8-2023

Visualizing Transaction-Level Modeling Simulations of Deep Neural Networks

Nataniel Farzan
Chapman University, farzan@chapman.edu

Emad Arasteh
Chapman University, arasteh@chapman.edu

System Platform Exploration Lab

Visualizing Transaction-Level Modeling Simulations of Deep Neural Networks

Nataniel Farzan, Emad Arasteh

FSE-TR-23-01
August 14, 2023

System Platform Exploration Lab
Dale E. and Sarah Ann Fowler School of Engineering
Chapman University
Orange, California, USA

{farzan,arasteh}@chapman.edu
www.chapman.edu/engineering

Visualizing Transaction-Level Modeling Simulations of Deep Neural Networks

Nataniel Farzan, Emad Arasteh

FSE-TR-23-01
August 14, 2023

System Platform Exploration Lab
Dale E. and Sarah Ann Fowler School of Engineering
Chapman University

Abstract

The growing complexity of data-intensive software demands constant innovation in computer hardware design. Performance is a critical factor in rapidly evolving applications such as artificial intelligence (AI). Transaction-level modeling (TLM) is a valuable technique used to represent hardware and software behavior in a simulated environment. However, extracting actionable insights from TLM simulations is not a trivial task. We present Netmemvisual, an interactive, cross-platform visualization tool for exposing memory bottlenecks in TLM simulations. We demonstrate how Netmemvisual helps system designers rapidly analyze complex TLM simulations to find memory contention. We describe the project's current features, experimental results with two state-of-the-art deep neural networks (DNNs), and planned future work.

Contents

1 Introduction **1**
- 1.1 Deep Neural Networks (DNNs) for Computer Vision 2
- 1.2 Transaction Level Modeling (TLM) 4
- 1.3 Transaction Level Modeling of DNNs 4

2 Background **5**
- 2.1 Computer Systems 5
  - 2.1.1 Processor 5
  - 2.1.2 Memory 5
  - 2.1.3 Systems Software 6
- 2.2 Electronic System-level (ESL) Design 6
  - 2.2.1 System-Level Design Languages 7
- 2.3 Memory Bottlenecks 7

3 Simulation & Visualization **8**
- 3.1 Visualization of Hardware-level Simulations 8
- 3.2 Visualization of System-level Simulations 9

4 Memory Contention Visualization **9**
- 4.1 Requirements & Features 10
- 4.2 System Overview 10
- 4.3 Software Architecture 11
  - 4.3.1 Design Challenges 12
- 4.4 Graphical User Interface (GUI) 13
- 4.5 Contention Visualization 15

5 Results & Visualizations **15**
- 5.1 Model Generation 15
- 5.2 Simulation Setup 15
- 5.3 Experiment Setup 16
- 5.4 GoogLeNet Results 16
- 5.5 Single Shot MultiBox Detector (SSD) Results 18

6 Conclusion **19**
- 6.1 Future Work 19

A Appendix **22**
- A.1 Netmemvisual Test Case 22
- A.2 Trace Log File Syntax 22
- A.3 Visualization Configuration File Syntax 23

List of Figures

1 Transaction level modeling (TLM) simulation, visualization workflow 2
2 GoogLeNet network with all the bells and whistles [29] 3
3 SSD network structure generated with Netron [24] 3
4 TLM-2.0 LT model interconnect with a single shared memory [4] 4
5 Nonuniform memory access (NUMA) diagram 6
6 Basic model of stacking memory contention from request 0 to 3 . 7
7 Netmemvisual timing diagram structure 10
8 Netmemvisual visualization workflow 11
9 UML class diagram of Netmemvisual 11
10 Netmemvisual graphical user interface (GUI) overview 13
11 Netmemvisual GUI: Modules panel 13
12 Netmemvisual GUI: Subplots panel 14
13 Netmemvisual GUI: Parameters panel 14
14 GoogLeNet timing diagram without contention (image #1/1) . . 16
15 GoogLeNet timing diagram with contention (image #1/1) 17
16 GoogLeNet timing diagram with contention (image #100/100) . 17
17 SSD 'fc7' timing diagram with contention (image #1/1) 18
18 GoogLeNet timing diagram waterfall with contention (image #1/1) 22

List of Tables

1 Timing diagram legend 15

Listings

1 Truncated and annotated TLM simulation trace log file (SSD) . 22
2 Example JSON configuration file for SSD 'fc7' 24

Visualizing Transaction-Level Modeling Simulations of Deep Neural Networks

N. Farzan, E. Arasteh
System Platform Exploration Lab
Dale E. and Sarah Ann Fowler School of Engineering
Chapman University
Orange, California, USA
{farzan,arasteh}@chapman.edu

Abstract

The growing complexity of data-intensive software demands constant innovation in computer hardware design. Performance is a critical factor in rapidly evolving applications such as artificial intelligence (AI). Transaction-level modeling (TLM) is a valuable technique used to represent hardware and software behavior in a simulated environment. However, extracting actionable insights from TLM simulations is not a trivial task. We present Netmemvisual, an interactive, cross-platform visualization tool for exposing memory bottlenecks in TLM simulations. We demonstrate how Netmemvisual helps system designers rapidly analyze complex TLM simulations to find memory contention. We describe the project's current features, experimental results with two state-of-the-art deep neural networks (DNNs), and planned future work.

1 Introduction

Developments in computer system design are fueled by the ever-increasing complexity of software applications. System-level modeling strategies address the challenges brought about by complex computer system design. Modeling, simulation, and verification are powerful techniques system designers use to tackle the ever-increasing complexity of designing future computer systems.

The organization of main memory in a computer system can have significant performance implications. Memory bottlenecks are further exacerbated by data-intensive applications that require frequent data transactions to and from the processor(s). Thus, is important to identify performance problems early in the design process.

Data visualization is an effective method for presenting findings to a broader audience. Various tools exist, both at the hardware and system level, for visualizing simulated behavior and performance.

In this report, we present Netmemvisual, a visualization tool for plotting timing diagrams of transaction-level modeling (TLM) simulations. Our tool allows users to analyze memory accesses of deep neural networks (DNNs) at the system level. The purpose of Netmemvisual is to aid in the system design process by exposing memory contention at the transaction-level to the user.

The flowchart depicted in Figure 1 shows our proposed TLM simulation and visualization workflow. A SystemC model of a DNN is simulated using a SystemC simulator, which produces a trace log file containing transaction timings. Netmemvisual (blue box) analyzes the trace log file and produces a timing diagram of the simulated transactions for efficient design space exploration (DSE). Moreover, Netmemvisual provides a user-friendly graphical user interface (GUI) tailored to designing visualizations of DNNs.

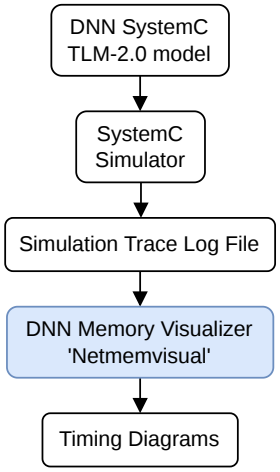

Figure 1: Transaction level modeling (TLM) simulation, visualization workflow

Section 1 briefly introduces deep neural networks (DNNs), the importance of transaction-level modeling (TLM), and TLM modeling of DNNs for efficient hardware/software co-design. In Section 2, we provide background knowledge on computer systems, electronic system-level (ESL) design, and the memory bottleneck problem. Related works and industry tools for simulation and visualization are discussed in Section 3. In Section 4, we describe Netmemvisual's design and feature set. Experimental results and analysis are presented in Section 5. We describe planned future work in Section 6.

1.1 Deep Neural Networks (DNNs) for Computer Vision

Large neural networks with many hidden layers are known as deep neural networks (DNNs). Increasing the depth of a network is a common strategy for improving accuracy. However, as the complexity of a network increases, computational resource requirements also increase [29]. Optimizing DNN performance

is essential when developing an efficient state-of-the-art DNN [29]. Additionally, targeting resource-constrained systems such as embedded computing devices further amplifies these design challenges [29].

Convolutional neural networks (CNNs) are a class of neural networks often used in the computer vision field. Common applications of CNNs include image classification and object detection [12]. Image classification is the identification of the main subject in an image based on a set of predefined classes, while object detection involves identifying one or more objects in an image and drawing a bounding box around the object(s) [12].

GoogLeNet and Single Shot MultiBox Detector (SSD) are two state-of-the-art deep CNNs used for advanced computer vision applications. GoogLeNet has proven to be a competitive network for accurate image classification [29]. When counting all the individual layers in the network, including those that constitute the 9 'inception' sections, GoogLeNet contains 142 distinct layers [4]. The most common layer types in the network include convolution, pooling, and concatenation layers. These inceptions serve to increase the depth of the network with a network in network (NIN) architecture [29] that does not increase computational complexity [12]. Figure 2 shows the structure of GoogLeNet.

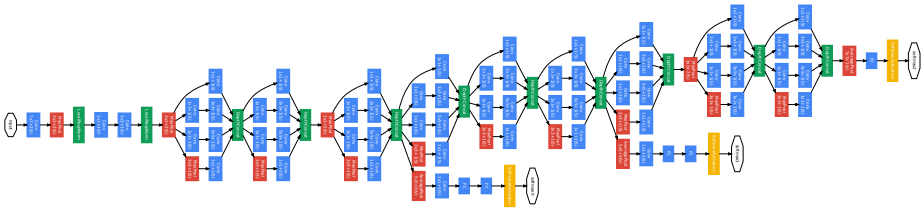

Figure 2: GoogLeNet network with all the bells and whistles [29]

Single Shot MultiBox Detector (SSD) is used for real-time object detection where it has been shown to outperform similar networks in both speed and accuracy [14]. The SSD network consists of 101 distinct layers, including convolution, flatten, and permute layers [5]. Figure 3 shows the structure of SSD.

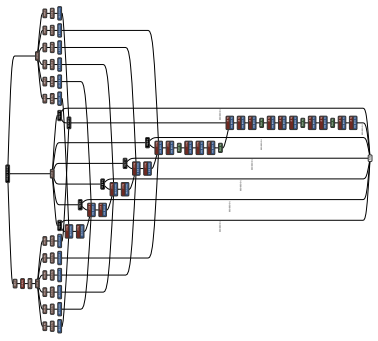

Figure 3: SSD network structure generated with Netron [24]

1.2 Transaction Level Modeling (TLM)

Transaction level modeling (TLM) aims to address the issue of hardware/software co-design. TLM allows for modeling, simulation, and verification early in the design process. System designers are able to quickly and cheaply simulate computer hardware and analyze application behavior. Compared to common application-level performance profiling tools, TLM simulation dips to a lower level of abstraction, which ultimately translates to much more accurate system description and analysis.

Different transaction-level models can be used to represent various abstraction levels, where there is an inverse relationship between simulation speed and accuracy [5]. TLM-1 allows for the modeling of basic communication between components via channels, whereas TLM-2.0 supports more accurate models with memory mapping techniques [5].

Virtual prototyping refers to the modeling and testing of hardware before it is actually produced [8]. This approach to product design is particularly useful in industries where physical prototypes are expensive to produce, such as computer hardware. TLM enables system-level virtual prototyping where system designers can experiment with various hardware and software designs in a simulated environment [1].

Work has also been done to automatically generate transaction-level models from abstract design descriptions [22] [4]. This makes it possible to rapidly prototype system designs in an accurate and user-friendly manner.

1.3 Transaction Level Modeling of DNNs

As briefly mentioned in Section 1.2, several TLM simulation coding styles exist, each at their own abstraction level. A loosely-timed (LT) model translates to a relatively fast simulation that reveals basic process ordering in a system. Approximately-timed (AT) models process timing information more accurately, but usually with significantly slower simulation. Another approach, loosely-timed contention-aware (LT-CA) modeling, redefines the established speed-accuracy trade-off with both the fast simulation speed of LT and the accurate memory contention of AT [4].

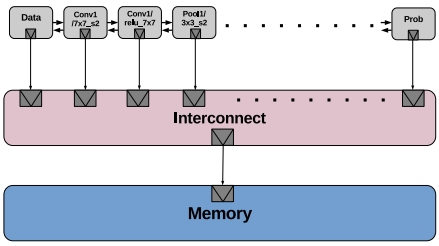

Figure 4: TLM-2.0 LT model interconnect with a single shared memory [4]

One possible TLM design of a neural network models each DNN layer as a processor, which corresponds to a SystemC module. An example of this strategy is shown in Figure 4. Each module has a TLM-2.0 initiator socket that connects to a target socket on a global interconnect. The interconnect connects to shared memory, which stores the input and output data required for processing each layer [4].

2 Background

This section provides a brief background on computer system components, system-level modeling, and the design challenges presented by memory bottlenecks.

2.1 Computer Systems

There are several essential hardware components and operating principles that are shared by virtually all computers. In order to study their behavior and design, we must first understand how these components have evolved to interact with one another.

2.1.1 Processor

A computer's processor is responsible for executing program instructions. Over the past couple decades, computers have evolved from housing just a single processing unit to multiple processors ('cores') in a single integrated circuit (IC). This enables parallel processing and better multithreading performance [21]. Designing complex multi-core and many-core processors, and software that takes full advantage of their capabilities, brings many challenges.

2.1.2 Memory

Program instructions and associated data are stored in memory, which is accessible to the processor. The processor can access memory through a read or write operation. Different technologies exist on a memory hierarchy, where there is a trade-off between access time and storage size. Instructions and data may be stored in cache memory or in main memory [21].

Multi-core systems commonly feature a shared memory multiprocessor (SMP) design, where all cores share the same main memory. When designing complex multi-core systems, there are two common approaches for sharing a single address space. SMP architectures can be implemented with either a uniform memory access (UMA) or nonuniform memory access (NUMA) strategy. The former approach means that all processors share the same access time to all main memory space. The latter takes a more nuanced approach in which access times for a section of memory can vary greatly depending on which processor initiates the request. Systems with multiple processors and memory channels often utilize a NUMA approach in which a processor requesting access to its

local memory is faster than accessing memory local to another processor [21]. These shared memory architectures bring challenges in designing performant software that utilizes memory in an optimal fashion.

Figure 5 depicts a NUMA memory organization strategy in which 'Processor 0' can access its local main memory (purple path) and main memory local to another processor (orange path). There is higher latency with the non-local memory access because 'Processor 0' has to access 'Main Memory 1' through the memory controller on 'Processor 1'.

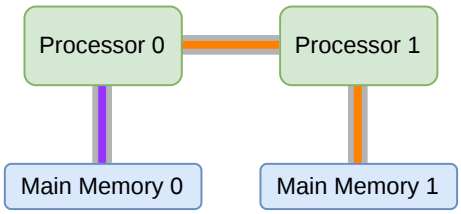

Figure 5: Nonuniform memory access (NUMA) diagram

2.1.3 Systems Software

Among the various abstraction levels in computer hardware and software, systems software lies above the hardware and below the applications software [21]. The implementation of systems software has an effect on overall system performance. Software can be examined at various abstraction levels, from data structures and algorithms implemented in source code to the machine code generated by the compiler infrastructure [21]. There are many approaches for improving software performance, including data level parallelism, instruction level parallelism, memory hierarchy optimization, and thread level parallelism [21].

2.2 Electronic System-level (ESL) Design

In the early days of computer design, hardware and software components were designed independently [9]. However, over the past several decades, the complexity of computer hardware has increased exponentially. According to Gordon Moore's 1965 predictions, later known as Moore's Law, the number of transistors on a chip has doubled roughly every 18 months [19]. The field of system design emerged to meet this drastic increase in design complexity.

System designers use techniques such as modeling, simulation, and verification [9]. These approaches represent a higher level of abstraction, where system models can be iterated upon using the specify-explore-refine methodology [9].

2.2.1 System-Level Design Languages

A System-Level Design Language (SLDL) is used to formally specify hardware and software structure and behavior in a system.

SpecC is a superset of ANSI-C which separates computation and communication logic for a turnkey system-level description solution. A SpecC program consists of a set of behaviors, channels, and interfaces. Given that the SpecC language is an extension of ANSI-C, existing C programs do not require a full rewrite to take advantage of system-level design principles [10].

SystemC is the IEEE standard SLDL, making it a good choice for standards-compliant system design, modeling, and verification [1]. The language is implemented as a C++ class library, meaning existing code can be augmented to benefit from system-level design principles. The reference implementation of the SystemC class library [3] includes a simulation kernel for running transaction-level modeling (TLM) simulations.

2.3 Memory Bottlenecks

A memory bottleneck occurs when a more than one processor ('core') attempts to access memory (read or write) at the same time. The result is inefficiency in a program in which one or more processors sit(s) idle while another finishes its task(s). This ultimately leads to wasted time and power. Therefore, one of the goals of performant software is minimizing memory bottlenecks.

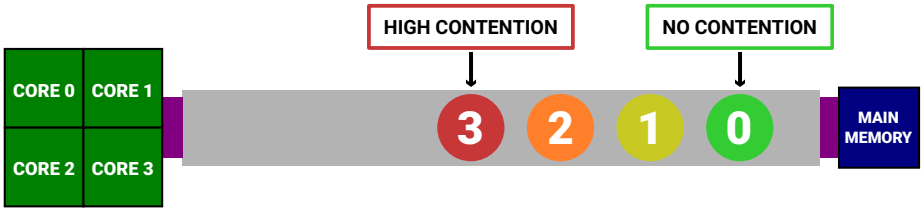

Figure 6: Basic model of stacking memory contention from request 0 to 3

Memory contention describes the amount of time a core must wait before accessing memory. A rudimentary model of stacking memory contention is depicted in Figure 6. In this example, four cores send simultaneous requests to access shared memory. For the sake of simplicity, we will assume that core 0 sent request 0, core 1 sent request 1, and so on. The result is memory contention stacking between each request. Request 0 experiences no contention and is able to access memory immediately, while request 3 experiences high contention as it waits for all of the previous requests to finish. This bottleneck may be further exacerbated depending on the significance of each memory access to the current software application. If the process that core 0 is working on relies on the completion of core 3's task to proceed, part of the system may sit idle for a significant amount of time. Of course, actual simulated behavior is more

complex and nuanced, but the underlying principles from this simple example still apply.

Various hardware and software strategies can be used to alleviate memory bottlenecks. In the case of software prefetching, the system attempts to predict areas of memory that will be needed by a future computation [21]. This can reduce the number of concurrent memory access requests, thereby reducing memory bottlenecks. Another approach, known as memory affinity, involves a NUMA arrangement where data is organized with the intent to maximize local memory accesses. [21]. Despite these mitigation strategies, the elimination of memory bottlenecks still presents a grand challenges in computer system design. Furthermore, the early detection of memory performance problems is valuable in the system design life cycle.

3 Simulation & Visualization

Visualization is a powerful tool used in a wide variety of disciplines. In the context of scientific research, visualization provides two unique benefits: presentation and exploration [31]. Presentation allows complex information to be shared in a digestible manner. For example, a graph might condense hundreds or thousands of rows in a spreadsheet into one simple figure that clearly displays a trend in the data.

Exploration enables researchers to examine their findings from different perspectives to gain new insights. Good visualizations communicate complex information in an efficient manner [31]. They provide a layer of abstraction above the raw data to aid in the interpretation of the information.

3.1 Visualization of Hardware-level Simulations

In this section, we provide a brief overview of visualization tools that exist for hardware-level simulation.

Computers have grown exponentially in complexity over the past several decades [19]. Various tools for designing computer hardware have been developed to meet this demand. As a result, there exists a hierarchy of abstraction levels in the hardware design space [13]. This systematic approach exposes only the most relevant design details at each phase in the process.

Individual transistors are considered at the transistor level, while groups of transistors that form logic gates are used at the gate level [13]. These techniques alone are can be used to construct very simple digital circuits. For complex microprocessors, however, higher level design tools are necessary. At the register-transfer level, system designers consider where data is stored and transported between areas known as registers [13]. At the behavioral level, hardware description languages (HDLs) such as Verilog and VHDL are used to define the circuit's high-level operating behavior [13].

A common form of visualization used in the hardware design field is the waveform viewer [13]. Waveform visualizers plot analog or digital signals over

time, allowing for extremely low level analysis of a system. GTKWave is an open-source tool for viewing HDL simulations in various formats [2]. There is also a wide range of commercial simulation waveform visualizer tools such as Verdi from Synopsys [28], the Questa Visualizer from Siemens [25], and Virisium Debug from Cadance [7].

3.2 Visualization of System-level Simulations

This section briefly describes some of the existing system-level simulation and visualization tools that are related to our work.

The proliferation of several computer system architectures has brought about the development of architecture simulators such as gem5 [15]. The gem5 simulator can simulate various computer architectures as well as other common components such as memory [15]. Using the gem5 simulation engine, applications and operating systems interact with simulated hardware to provide statistics at the cycle level [15].

There has been development towards improving interoperability between the gem5 simulator and SystemC transaction-level models [18]. This allows gem5 simulations to utilize external TLM modules such as DRAMSys to provide enhanced design space exploration opportunities [26].

The SycView project provides timing diagram visualization and performance statistics for loosely-timed SystemC simulations [6]. Key simulation events are recorded in a trace file which is then loaded into a graphical visualization tool to display a timing diagram of all the events and platform statistics [6].

Impulse VP includes a SystemC TLM-2.0 debugger and transaction flow visualizer that supports various views and plot types [30]. It reads SystemC transaction traces from a CSV-like database file for analysis and visualization [30].

Visualization tools exist for virtual prototyping [27], examining memory behavior [26], and signal analysis [16]. However, to the best of our knowledge, we are not aware of any software built specifically for visualization at the transaction-level that is focused on memory contention in deep neural networks.

4 Memory Contention Visualization

Given the growing complexity of deep neural networks, and their prevalence on resource-constrained systems, evaluating performance is a necessary step in the design process. Memory contention visualization is useful for identifying performance bottlenecks in a system. Our tool, Netmemvisual, enables memory contention visualization at the system level.

Figure 7 displays the general structure of timing diagrams generated by Netmemvisual. Each module (purple) represents a layer in the deep neural network. Modules are arranged into horizontal collections called tracks (blue). Ideally, these module operations occur in sequence, as they will be plotted one after another in respect to the horizontal axis (simulated time). A subplot (green)

groups one or more related tracks together along the vertical axis. A collection of one or more subplots can be plotted as a timing diagram (grey). Dividing groups of tracks into many subplots may be necessary for long simulations, where each subplot represents a small portion of the total simulated time.

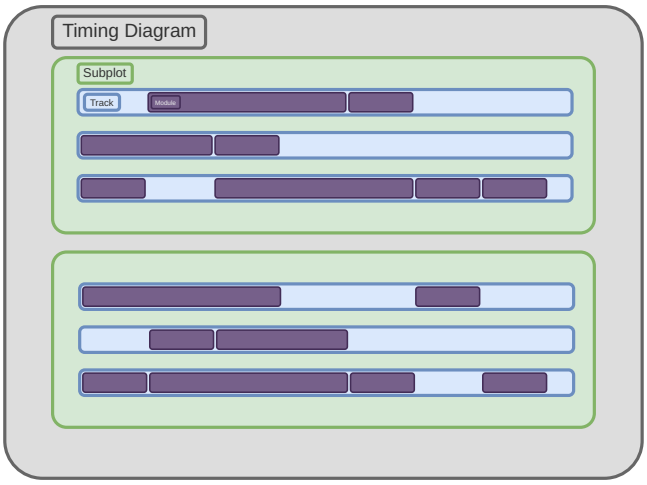

Figure 7: Netmemvisual timing diagram structure

4.1 Requirements & Features

The goal of Netmemvisual is to aid in the analysis of DNN performance by exposing memory bottlenecks. Our tool allows system designers to easily identify performance bottlenecks early in the design process. We intend to aid in the system platform exploration workflow in which different design approaches can be quickly evaluated and iterated upon.

Netmemvisual supports a command-line interface (CLI) and a graphical user interface (GUI). The CLI can be used for quick timing diagram generation and automated batch processing. The GUI presents a more user-friendly interface intended for exploring visualization techniques and configurations. The GUI allows the user generate a timing diagram based on the currently specified parameters and optionally export a `JSON` configuration file for future use. Previously saved configuration files can be imported for further editing.

4.2 System Overview

First, the user supplies a simulation trace log file. The log file is parsed to display a list of modules to the user. The user arranges these modules into tracks and subplots using the graphical user interface. The configuration file is parsed to determine the layout of the timing diagrams as well as various simulation parameters. Finally, a timing diagram of the simulation is generated

using the user-specified configuration and it is displayed to the user. Figure 8 shows a block diagram of this workflow.

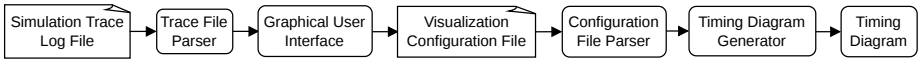

Figure 8: Netmemvisual visualization workflow

4.3 Software Architecture

This section describes the software architecture of Netmemvisual. We provide a class diagram and a brief description about the purpose of each class.

Class diagrams are useful for determining the high-level structure and functions of classes in a software project. The Unified Modeling Language (UML) is the ISO standard for visual modeling applications [20]. Figure 9 shows a UML class diagram of Netmemvisual.

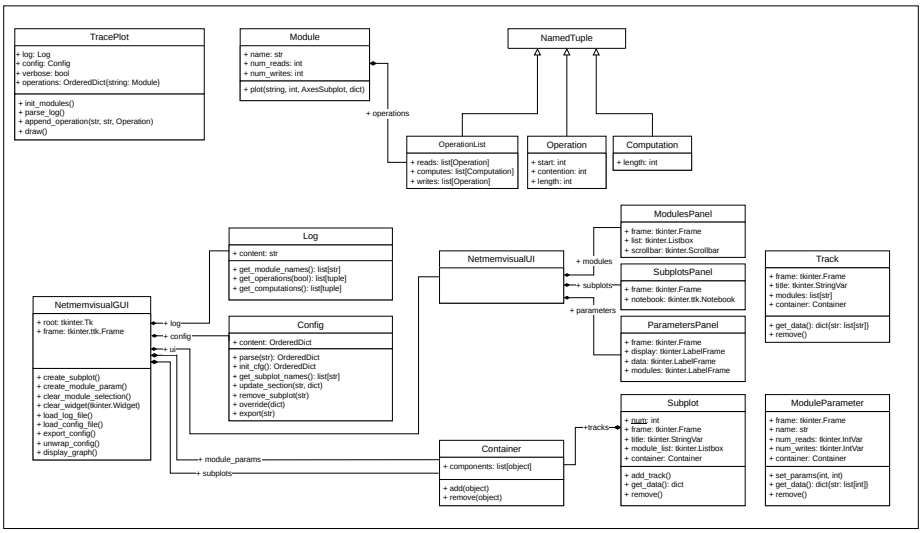

Figure 9: UML class diagram of Netmemvisual

The purpose of each class in the UML class diagram is briefly summarized below.

- `TracePlot`: Backend for tracing log files and plotting timing diagrams
- `Log`: Represents a simulation trace log file
- `Config`: Represents a timing diagram visualization configuration
- `Module`: Stores timing data and parameters for each module found in the log file

- `OperationsList`: Stores a list of memory accesses (reads and writes) and processor computations for a given module
- `Operation`: Stores starting times and lengths of memory accesses as well as the amount of contention experienced (if enabled)
- `Computation`: Stores the length of processor computations
- `NetmemvisualGUI`: Responsible for presenting a graphical interface to the user
- `NetmemvisualUI`: Organizes the various user interface panels
- `ModulesPanel`: Organizes the user interface widgets in the modules panel
- `SubplotsPanel`: Organizes the user interface widgets in the subplots panel
- `Parameters`: Organizes the user interface widgets in the parameters panel
- `Container`: A wrapper class to interact with the `tkinter` application programming interface (API)
- `Subplot`: Stores a collection of `Track` objects
- `Track`: Groups sequential modules together
- `ModuleParameter`: Specifies the number of memory reads and writes per image for a module

Both the command-line interface (CLI) and graphical user interface (GUI) versions of the tool share the same backend (`TracePlot`) for timing diagram generation. The Netmemvisual GUI relies on the `tkinter` module, which provides a Python interface to the cross-platform Tcl/Tk GUI toolkit [23]. We use the `matplotlib` visualization library to display timing diagrams [17]. The library supports a wide variety of configuration options, which allows us to experiment with various visualization styles and techniques.

4.3.1 Design Challenges

This section describes some of the software design challenges we encountered while developing Netmemvisual.

Interacting with the `tkinter` application programming interface (API) brought about difficulties in properly managing application state. Specifically, we had trouble ensuring that when graphical widgets were destroyed, their underlying application data was also disposed of. This caused discrepancies between what was displayed to the user and the internal visualization configuration that was stored in the application and later exported. We implemented the 'Composite' design pattern, which was created to address issues where both individual objects and compositions of objects must be treated the same externally [11].

4.4 Graphical User Interface (GUI)

The graphical user interface (GUI) provides a user-friendly and approachable interface for interacting with Netmemvisual. The GUI consists of three distinct panels for configuring visualizations: (1) Modules, (2) Subplots, (3) Parameters.

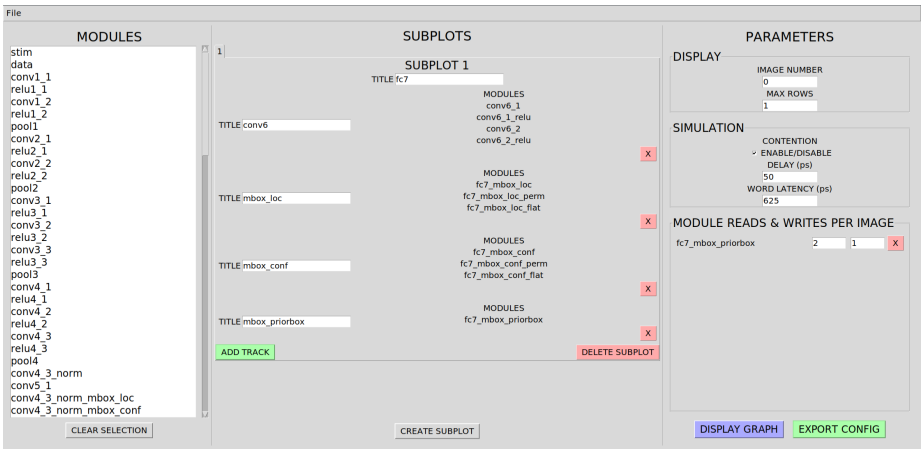

Figure 10: Netmemvisual graphical user interface (GUI) overview

Once a simulation trace log file has been loaded into the GUI, the 'Modules' panel populates with a list of all modules found in the log file. One or more modules can be selected at a time to be used in the next panel.

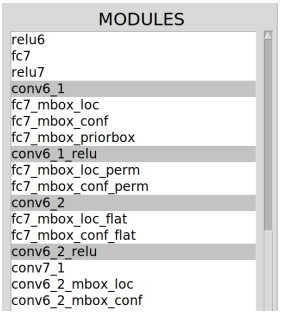

Figure 11: Netmemvisual GUI: Modules panel

The 'Subplots' panel is used to configure the arrangement of the selected modules in the diagram. Once the desired module(s) are selected in the 'Modules' panel, they can be added to a new track in the 'Subplots' panel. Each subplot is displayed in its own tab for easy navigation.

The 'Parameters' panel contains three subpanels: (1) Display, (2) Simulation, (3) Module Reads & Writes Per Image. The first section specifies the image number to display and the maximum number of subplot rows that will

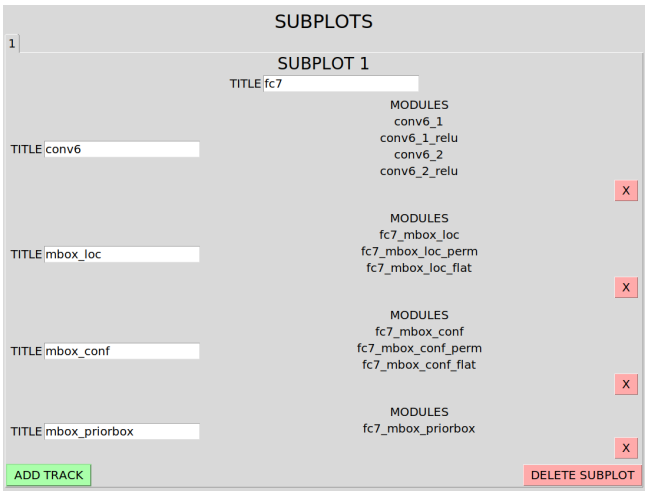

Figure 12: Netmemvisual GUI: Subplots panel

be used before a new column is created. The second section allows the user to configure simulation parameters such as contention-aware modeling and memory latency parameters. The third section is used to specify the number of reads (inputs) and writes (outputs) per image for a given module. Modules that are not configured here will have a default of 1 read and 1 write per image.

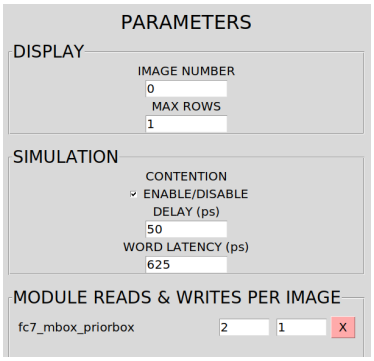

Figure 13: Netmemvisual GUI: Parameters panel

A visualization based on the current configuration can be previewed throughout the diagram building process. The configuration can be exported as a `JSON` file for future use. An existing configuration file can also be loaded into the GUI for further editing.

4.5 Contention Visualization

We run TLM simulations in the loosely-timed (LT) style, which can be useful for determining relative module timing and ordering. However, without memory contention information, the visualization does not convey the network's simulated performance. In order to gain a more detailed understanding of a DNN's performance bottlenecks, we must simulate detailed TLM-2.0 approximately-timed or cycle-accurate RTL. Alternatively, we can use the TLM-2.0 loosely-timed contention-aware (LT-CA) modeling style for simulation speed similar to loosely-timed (LT) models, and the memory contention accuracy of approximately-timed (AT) models [4]. These simulations model hardware/software interactions much more accurately and can be used to identify memory bottlenecks.

Table 1 defines the colors used to represent various TLM operations in the following timing diagrams. Memory contention is only plotted if the simulation supports it. Simulated time in milliseconds (ms) is plotted along horizontal axis, while module tracks are plotted on the vertical axis.

Operation Type	**Color**
Memory read	light green
Processor computation	dark green
Memory write	blue
Memory contention	red

Table 1: Timing diagram legend

5 Results & Visualizations

In this section, we present experimental results for Netmemvisual. We provide platform setup information and visualizations of two state-of-the-art deep neural networks. We also provide some analysis of the generated timing diagrams.

5.1 Model Generation

We utilize `Netspec` [4], a tool for automatically generating SystemC code for DNNs built with the `Caffe` deep learning framework. `Netspec` allows us to quickly generate TLM simulations with a variety of different parameters for testing with Netmemvisual. However, Netmemvisual is designed to accept trace log files with a relatively generic syntax, which is not exclusive to SystemC models generated by `Netspec`.

5.2 Simulation Setup

We use `SystemC 2.3.4-Accellera` for model simulation and `OpenCV 3.4.1` for high performance computer vision functions.

5.3 Experiment Setup

We explore visualization techniques by running several simulations of two different networks with varying parameters. We visualize different sections of the network with a focus on areas with high parallelism. Results are validated by confirming the generated timing diagrams reflect the intended transactions in the trace log files.

5.4 GoogLeNet Results

The structure of the GoogLeNet network is quite complex, meaning there are quite a few areas of interest for performance analysis. The parts of the network we choose to study are those with high degrees of parallelism, as these sections are likely to encounter high levels of memory contention as a result. There are nine portions of the network, known as 'inceptions', in which several modules work in parallel. This can be seen in Figure 2 where the network is three to four modules wide. These inceptions happen sequentially ('inception_3a', 'inception_3b', 'inception_4a', 'inception_4b', etc.), with some processing that happens in between each inception that is not included in our diagrams.

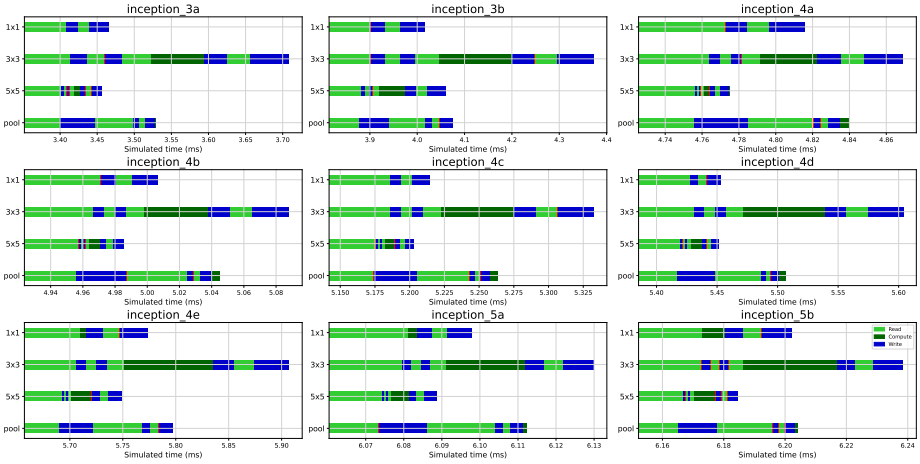

Figure 14: GoogLeNet timing diagram without contention (image #1/1)

A timing diagram of a GoogLeNet TLM-2.0 LT simulation that does not model memory contention is shown in Figure 14. Without contention data, our visualization does not show where the network experiences performance bottlenecks. However, contention-aware TLM visualizations reveal the extensive memory contention experienced in the highly parallel inceptions of this network.

A contention-aware TLM simulation of GoogLeNet (TLM-2.0 LT-CA), depicted in Figure 15, shows a much more accurate representation of its performance. This diagram reveals that every track experiences a significant amount of memory contention. A clear trend across all nine of the plotted inceptions is

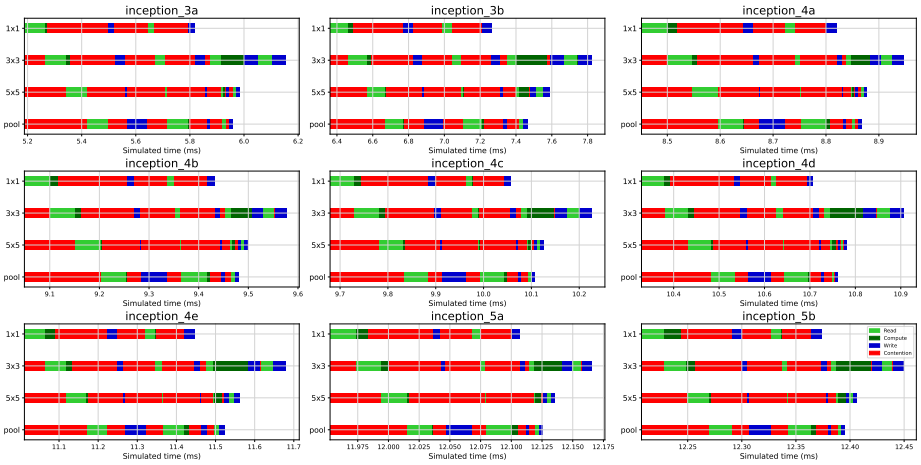

Figure 15: GoogLeNet timing diagram with contention (image #1/1)

stacking memory contention at the start of the subplot. Due to the underlying SystemC implementation of the network, track '1x1' always starts its memory read first, followed by '3x3', '5x5', and 'pool'.

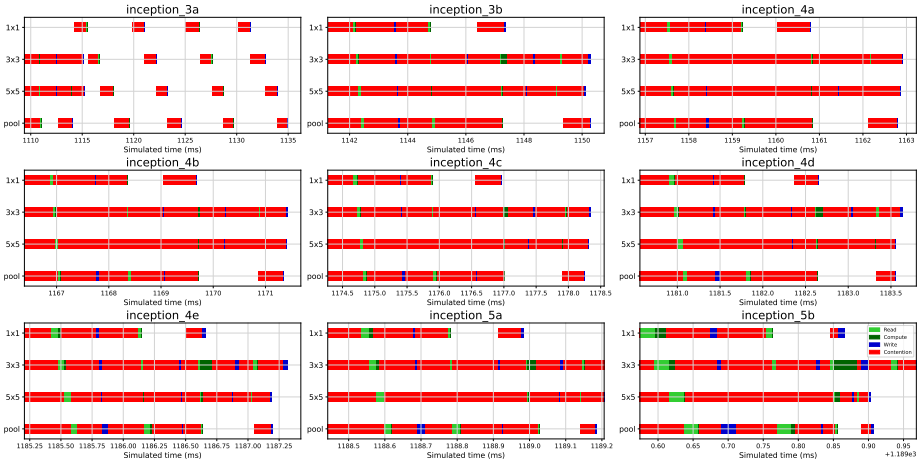

Figure 16: GoogLeNet timing diagram with contention (image #100/100)

Figures 14 and 15 are visualizations of GoogLeNet processing one image. Simulations of many images help us understand the long-term behavior characteristics of a DNN. Figure 16 is a timing diagram of the last image processed in a sequence of 100 images. By this point in the simulation, the model is experiencing extremely high levels of memory contention. Due to the high amounts of memory contention, it is difficult to see many of the read, compute, and write

operations in the diagram. A unique feature of this diagram is the presence of several empty gaps between operations. More investigation is necessary to determine the cause of these gaps.

The critical path of a network is important to consider when evaluating performance. In our timing diagrams of GoogLeNet, the critical path of every inception is the longest horizontal bar on each subplot. This is significant because it is this track that the rest of the system must wait on before finishing the inception. If a track becomes the critical path because of severe memory contention, that might signal to system designers where performance should be optimized.

5.5 Single Shot MultiBox Detector (SSD) Results

Using `Netspec`, we generate SystemC model files for SSD and simulate object detection. We present our visualization result for an interesting part of the network in which several modules execute in parallel.

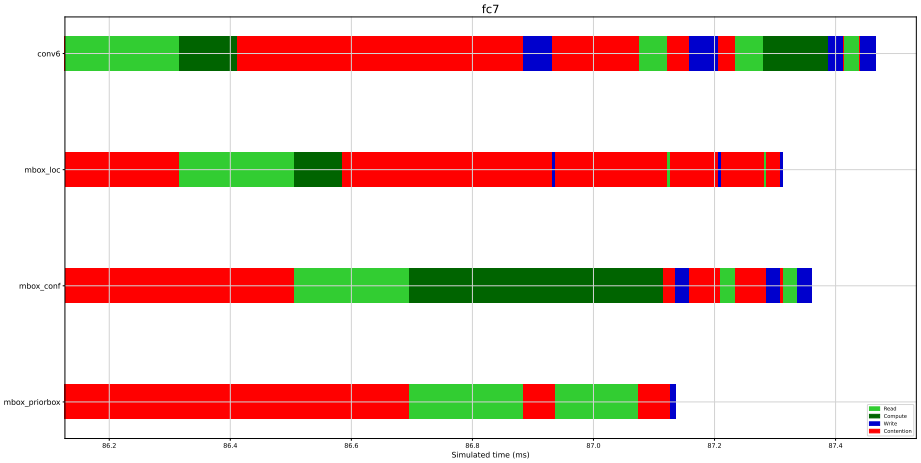

Figure 17: SSD 'fc7' timing diagram with contention (image #1/1)

The section of the network shown in Figure 17 is titled 'fc7'. Our visualization reveals that SSD experiences significant memory contention in this part of the network. A notable feature of this timing diagram is the cascade of memory read start times as we look down the diagram. The first track, 'conv6' starts a memory read immediately. The second track, 'mbox_loc' must wait until 'conv6' finishes accessing memory before it can start its read. This trend continues with each subsequent track in 'fc7', meaning that 'mbox_priorbox' experiences the most memory contention before its first memory read. This memory contention pattern seems to reflect the basic memory contention stacking principle described in Figure 6.

6 Conclusion

The ever-growing complexity of hardware and software drives the need for advanced system-level modeling and analysis. Accurate deep neural networks (DNNs) require significant computational resources and memory bandwidth. Resource-constrained platforms, such as embedded systems, present many challenges when designing complex computer systems.

Various transaction-level modeling (TLM) styles have been developed to model and simulate hardware at different abstraction levels. TLM enables system platform exploration and the identification of memory bottlenecks early in the design process.

Netmemvisual is a visualization tool for plotting timing diagrams of loosely-timed TLM simulations. It supports a command-line interface (CLI) for quick visualizations and a graphical user interface (GUI) to help users configure timing diagrams. Netmemvisual is useful for quickly evaluating performance bottlenecks such as excessive memory contention. This information can be used during the computer system design and optimization process.

We study two state-of-the-art neural networks for memory contention visualization: (1) GoogLeNet and (2) Single Shot MultiBox Detector. Experimental results show significant amounts of memory contention in these two deep convolutional neural networks.

6.1 Future Work

In the future, we plan to run more experiments with different DNNs for contention visualization and performance analysis. In addition, we would like to expand Netmemvisual's functionality using automation and statistical analysis with the goal of creating a TLM simulation performance profiler.

References

[1] Ieee standard for standard systemc language reference manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

[2] GTKWave. https://github.com/gtkwave/gtkwave, 2023.

[3] Accellera Systems Initiative. Systemc reference implementation. https://github.com/accellera-official/systemc, 2023.

[4] Emad M. Arasteh and Rainer Dömer. Fast loosely-timed deep neural network models with accurate memory contention. *ACM Trans. Embed. Comput. Syst.*, July 2023.

[5] Emad Malekzadeh Arasteh. *Transaction-Level Modeling of Deep Neural Networks for Efficient Parallelism and Memory Accuracy.* Ph.D. Dissertation, UC Irvine, Irvine, CA, USA, 2022.

[6] Denis Becker, Matthieu Moy, and Jérôme Cornet. SycView: Visualize and Profile SystemC Simulations. In *3rd Workshop on Design Automation for Understanding Hardware Designs, DUHDe 2016*, Dresden, Germany, Mar 2016.

[7] Cadence Design Systems, Inc. Verisium debug. https://www.cadence.com/en_US/home/tools/system-design-and-verification/ai-driven-verification/verisium-debug.html, 2023.

[8] S.H. Choi and A.M.M. Chan. A virtual prototyping system for rapid product development. *Computer-Aided Design*, 36(5):401–412, 2004.

[9] Daniel Gajski, Andreas Gerstlauer, Samar Abdi, and Gunar Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer US, 09 2009.

[10] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer, 2000.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[12] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–377, 2018.

[13] E.O. Hwang. *Digital Logic and Microprocessor Design with VHDL*. Electrical engineering handbook series. Thomson/Nelson, 2006.

[14] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot MultiBox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016.

[15] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. The gem5 simulator: Version 20.0+, 2020.

[16] MathWorks. Signal analyzer. https://www.mathworks.com/help/signal/ref/signalanalyzer-app.html, 2023.

[17] Matplotlib. Matplotlib: Visualization with python. https://matplotlib.org/, 2023.

[18] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and systemc: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69, 2017.

[19] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[20] Object Management Group. Unified modeling language specification. https://www.omg.org/spec/UML, 2023.

[21] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[22] Junyu Peng, Andreas Gerstlauer, Daniel D. Gajski, Rainer Dömer, and Dongwan Shin. Automatic generation of transaction level models for rapid design space exploration. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 64–69, 2006.

[23] Python Software Foundation. tkinter — python interface to tcl/tk. https://docs.python.org/3/library/tkinter.html, 2023.

[24] Lutz Roeder. Netron. https://netron.app/, 2023.

[25] Siemens. Questa visualizer debug environment. https://eda.sw.siemens.com/en-US/ic/debug-coverage/visualizer-debug/, 2023.

[26] Lukas Steiner, Matthias Jung, Felipe S Prado, Kirill Bykov, and Norbert Wehn. Dramsys4.0: A fast and cycle-accurate systemc/tlm-based dram simulator. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20*, pages 110–126. Springer, 2020.

[27] Synopsys. Platform architect. https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html, 2023.

[28] Synopsys. Verdi automated debug system. https://www.synopsys.com/verification/debug/verdi.html, 2023.

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

[30] toem GmbH. impulse vp. https://www.toem.de/index.php/products/impulse-vp, 2023.

[31] J.J. van Wijk. The value of visualization. In *VIS 05. IEEE Visualization, 2005.*, pages 79–86, 2005.

A Appendix

A.1 Netmemvisual Test Case

Figure 18 depicts a waterfall timing diagram of the entire GoogLeNet network. It serves as a stress test for Netmemvisual, as we plot the entire network structure, rather than specific sections of interest. The configuration file for this diagram was created manually using a text editor.

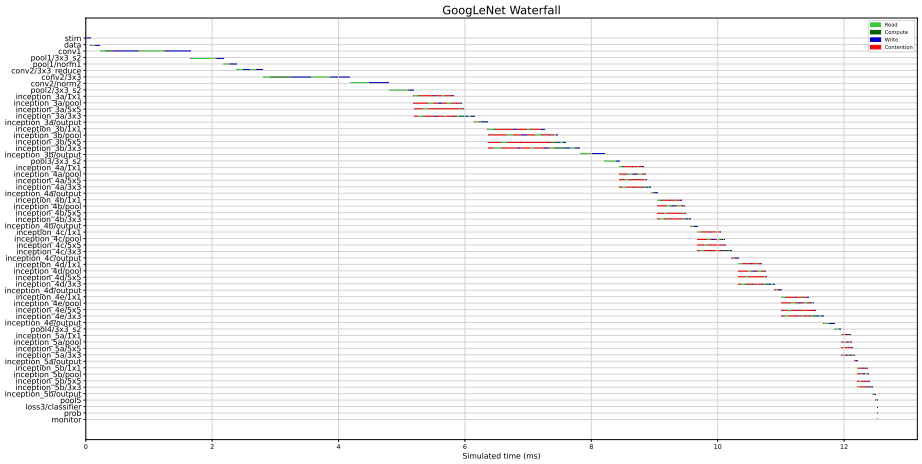

Figure 18: GoogLeNet timing diagram waterfall with contention (image #1/1)

A.2 Trace Log File Syntax

This section provides a technical description of the trace log file syntax that Netmemvisual expects.

```
# ...
# read operation
RD 86126749384:000223 conv6_1 @ 213173120 1478656 bytes
sending delay 0
# ...
# processor computation time
conv6_1 comp delay 94633984
# ...
# write operation
WR 86410836168:000226 conv6_1 @ 214651776 369664 bytes
sending delay 473724416
# ...
```

Listing 1: Truncated and annotated TLM simulation trace log file (SSD)

A small section of a TLM simulation trace log file, shown in Listing 1, will be referenced when discussing log file syntax. Netmemvisual analyzes the supplied

trace log file, looking for the following three operation types: (1) Memory read, (2) Processor computation, (3) Memory write.

Netmemvisual accepts trace log files with the syntax defined below.

1. Keyword Definitions
    - `[OP_TYPE]`: Type of memory access operation; acceptable values: `RD` for read or `WR` for write
    - `[TIMESTAMP_PS]`: Start time of the operation in picoseconds (ps); amount of time that has passed since the TLM simulation started
    - `[MODULE]`: Name of the module involved in the memory access or computation operation
    - `[SIZE]`: Operation size in bytes; used in conjunction with the configuration file to determine the length of this memory operation
    - `[DURATION_PS]`: Length of a compute operation or memory contention event in picoseconds (ps)
    - `[...]`: Extra simulation data that is not relevant to Netmemvisual (such as memory addresses)
2. Memory Access Syntax
    - `[OP_TYPE] [TIMESTAMP_PS]:[...] [MODULE] @ [...] [SIZE] bytes sending delay [DURATION_PS]`
3. Processor Computation Syntax
    - `[MODULE] comp delay [DURATION_PS]`

A.3 Visualization Configuration File Syntax

This section provides a technical description of the `JSON` visualization configuration file that Netmemvisual requires. The purpose of the config file is to specify various visualization settings including the graphical layout of modules and subplots in the diagram, simulation-specific parameters, and module configuration.

As shown in Listing 2, there are four sections in a config file that our tool expects: (1) display, (2) data, (3) subplots, (4) modules. Sections 1-3 are required, while 4 is optional as it depends on specific module parameters. The `display` section configures the image number to display and the grid layout of subplots in the diagram. The `data` section describes the TLM configuration, including contention-aware modeling and memory timing data. The `subplots` section describes the layout of the modules into tracks and subplots on the displayed diagram. The `modules` section configures individual module parameters relevant to the visualization, specifically the number of memory reads and writes per image.

```
{
    "display": {
        "image_num": 0,
        "max_rows": 1
    },
    "data": {
        "contention": true,
        "delay_ps": 50,
        "word_latency_ps": 625
    },
    "subplots": {
        "fc7": {
            "conv6": [
                "conv6_1",
                "conv6_1_relu",
                "conv6_2",
                "conv6_2_relu"
            ],
            "mbox_loc": [
                "fc7_mbox_loc",
                "fc7_mbox_loc_perm",
                "fc7_mbox_loc_flat"
            ],
            "mbox_conf": [
                "fc7_mbox_conf",
                "fc7_mbox_conf_perm",
                "fc7_mbox_conf_flat"
            ],
            "mbox_priorbox": [
                "fc7_mbox_priorbox"
            ]
        }
    },
    "modules": {
        "fc7_mbox_priorbox": [
            2,
            1
        ]
    }
}
```

Listing 2: Example JSON configuration file for SSD 'fc7'

The purpose of each visualization configuration option is summarized below.

1. `display`
    - `image_num`: Image number to display in the visualization. (integer $>= 0$)
    - `max_rows`: Maximum number of rows to populate with subplots before creating a new column. A number that is multiple of the total number of subplots works well here. (integer > 0)
2. `data`

- `contention`: Enable or disable contention plotting, depending on the TLM simulation log file used. (boolean)
- `delay_ps`: Memory access delay in picoseconds (integer)
- `word_latency_ps`: Memory word latency in picoseconds (integer)

3. `subplots`

   - This field accepts a list of one or more `subplot` dictionaries in the following format:

     ```
     "subplot0": {
        "track0": ["module0","module1","module2"],
        "track1": ["module3", module4"]
     }
     ```

   - A subplot dictionary consists of one or more `track` dictionaries to be plotted together. Each `track` dictionary contains an array of one or more module names that will be plotted on the same horizontal axis in the diagram.

4. `modules`

   - This field accepts a list of one or more `module` parameter dictionaries in the following format:

     ```
     "module0": [1, 0],
     "module1": [2, 1]
     ```

   - The first value in the array is the number of memory reads per image, while the second is the number of writes per image. Any modules that are not configured in this section will default to 1 read and 1 write per image.