.1 and the five latest RISC versions.

Our extensive experimental results confirmed four hypotheses as follows: (1) more aggressive simulation modes exploited more parallelism, (2) newer RISC versions showed higher simulation speedup, (3) less restrictive transaction types enabled higher parallelism and (4) abstract TLM-1.0 models carried less

	Omicron	Omicron HT	Phi	Phi HT
User time	OOO	OOO	OOO	OOO
tlm1_sc_mul	662.16	1,049.94	931.58	1,259.13
tlm1_blk_min	659.50	1,043.16	927.68	1,250.82
tlm1_blk_mul	659.37	1,047.70	926.36	1,254.42
tlm1_nb_max	656.81	1,029.79	923.79	1,185.25
tlm2_nil_mul				
tlm2_db_mul	684.32	1,003.92	957.86	1,003.05

Table 10: Measurement results for validating hypothesis H4 (OOO)

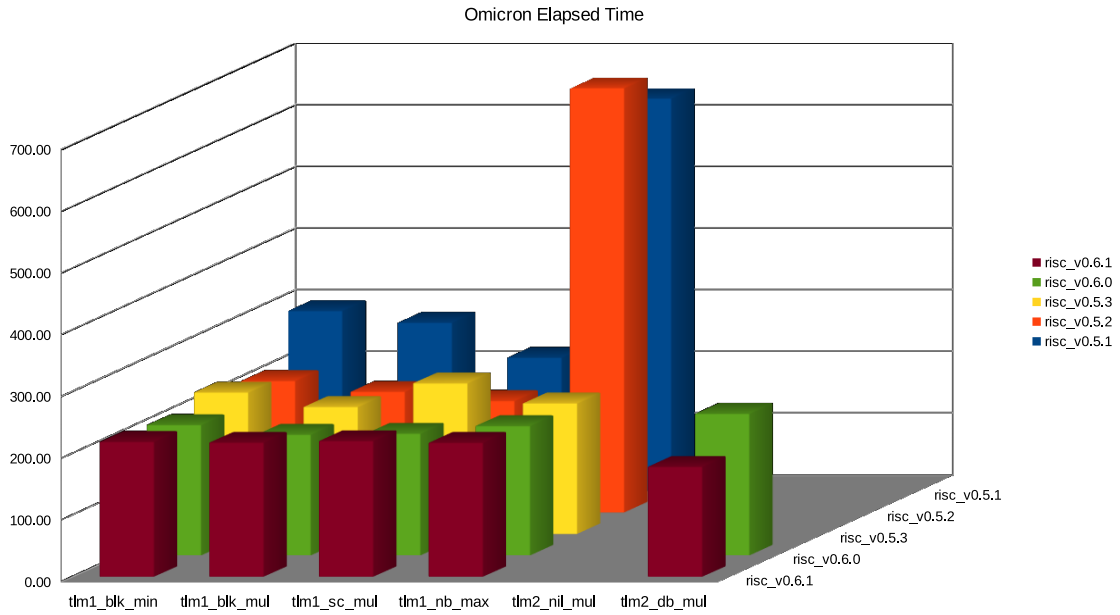

Figure 10: Elapsed time for OOO simulation on 4-core machine

workload than memory accurate TLM-2.0 models.

5.1 Future work

Memory accurate TLM-2.0 models enable detailed inspection of memory accesses which are generated by each module. In our future work, we plan to investigate memory access patterns in GoogLeNet and examine the famous memory bottleneck problem, better known as the von Neumann bottleneck [2].

References

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

[2] John 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.

[3] Weiwei Chen, Xu Han, and Rainer Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2012.

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

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[7] 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, November 1989.

[8] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object Recognition with Gradient-Based Learning. In *Shape, Contour and Grouping in Computer Vision*, page 319, 1999.

[9] Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza, and Rainer 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, September 2018.

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

[11] Daniel Mendoza and Rainer Dömer. A Tool for Visualization of SystemC Models. Technical Report CECS-TR-17-06, Center for Embedded and Cyber-physical Systems, University of California, Irvine, November 2017.

[12] OpenCV Tutorials, Load Caffe framework models. https://docs.opencv.org/3.4/d5/de7/tutorial_dnn_googlenet.html. Accessed: 2019-05-11.

[13] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pages 1–9, 2015.

A Measurements

Tables 11 to 14 shows detailed measurements of user time, system time, elapsed time and CPU usage for all TLM models across four hardware platforms. Table 15 summarizes the elapsed time part and illustrates the values in a heat map table.

	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
omicron_tlm1_blk_min	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	620.56	660.94	650.73	666.70	621.17	692.02	704.69	707.60	621.23	711.89	701.44	703.50	620.82	702.35	697.30	697.50	621.19	657.74	664.03	662.16
Sys	1.31	0.99	0.88	0.99	1.40	1.16	1.13	1.16	1.34	1.17	1.03	1.09	1.37	1.18	1.10	1.11	1.33	0.72	0.77	0.70
Elapsed	621.74	293.50	252.83	291.65	622.42	255.08	213.34	213.10	622.44	256.73	211.37	229.22	622.05	260.88	210.47	211.17	622.39	254.92	220.14	218.53
CPU	100%	225%	257%	228%	100%	271%	330%	332%	100%	277%	332%	307%	100%	269%	331%	330%	100%	258%	301%	303%
Speedup	1.00	2.12	2.46	2.13	1.00	2.44	2.92	2.92	1.00	2.42	2.94	2.72	1.00	2.38	2.96	2.95	1.00	2.44	2.83	2.85
omicron_tlm1_blk_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	617.60	652.49	646.26	656.03	617.07	667.85	678.88	680.35	617.01	681.47	677.02	678.91	617.67	675.63	674.99	676.23	617.46	655.94	661.49	659.50
Sys	0.79	0.72	0.55	0.72	0.76	0.80	0.75	0.73	0.78	0.81	0.72	0.66	0.76	0.81	0.72	0.74	0.70	0.53	0.57	0.64
Elapsed	618.35	282.02	242.98	272.54	617.77	252.11	198.52	195.62	617.74	241.27	208.74	205.75	618.37	257.07	198.02	195.55	618.11	234.35	219.24	217.33
CPU	100%	231%	266%	240%	100%	265%	342%	348%	100%	282%	324%	330%	100%	263%	341%	346%	100%	280%	301%	303%
Speedup	1.00	2.19	2.54	2.27	1.00	2.45	3.11	3.16	1.00	2.56	2.96	3.00	1.00	2.41	3.12	3.16	1.00	2.64	2.82	2.84
omicron_tlm1_sc_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	620.41	652.65	652.68	652.91	620.95	665.06	665.00	665.17	621.00	660.74	660.01	660.92	620.65	661.63	661.39	662.67	620.96	658.96	659.09	659.37
Sys	1.32	0.85	0.85	0.84	1.34	0.84	0.89	0.83	1.35	0.93	0.97	0.91	1.28	0.92	1.03	0.92	1.37	0.73	0.66	0.71
Elapsed	621.60	216.92	217.18	215.92	622.16	180.03	181.27	180.80	622.22	239.29	238.70	244.04	621.81	209.76	213.27	197.24	622.20	219.86	219.61	219.92
CPU	100%	301%	300%	302%	100%	369%	367%	368%	100%	276%	276%	271%	100%	315%	310%	336%	100%	300%	300%	300%
Speedup	1.00	2.87	2.86	2.88	1.00	3.46	3.43	3.44	1.00	2.60	2.61	2.55	1.00	2.96	2.92	3.15	1.00	2.83	2.83	2.83
omicron_tlm1_nb_max	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	620.88	623.84	623.93	634.44	620.76	644.34	676.13	686.24	620.79	664.40	666.34	668.87	621.00	665.95	662.19	665.75	621.22	655.42	656.92	656.81
Sys	2.07	1.50	1.59	1.65	2.03	2.17	2.20	2.17	1.98	1.42	1.32	1.34	1.92	1.42	1.32	1.30	1.95	1.30	1.24	1.37
Elapsed	622.82	624.39	624.43	634.99	622.67	644.61	676.32	686.41	622.64	259.93	218.67	211.51	622.80	263.09	221.89	209.65	623.04	231.45	226.15	217.14
CPU	100%	100%	100%	100%	100%	100%	100%	100%	100%	256%	305%	316%	100%	253%	299%	318%	100%	283%	291%	303%
Speedup	1.00	1.00	1.00	0.98	1.00	0.97	0.92	0.91	1.00	2.40	2.85	2.94	1.00	2.37	2.81	2.97	1.00	2.69	2.75	2.87
omicron_tlm2_nil_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									626.80	743.88			626.33	731.21			626.38	670.86		
Sys									1.62	1.38			1.62	1.42			1.54	0.84		
Elapsed									628.21	249.92			627.73	272.38			627.70	223.63		
CPU									100%	298%			100%	268%			100%	300%		
Speedup									1.00	2.51			1.00	2.30			1.00	2.80		
omicron_tlm2_db_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									626.87	783.36			627.26	762.91	723.94	791.96	626.69	671.50	685.86	684.32
Sys									1.50	1.56			1.60	1.52	1.36	1.64	1.63	0.79	0.93	1.10
Elapsed									628.21	250.61			628.69	282.91	263.46	229.37	628.15	223.81	225.52	178.33
CPU									100%	313%			100%	270%	275%	345%	100%	300%	304%	384%
Speedup									1.00	2.51			1.00	2.22	2.39	2.74	1.00	2.81	2.79	3.52

Table 11: Measurement results on 4-core host (‘omicron’, HT off)

	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
omicron_ht_tlm1_blk_min	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	621.93	942.77	964.95	963.94	621.73	999.94	1,058.72	1,050.32	621.78	1,021.09	1,063.58	1,073.60	622.08	1,014.96	1,048.22	1,041.64	621.37	983.85	1050.81	1049.94
Sys	1.27	1.44	1.05	1.53	1.30	1.49	1.65	1.61	1.31	1.55	1.41	1.56	1.22	1.56	1.47	1.51	1.26	1.11	1.23	1.07
Elapsed	622.99	298.75	253.89	291.46	622.83	247.55	214.49	222.64	622.89	253.79	212.13	206.27	623.09	252.19	214.11	215.72	622.5	255.23	212.18	211.9
CPU	100%	316%	380%	331%	100%	404%	494%	472%	100%	402%	502%	521%	100%	403%	490%	483%	100%	385%	495%	495%
Speedup	1.00	2.09	2.45	2.14	1.00	2.52	2.90	2.80	1.00	2.45	2.94	3.02	1.00	2.47	2.91	2.89	1	2.44	2.93	2.94
omicron_ht_tlm1_blk_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	618.14	925.08	960.56	939.24	618.49	979.55	1,038.15	1,041.82	617.60	1,002.62	1,043.16	1,054.66	617.66	991.52	1,034.14	1,034.64	617.66	1007.26	1046.1	1043.16
Sys	0.74	1.01	0.74	0.93	0.72	1.03	0.92	1.04	0.71	0.97	0.93	1.01	0.72	1.10	0.99	0.96	0.76	0.93	0.79	0.76
Elapsed	618.75	291.86	242.36	284.67	619.08	243.30	212.23	212.78	618.19	246.45	211.91	205.71	618.25	246.73	212.84	210.48	618.36	230.06	212.43	210.49
CPU	100%	317%	396%	330%	100%	403%	489%	490%	100%	407%	492%	513%	100%	402%	486%	492%	100%	438%	492%	495%
Speedup	1.00	2.12	2.55	2.17	1.00	2.54	2.92	2.91	1.00	2.51	2.92	3.01	1.00	2.51	2.90	2.94	1	2.69	2.91	2.94
omicron_ht_tlm1_sc_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	621.33	1,018.03	1,011.27	1,016.78	621.53	1,066.66	1,065.72	1,066.94	621.46	1,065.49	1,062.53	1,065.16	621.58	1,052.32	1,026.09	1,025.00	620.97	1047.39	1046.86	1047.7
Sys	1.31	1.01	0.99	1.05	1.24	1.11	1.15	1.09	1.24	1.08	1.14	1.10	1.19	1.22	1.10	1.14	1.17	0.98	1.03	0.94
Elapsed	622.45	204.63	205.66	208.08	622.57	187.23	186.69	187.68	622.51	187.30	186.47	187.20	622.57	186.82	202.18	207.74	622.02	213.09	212.43	212.81
CPU	100%	497%	492%	489%	100%	570%	571%	569%	100%	569%	570%	569%	100%	563%	508%	493%	100%	491%	493%	492%
Speedup	1.00	3.04	3.03	2.99	1.00	3.33	3.33	3.32	1.00	3.32	3.34	3.33	1.00	3.33	3.08	3.00	1	2.92	2.93	2.92
omicron_ht_tlm1_nb_max	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	621.31	624.63	624.85	635.53	621.35	644.87	674.31	685.36	621.52	954.05	1,009.74	1,036.59	621.26	955.79	986.26	1,030.52	621	999.04	1015.64	1029.79
Sys	2.04	1.61	1.72	1.60	1.93	2.12	2.12	2.23	1.97	1.71	1.79	1.67	2.06	1.89	1.72	1.69	2.04	1.93	1.78	1.77
Elapsed	623.16	625.19	625.38	635.97	623.09	645.09	674.41	685.51	623.30	266.84	226.43	212.93	623.12	255.32	230.27	209.96	622.93	231.37	220.9	211.02
CPU	100%	100%	100%	100%	100%	100%	100%	100%	100%	358%	446%	487%	100%	375%	429%	491%	100%	432%	460%	488%
Speedup	1.00	1.00	1.00	0.98	1.00	0.97	0.92	0.91	1.00	2.34	2.75	2.93	1.00	2.44	2.71	2.97	1	2.69	2.82	2.95
omicron_ht_tlm2_nil_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									626.96	1,097.06			626.74	1,018.80			626.7	1059.68		
Sys									1.47	1.95			1.51	2.23			1.48	1.25		
Elapsed									628.25	220.29			628.07	253.78			628	215.28		
CPU									100%	498%			100%	402%			100%	492%		
Speedup									1.00	2.85			1.00	2.47			1	2.91		
omicron_ht_tlm2_db_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									627.22	1,092.44			626.75	1,018.24	1,033.62	1,136.69	627.22	1061.99	1078.07	1003.92
Sys									1.46	2.55			1.44	2.55	2.12	2.47	1.48	1.24	1.57	1.63
Elapsed									628.53	240.68			628.03	276.56	250.40	218.64	628.55	217.06	217.3	178.14
CPU									100%	454%			100%	369%	413%	521%	100%	489%	496%	564%
Speedup									1.00	2.61			1.00	2.27	2.51	2.87	1	2.9	2.89	3.53

Table 12: Measurement results on 8-core host (‘omicron’, HT on)

	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
phi_tlm1_blk_min	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	913.81	990.71	922.07	1,006.85	913.42	975.12	995.18	1,001.43	912.87	1,010.71	999.08	1,009.74	913.35	1,000.79	989.01	990.46	912.58	926.94	932.96	931.58
Sys	35.61	42.69	42.29	42.50	35.63	39.83	41.10	43.19	35.63	41.56	40.32	42.53	35.69	40.03	40.76	41.69	35.93	42.24	40.54	38.59
Elapsed	949.83	365.08	274.79	334.44	949.48	265.87	210.10	210.56	948.99	280.89	209.14	201.75	949.43	270.42	207.24	207.98	949.02	266.66	194.77	193.94
CPU	99%	283%	350%	313%	99%	381%	493%	496%	99%	374%	496%	521%	99%	384%	496%	496%	99%	363%	499%	500%
Speedup	1.00	2.60	3.46	2.84	1.00	3.57	4.52	4.51	1.00	3.38	4.54	4.70	1.00	3.51	4.58	4.57	1.00	3.56	4.87	4.89
phi_tlm1_blk_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	909.88	963.94	918.38	982.29	910.04	947.57	954.58	959.41	910.13	967.67	956.76	962.93	909.69	960.18	952.36	952.88	909.30	923.00	929.81	927.68
Sys	30.06	35.20	34.90	35.76	29.95	35.86	35.74	36.15	29.67	36.25	33.39	33.23	30.07	36.16	35.54	33.65	30.19	34.97	34.79	32.90
Elapsed	940.49	345.62	269.68	327.27	940.55	253.31	198.67	196.37	940.33	255.86	196.66	193.62	940.39	248.06	195.32	194.74	940.12	229.04	193.63	193.08
CPU	99%	289%	353%	311%	99%	388%	498%	506%	99%	392%	503%	514%	99%	401%	505%	506%	99%	418%	498%	497%
Speedup	1.00	2.72	3.49	2.87	1.00	3.71	4.73	4.79	1.00	3.68	4.78	4.86	1.00	3.79	4.81	4.83	1.00	4.10	4.86	4.87
phi_tlm1_sc_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	914.03	923.94	921.60	924.86	913.94	934.61	932.36	934.25	913.77	937.37	936.55	938.14	914.30	934.04	934.75	936.03	913.64	926.92	927.29	926.36
Sys	30.91	36.58	33.09	36.37	31.20	36.54	37.55	37.73	31.32	38.32	38.98	39.44	30.94	36.57	38.17	39.35	31.16	36.12	38.78	38.20
Elapsed	945.36	190.69	189.44	191.53	945.56	178.83	178.86	178.63	945.54	179.33	179.37	179.64	945.65	203.81	183.45	201.74	945.24	197.02	197.93	194.90
CPU	99%	503%	503%	501%	99%	543%	542%	544%	99%	544%	543%	544%	99%	476%	530%	483%	99%	488%	488%	494%
Speedup	1.00	4.96	4.99	4.94	1.00	5.29	5.29	5.29	1.00	5.27	5.27	5.26	1.00	4.63	5.15	4.68	1.00	4.80	4.78	4.85
phi_tlm1_nb_max	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	914.97	922.75	922.09	937.92	914.79	949.77	991.12	1,006.49	915.41	946.37	943.40	951.34	915.08	945.07	940.29	943.26	914.65	926.89	926.81	923.79
Sys	26.75	25.39	26.22	25.68	26.67	26.84	27.42	27.22	26.64	27.51	28.05	24.67	26.82	28.50	27.82	29.65	26.80	31.09	28.04	26.33
Elapsed	942.20	947.62	947.55	963.10	941.96	974.97	1,016.86	1,032.04	942.55	289.54	231.26	207.42	942.39	276.22	225.26	202.45	941.93	231.79	220.32	200.29
CPU	99%	100%	100%	100%	99%	100%	100%	100%	99%	336%	420%	470%	99%	352%	429%	480%	99%	413%	433%	474%
Speedup	1.00	0.99	0.99	0.98	1.00	0.97	0.93	0.91	1.00	3.26	4.08	4.54	1.00	3.41	4.18	4.65	1.00	4.06	4.28	4.70
phi_tlm2_nil_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									923.25	1,069.39			923.43	1,061.29			923.17	941.30		
Sys									32.85	38.77			32.46	37.98			32.74	37.90		
Elapsed									956.45	234.60			956.24	293.31			956.28	198.31		
CPU									99%	472%			99%	374%			99%	493%		
Speedup									1.00	4.08			1.00	3.26			1.00	4.82		
phi_tlm2_db_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									922.84	1,127.78			923.41	1,114.25	1,095.29	1,156.67	923.13	942.78	959.11	957.86
Sys									32.74	37.30			32.48	35.60	40.90	39.68	32.81	38.74	38.56	37.46
Elapsed									955.90	305.70			956.25	341.83	317.93	230.76	956.28	198.66	199.31	170.44
CPU									99%	381%			99%	336%	357%	518%	99%	494%	500%	583%
Speedup									1.00	3.13			1.00	2.80	3.01	4.14	1.00	4.81	4.80	5.61

Table 13: Measurement results on 16-core host ('phi', HT off)

	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
phi_ht_tlm1_blk_min	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	911.53	1,219.94	1,182.63	1,266.90	911.55	1,168.40	1,172.84	1,160.68	911.94	1,188.08	1,193.80	1,212.90	911.33	1,171.70	1,167.93	1,161.55	912.78	1,174.97	1,264.60	1,259.13
Sys	35.80	50.93	52.84	50.02	35.56	48.41	47.56	47.37	35.77	49.04	47.75	47.61	35.86	46.02	46.12	44.92	36.33	53.70	50.38	50.82
Elapsed	947.80	353.27	275.53	359.65	947.50	275.81	214.72	219.24	948.18	285.54	220.52	210.01	947.59	276.92	216.94	215.02	949.56	276.04	198.40	197.17
CPU	99%	359%	448%	366%	99%	441%	568%	551%	99%	433%	562%	600%	99%	439%	559%	561%	99%	445%	662%	664%
Speedup	1.00	2.68	3.44	2.64	1.00	3.44	4.41	4.32	1.00	3.32	4.30	4.51	1.00	3.42	4.37	4.41	1.00	3.44	4.79	4.82
phi_ht_tlm1_blk_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	909.22	1,157.43	1,179.46	1,172.94	909.03	1,168.54	1,206.42	1,193.06	909.23	1,161.98	1,225.45	1,240.29	908.85	1,166.31	1,214.16	1,210.00	909.23	1,181.75	1,262.49	1,250.82
Sys	29.85	41.90	42.42	41.05	30.03	41.88	42.30	41.98	29.81	42.15	42.58	42.64	29.98	41.03	42.53	41.45	29.86	42.94	43.52	42.28
Elapsed	939.65	334.12	273.51	336.35	939.60	252.04	205.71	204.96	939.56	257.85	206.72	201.77	939.38	252.51	204.93	203.63	939.61	237.69	196.73	197.47
CPU	99%	358%	446%	360%	99%	480%	607%	602%	99%	466%	613%	635%	99%	478%	613%	614%	99%	515%	663%	654%
Speedup	1.00	2.81	3.44	2.79	1.00	3.73	4.57	4.58	1.00	3.64	4.55	4.66	1.00	3.72	4.58	4.61	1.00	3.95	4.78	4.76
phi_ht_tlm1_sc_mul	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	912.14	1,267.97	1,261.70	1,262.00	912.57	1,289.98	1,292.03	1,292.56	912.37	1,265.77	1,262.97	1,287.68	913.01	1,278.21	1,301.01	1,278.01	912.26	1,251.91	1,245.82	1,254.42
Sys	31.48	46.58	46.55	46.99	31.09	49.69	49.36	49.62	30.75	49.31	49.82	49.76	30.80	47.88	48.33	48.29	31.18	50.10	48.65	49.30
Elapsed	944.04	197.64	196.87	197.40	944.15	193.55	195.01	193.07	943.52	194.74	194.15	194.19	944.31	197.73	204.20	192.57	943.98	201.13	195.35	195.40
CPU	99%	665%	664%	663%	99%	692%	687%	695%	99%	675%	676%	688%	99%	670%	660%	688%	99%	647%	662%	667%
Speedup	1.00	4.78	4.80	4.78	1.00	4.88	4.84	4.89	1.00	4.85	4.86	4.86	1.00	4.78	4.62	4.90	1.00	4.69	4.83	4.83
phi_ht_tlm1_nb_max	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr	913.52	922.68	921.54	937.69	914.01	948.89	990.62	1,004.41	913.95	1,134.40	1,174.70	1,166.17	913.51	1,135.21	1,173.54	1,180.90	913.59	1,196.01	1,167.82	1,185.25
Sys	27.35	25.68	26.29	25.42	27.09	27.15	27.56	27.64	27.11	35.04	35.46	33.96	27.44	34.69	34.66	34.17	26.83	40.03	34.83	33.73
Elapsed	941.43	948.41	947.93	963.17	941.58	975.46	1,017.61	1,031.39	941.56	284.35	236.79	223.02	941.47	261.19	225.66	206.30	940.93	235.12	224.39	202.90
CPU	99%	99%	99%	99%	99%	100%	100%	100%	99%	411%	511%	538%	99%	447%	535%	588%	99%	525%	535%	600%
Speedup	1.00	0.99	0.99	0.98	1.00	0.97	0.93	0.91	1.00	3.31	3.98	4.22	1.00	3.60	4.17	4.56	1.00	4.00	4.19	4.64
phi_ht_tlm2_nil_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									921.72	1,203.35			921.37	1,132.58			920.71	1,252.86		
Sys									34.55	42.09			34.56	41.96			34.74	48.47		
Elapsed									956.67	246.41			956.32	290.94			955.81	203.29		
CPU									99%	505%			99%	403%			99%	640%		
Speedup									1.00	3.88			1.00	3.29			1.00	4.70		
phi_ht_tlm2_db_mul									SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
Usr									921.70	1,245.02			921.89	1,181.62	1,183.58	1,264.67	921.66	1,249.43	1,268.47	1,003.05
Sys									33.94	41.47			33.88	41.53	42.40	42.61	33.95	48.66	44.42	40.75
Elapsed									956.04	312.41			956.19	337.94	323.45	238.25	956.01	203.17	204.89	170.68
CPU									99%	411%			99%	361%	379%	548%	99%	638%	640%	611%
Speedup									1.00	3.06			1.00	2.83	2.96	4.01	1.00	4.71	4.67	5.60

Table 14: Measurement results on 32-core host ('phi', HT on)

Omicron

Elapsed time	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	621.74	293.50	252.83	291.65	622.42	255.08	213.34	213.10	622.44	256.73	211.37	229.22	622.05	260.88	210.47	211.17	622.39	254.92	220.14	218.53
tlm1_blk_mul	618.35	282.02	242.98	272.54	617.77	252.11	198.52	195.62	617.74	241.27	208.74	205.75	618.37	257.07	198.02	195.55	618.11	234.35	219.24	217.33
tlm1_sc_mul	621.60	216.92	217.18	215.92	622.16	180.03	181.27	180.80	622.22	239.29	238.70	244.04	621.81	209.76	213.27	197.24	622.20	219.86	219.61	219.92
tlm1_nb_max	622.82	624.39	624.43	634.99	622.67	644.61	676.32	686.41	622.64	259.93	218.67	211.51	622.80	263.09	221.89	209.65	623.04	231.45	226.15	217.14
tlm2_nil_mul									628.21	249.92			627.73	272.38			627.70	223.63		
tlm2_db_mul									628.21	250.61			628.69	282.91	263.46	229.37	628.15	223.81	225.52	178.33

Omicron HT

Elapsed time	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	622.99	298.75	253.89	291.46	622.83	247.55	214.49	222.64	622.89	253.79	212.13	206.27	623.09	252.19	214.11	215.72	622.50	255.23	212.18	211.90
tlm1_blk_mul	618.75	291.86	242.36	284.67	619.08	243.30	212.23	212.78	618.19	246.45	211.91	205.71	618.25	246.73	212.84	210.48	618.36	230.06	212.43	210.49
tlm1_sc_mul	622.45	204.63	205.66	208.08	622.57	187.23	186.69	187.68	622.51	187.30	186.47	187.20	622.57	186.82	202.18	207.74	622.02	213.09	212.43	212.81
tlm1_nb_max	623.16	625.19	625.38	635.97	623.09	645.09	674.41	685.51	623.30	266.84	226.43	212.93	623.12	255.32	230.27	209.96	622.93	231.37	220.90	211.02
tlm2_nil_mul									628.25	220.29			628.07	253.78			628.00	215.28		
tlm2_db_mul									628.53	240.68			628.03	276.56	250.40	218.64	628.55	217.06	217.30	178.14

Phi

Elapsed time	risc_v0.5.1				risc_v0.5.2				risc_v0.5.3				risc_v0.6.0				risc_v0.6.1			
	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	949.83	365.08	274.79	334.44	949.48	265.87	210.10	210.56	948.99	280.89	209.14	201.75	949.43	270.42	207.24	207.98	949.02	266.66	194.77	193.94
tlm1_blk_mul	940.49	345.62	269.68	327.27	940.55	253.31	198.67	196.37	940.33	255.86	196.66	193.62	940.39	248.06	195.32	194.74	940.12	229.04	193.63	193.08
tlm1_sc_mul	945.36	190.69	189.44	191.53	945.56	178.83	178.86	178.63	945.54	179.33	179.37	179.64	945.65	203.81	183.45	201.74	945.24	197.02	197.93	194.90
tlm1_nb_max	942.20	947.62	947.55	963.10	941.96	974.97	1,016.86	1,032.04	942.55	289.54	231.26	207.42	942.39	276.22	225.26	202.45	941.93	231.79	220.32	200.29
tlm2_nil_mul									956.45	234.60			956.24	293.31			956.28	198.31		
tlm2_db_mul									955.90	305.70			956.25	341.83	317.93	230.76	956.28	198.66	199.31	170.44

Phi HT

Elapsed time	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO	SEQ	SYN	NPD	OOO
tlm1_blk_min	947.80	353.27	275.53	359.65	947.50	275.81	214.72	219.24	948.18	285.54	220.52	210.01	947.59	276.92	216.94	215.02	949.56	276.04	198.40	197.17
tlm1_blk_mul	939.65	334.12	273.51	336.35	939.60	252.04	205.71	204.96	939.56	257.85	206.72	201.77	939.38	252.51	204.93	203.63	939.61	237.69	196.73	197.47
tlm1_sc_mul	944.04	197.64	196.87	197.40	944.15	193.55	195.01	193.07	943.52	194.74	194.15	194.19	944.31	197.73	204.20	192.57	943.98	201.13	195.35	195.40
tlm1_nb_max	941.43	948.41	947.93	963.17	941.58	975.46	1,017.61	1,031.39	941.56	284.35	236.79	223.02	941.47	261.19	225.66	206.30	940.93	235.12	224.39	202.90
tlm2_nil_mul									956.67	246.41			956.32	290.94			955.81	203.29		
tlm2_db_mul									956.04	312.41			956.19	337.94	323.45	238.25	956.01	203.17	204.89	170.68

Table 15: Summary of elapsed time on 4, 8, 16, 32 core hosts

B Visualization

Figure 11 shows a visualization of the TLM-1.0 SystemC model of GoogLeNet using visual tool, a graphical SystemC module visualizer using RISC [11]. As shown in the Figure 11, the stimulus module is placed at bottom left corner and the monitor is placed at top left corner. The majority of Figure 11 is DUT which is enclosed in the light blue rectangle. DUT comprises of 142 modules which are drawn in colored rectangles on the right side of the figure. Modules are connected to their neighboring modules by channels that are drawn in line segments.

Figure 12 shows a visualization of TLM-2.0 SystemC model of GoogLeNet using also visual tool. Memory is drawn as green box at the bottom part of DUT and each arc shows a socket connection between the memory and a module.

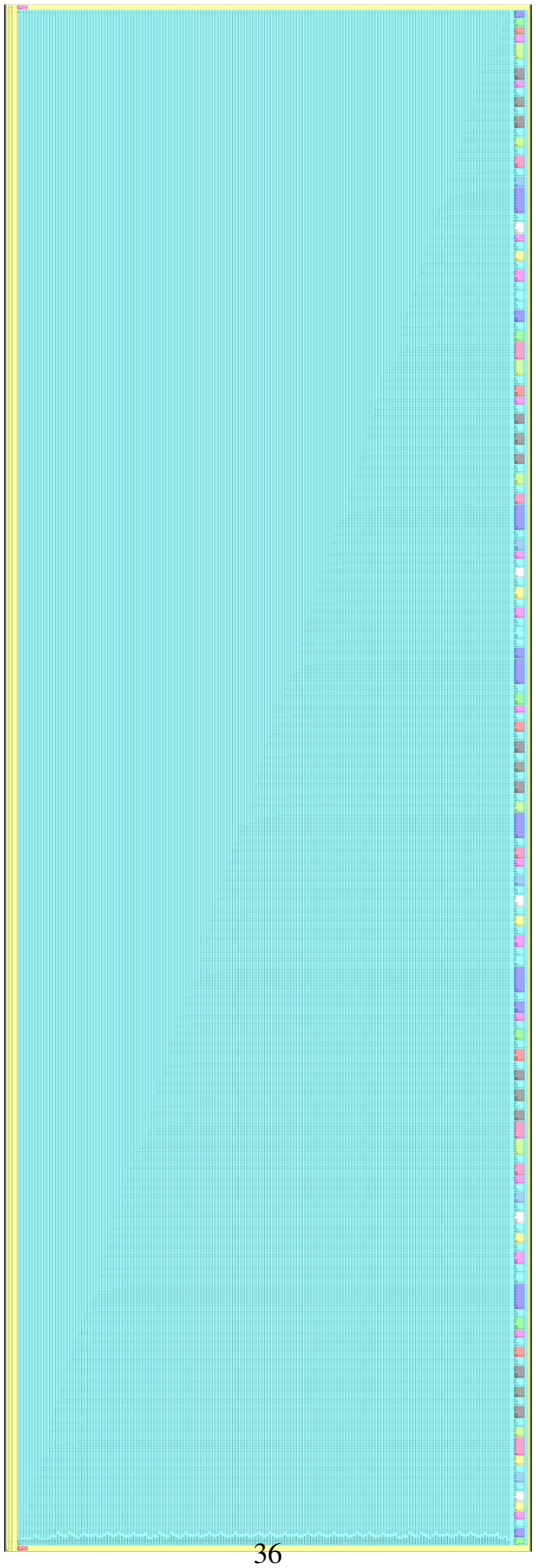

Figure 11: Visualized SystemC TLM-1.0 model of GoogLeNet generated by visual [11]

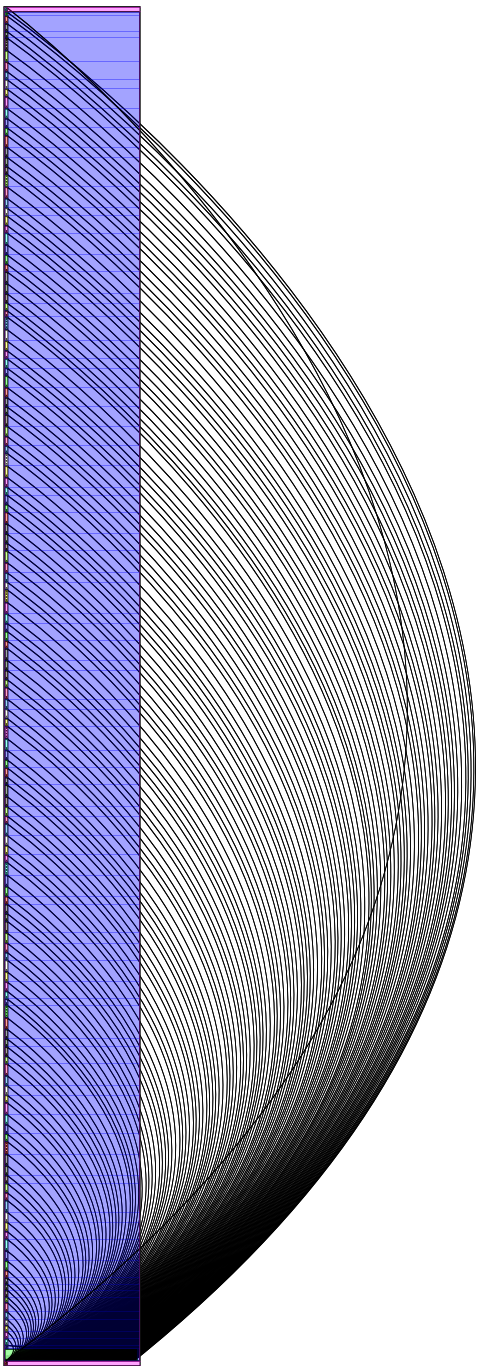

Figure 12: Visualized SystemC TLM-2.0 model of GoogLeNet generated by visual [11]