2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) | 978-1-6654-2603-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSREW53611.2021.00045

(WiP) LLTFI: Low-Level Tensor Fault Injector

Abraham Chan, Udit Kumar Agarwal, Karthik Pattabiraman
The University of British Columbia , Canada.
Email: abrahamc@ece.ubc.ca, uditag97@student.ubc.ca, karthikp@ece.ubc.ca

***Abstract*—As machine learning (ML) has become more prevalent across many critical domains, so has the need to understand ML system resilience. While previous work has focused on building ML fault injectors at the application level, there has been little work enabling fault injection of ML applications at a lower level. We present LLTFI, a tool under development, which allows users to run fault injection experiments on C/C++, TensorFlow and PyTorch applications at the LLVM IR level. LLTFI provides users with greater fault injection granularity and a better ability to understand how faults manifest and propagate between programmed and ML components. We demonstrate how LLTFI can be applied to a ML application with an end-to-end example.**

***Index Terms*—Error resilience, Machine learning, Testing**

I. Introduction

Machine learning (ML) applications have been ubiquitously deployed across critical domains such as autonomous vehicles (AVs), and medical diagnosis. For example, AVs that make incorrect decisions can cause injuries and fatalities. Therefore, it is important to understand the resilience of ML applications in order to mitigate software bugs and hardware faults.

While faults could be either systematic (e.g., software bugs) or random (e.g., radiation induced bit flips), we focus on the latter, i.e., random transient hardware faults. Due to their unpredictable nature, transient hardware faults are difficult to avoid, and hence need mitigation. Fault Injection (FI) is the traditional way to evaluate the resilience of systems. Rather than injecting faults into hardware, software-implemented fault injection (SWiFI) has become the preferred choice to simulate faults due to their cost and time efficiency benefits.

Low-level SWiFI for C/C++ programs has been well explored, given the abundant selection of tools available, such as LLFI [1], PINFI [2], DOCTOR [3], Ferrari [4], G-SWIFT [5]. These tools usually operate at compile time or runtime, where intermediate representation (IR), assembly instructions or binary code are perturbed. However, most developers do not write ML applications in C/C++, but rather in higher level languages like Python [6]. Additionally, they use popular ML frameworks such as TensorFlow [7] and PyTorch [8]. ML-specific SWiFI tools have therefore been developed e.g., TensorFI [9], PyTorchFI [10], and TF-DM [11] - we refer to these as ML FI tools. ML FI tools inject faults into tensor operators and tensors, representing weights and neurons.

Unlike low-level SWiFI tools, ML FI tools are unable to inject faults into specific registers and instructions. For example, a convolution operation can be lowered into many instructions including add, multiply, and arithmetic shifts. A hardware fault may only impact a single instruction (i.e., a left shift), but a ML FI tool would modify the output of the entire convolution operation due to its limited visibility into those instructions. As a result, ML FI tools' ability to simulate transient hardware faults are constrained by the high-level abstractions of the ML frameworks. However, lower level FI tools can inject faults directly into individual instructions.

In this paper, we introduce LLTFI[1], a unified SWiFI tool that supports fault injection of *both C/C++ programs and ML applications written using high-level frameworks such as TensorFlow and PyTorch*. To the best of our knowledge, we are the first to propose a single unifying fault injection tool as described. LLTFI can inject into specific instructions and registers, in ML models written in high-level frameworks, and visualize fault injection in low-level control flow graphs (as opposed to TensorFlow or PyTorch graphs) to study how errors propagate between instructions and nodes. Note that we do not intend for LLTFI to replace high-level ML fault injectors, but rather to complement the injectors.

We develop LLTFI on top of LLFI [1], since it conducts fault injection at the LLVM IR level. We choose the LLVM IR abstraction level as it offers instruction and virtual register information, while remaining hardware independent.

We make the following contributions:

- Build a SWiFI tool that supports C/C++ programs, as well as ML models written in TensorFlow and PyTorch.
- We adapt instruction selection to facilitate low-level fault injection in ML programs.

II. Background and Fault Model

In this section, we explain what ML Frameworks, LLFI, MLIR, ONNX are, and why they are relevant to LLTFI. We also explain LLTFI's fault model.

ML Frameworks. ML developers typically use ML frameworks so they can avoid rewriting well optimized ML operators and exploit support for hardware acceleration. TensorFlow [7] and PyTorch [8] are examples of popular ML frameworks. Both frameworks represent sequences of ML operations as directed acyclic graphs to facilitate optimizations.

LLFI [1] is a low level fault injector that injects faults at the LLVM Intermediate Representation (IR) level. It leverages the LLVM [12] compiler to compile programs, written in high level programming languages such as C/C++, to LLVM IR. LLFI instruments (i.e., inserts fault injection calls) into the LLVM IR code. Subsequently, LLFI randomly activates these fault injection calls during runtime to simulate fault injection.

[1]LLTFI is available at https://github.com/DependableSystemsLab/LLTFI

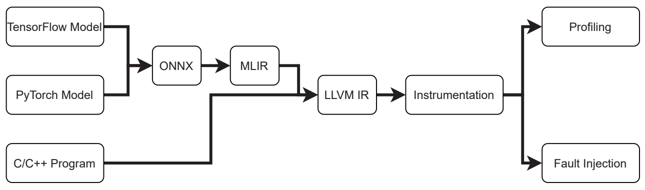


Fig. 1: Workflow diagram of LLTFI

ONNX [13] stands for Open Neural Network Exchange, which is an open serialization format for ML models. It provides a common set of ML operators across popular ML frameworks like TensorFlow and PyTorch, as well as others like scikit-learn. An example ONNX operator is MaxPool, since many ML models utilize maximum pooling. While MaxPool is supported at the application level, it is not directly lowerable to LLVM IR without substantial transformations.

MLIR [14] stands for Multi-Level Intermediate Representation. It is a hybrid IR that supports heterogeneous optimizations across ML operations. MLIR has multiple dialects - each represents a group of MLIR operations. In this paper, we refer to the LLVM dialect of MLIR as MLIR. Despite MLIR's LLVM dialect being isomorphic with LLVM IR, it does not have LLVM IR features like phi nodes. We use MLIR as an intermediary between LLVM IR and ONNX.

Fault Model. We primarily consider transient hardware faults in our work. Transient hardware faults occur as a result of random particle strikes on flip-flops and the chip's logic elements, and cause bits to flip from 0 to 1 or vice versa. We do not consider faults in the instruction encoding or in the control logic of the processor. We also assume that these faults do not modify the structure of the control flow graph. These assumptions are in line with other papers [1, 15, 16].

III. Methodology

Fig. 1 demonstrates the workflow diagram of LLTFI. We show how LLTFI can work for both programmed applications (C/C++) programs and ML models (that use ML Frameworks).

A. C/C++ Programs

LLTFI is built on top of LLFI and is fully backwards compatible. While developing LLTFI, we upgraded the entire LLFI tool, which was originally written for LLVM 3.4, to LLVM 12.0. This is a critical step as LLVM 3.4 has no support for MLIR, which is required to lower ML programs to LLVM IR. This upgrade also ensures that LLTFI is compatible all of the newest C/C++ features, and LLVM optimization passes.

B. ML Programs

Selecting the lowering mechanism. There are multiple methods to lower high-level ML models to intermediate representation (IR). We explain why LLTFI lowers ML models to MLIR using ONNX-MLIR [17] over other alternatives like Glow [18], XLA [19] or TVM [20].

Glow is a ML compiler, which lowers a neural network dataflow graph into high-level Glow IR, followed by low-level Glow IR. However, Glow does not lower to LLVM IR.

XLA and TVM are both compilers capable of lowering TensorFlow and PyTorch models into LLVM IR. XLA first lowers into XLA high-level optimizer representation, while TVM first lowers into Halide IR [21], before lowering to LLVM IR. While XLA and Halide enables more speed optimizations than MLIR, their compilation time overhead is also greater. MLIR, built with testability in mind, better preserves the semantics of ML models. Additionally, MLIR's development is well integrated with LLVM and offers easier extensibility.

A script to compile ML models. LLTFI provides a single script that converts ML models into LLVM IR, using several publicly available tools. To be clear, LLTFI does not convert Python code into LLVM IR for fault injection. Rather, it lowers a trained ML model, with weight and architectural information, into LLVM IR and performs fault injection.

LLTFI first converts all ML models to the ONNX [13] format. ONNX's open exchange format allows LLTFI to support both TensorFlow and PyTorch. Then, the ONNX file is converted into MLIR through ONNX-MLIR [17]. Finally, we convert MLIR into LLVM IR, using the `mlir-translate` tool in LLVM 12.0. From this point onwards, LLTFI can inject faults into the LLVM IR, alike lowered C/C++ programs. The LLVM IR of the ML models can also be linked to that of the programmed components at this step. This allows for fault injection experiments on applications consisting of both programmed and ML components.

Adapting low-level fault injection for ML models. LLTFI injects faults into values at uniform probability, including address values. Since address values are abundant and accessing illegal addresses cause crashes, fault injected models are much more likely to crash rather than produce silent data corruptions (i.e, misclassifications). This is problematic for users aiming to compare fault injection results with ML FI tools, which do not cause program crashes. Hence, we offer a ML injection pass for users to skip certain instructions while injecting directly into math and logic operations in basic blocks representing the high-level ML computation graphs.

C. Fault Types

Presently, LLTFI supports the injection of bitflips, which flips bits from 0 to 1 or vice versa, in the destination register of instructions. These bitflips can be single (i.e., one bitflip on a single instruction in a program) or multiple (i.e., multiple bitflips across multiple instructions or multiple bitflips in a single instruction). Users can also specify the bit position in which fault injection is performed. All of these options can be specified in a YAML file by the user.

IV. Example of running LLTFI on a ML Program

We demonstrate how LLTFI lowers a ML model for low-level fault injection (FI) and how it helps users visually understand the effects of FI. Suppose we have a ML model written in Python, using the TensorFlow and Keras [22]

APIs, as shown in Listing 1. This model is trained on the MNIST [23] dataset and aims to classify handwritten images of digits into a single numerical value between 0 and 9. We compile this model, train it and export its weights and architecture information to a saved model format.

LLTFI's compile script automatically converts the saved model into LLVM IR, from Listing 1 to 3 as depicted in Fig. 2. In this example, however, we break down the internal steps. The saved model is first converted into ONNX using the `tf2onnx` tool, resulting in model.onnx. Then, model.onnx is lowered into a MLIR file, model.mlir, using the ONNX-MLIR tool. We show a snippet of *model.mlir* in Listing 2. Notice that the MLIR is very similar to LLVM IR [12]. Finally, model.mlir is converted into LLVM IR, *model.ll*, using the `mlir-translate` tool. Listing 3 shows the corresponding snippet of the LLVM IR file.

Despite obtaining the LLVM IR code, model.ll, it cannot be executed directly. The model is contained within a function called *main_graph()*, which is invokable by the *run_main_graph()* function. We write a controller program in C to read an image file into a tensor, and to invoke the model with the input tensor, as shown in Listing 4. We compile the controller program using LLVM's clang to obtain controller.ll, which we then link with model.ll, to obtain a single unified LLVM IR file, *program.ll*.

Finally, we apply LLTFI to perform FI on *program.ll*. We first instrument it to add fault injection calls and profile it (i.e., run the instrumented code without any faults). In this example, we pass in a handwritten image of 8 to the controller program, as shown in Fig. 3a. The output of the model is an array of length 10, representing the digit-wise softmax output by the ML model. In the figure, we show the classes inferred with the highest probability. During the profiling run, LLTFI generates a golden output of $[0, \ldots, 0, 0, 0.999989, 0]$. This means that the model has classified the test image as 8 with a probability of 0.999989. However, in the fault injection run, LLTFI generates an erroneous output of $[0, \ldots, 0, 1, 0, 0]$. Instead of correctly classifying it as 8, the model misclassifies the image as 7.

LLTFI allows users to visualize the effects of FI experiments, in the form of a control flow graph (CFG). We use Graphviz [24] to generate the visual CFG. In Fig. 3b, we show a small snippet of the CFG generated by LLTFI for this example. Each node represents a LLVM instruction - the top-most node corresponds to the *extractvalue* instruction on line 3 in Listing 3. We observe that LLTFI injected a fault at the highlighted node, the destination register of a *fadd* instruction. Using the line number information, we can map this error back onto the LLVM IR and MLIR code. This *fadd* instruction is located on line 989, which can be backtraced to line 6 in both Listing 3 and Listing 2. LLTFI's graph visualization feature offers an intuitive way for users to understand how low-level errors can affect their ML output.

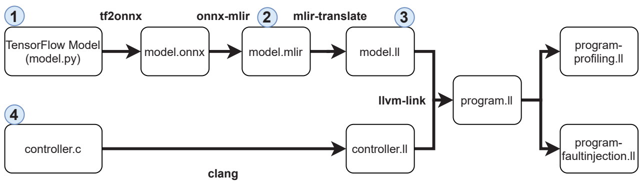


Fig. 2: Example applying LLTFI on a TensorFlow model with a controller. Numbered labels correspond to code listings.

Listing 1: model.py - TensorFlow model

```
1    model = models.Sequential()
2    model.add(
3      layers.Conv2D(32, (5, 5), activation="relu", ↩
           input_shape=(28, 28, 1)))
4    model.add(layers.MaxPooling2D((2, 2)))
5    model.add(layers.Conv2D(64, (5, 5), activation="↩
       relu"))
6    model.add(layers.MaxPooling2D((2, 2)))
7    model.add(layers.Flatten())
8    model.add(layers.Dense(10, activation="softmax"))
```

Listing 2: model.mlir - a basic block in *main_graph()*

```
1    ^bb170:  // pred: ^bb169
2    ...
3    %1672 = llvm.extractvalue %1616[1] : !llvm.struct↩
         <(ptr<f32>, ptr<f32>, i64)>
4    %1673 = llvm.load %1672 : !llvm.ptr<f32>
5    %1674 = llvm.fmul %1661, %1671  : f32
6    %1675 = llvm.fadd %1674, %1673  : f32
7    %1676 = llvm.extractvalue %1616[1] : !llvm.struct↩
         <(ptr<f32>, ptr<f32>, i64)>
8    llvm.store %1675, %1676 : !llvm.ptr<f32>
9    %1677 = llvm.add %1650, %1649  : i64
10   llvm.br ^bb169(%1677 : i64)
```

Listing 3: model.ll - a basic block in *main_graph()*

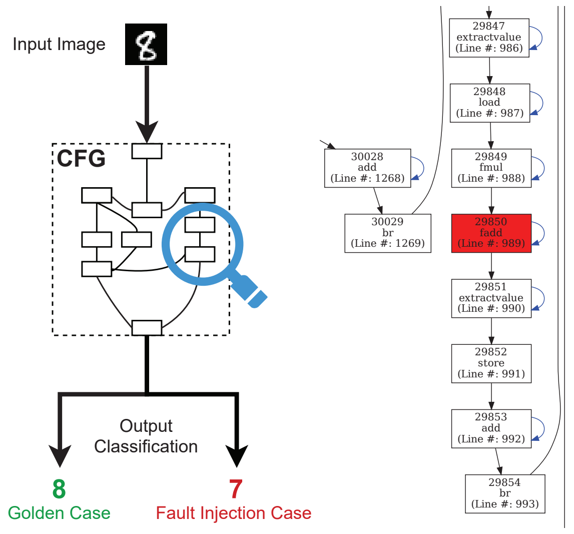
```
1   553:        ; preds = %550
2     ...
3    %575 = extractvalue { float*, float*, i64 } %514,↩
        1, !dbg !813
4    %576 = load float, float* %575, align 4, !dbg ↩
        !814
5    %577 = fmul float %566, %574, !dbg !815
6    %578 = fadd float %576, %577, !dbg !816
7    %579 = extractvalue { float*, float*, i64 } %514,↩
        1, !dbg !817
8    store float %578, float* %579, align 4, !dbg !818
9    %580 = add i64 %551, 1, !dbg !819
10   br label %550, !dbg !820
11    ...
12  !816 = !DILocation(line: 989, column: 12, scope: ↩
        !263)
```

Listing 4: controller.c - to invoke TensorFlow model

```
1   OMTensorList *run_main_graph(OMTensorList *);
2
3   int main(...) {
4     OMTensorList *input = read_input(image_file);
5     OMTensorList *outputList = run_main_graph(input);
6   }
```

V. Preliminary Evaluation

As LLTFI is a work in progress, we report the time overhead of profiling and fault injection (FI) by LLTFI on working benchmarks in Table I. Profiling time is a one-time



(a) Classifying an input image (b) LLTFI generated CFG

Fig. 3: Running LLTFI on MNIST. Snippet of CFG in *main_graph()*. The fault injected instruction is highlighted.

cost to analyze the program (e.g., obtain instruction counts) at runtime, while FI time is a recurring cost, the time taken to run the fault injected program - these times are compared to the baseline program execution time to obtain the overhead as a slowdown factor. For FI, we inject a single bit flip into each of the benchmarks. Our benchmarks include TensorFlow and PyTorch ML programs trained on MNIST [23] and C/C++ programs from the SPEC CPU2006 [25], Parsec [26], and Rodinia [27] benchmarks. We indicate the inputs used for running each benchmark. The benchmark ML programs consist of simple neural networks (i.e., 5 layers) and are invoked with a controller program, written in C, as described in Section IV. The MNIST-CNN model contains convolution layers, while MNIST-NN only feature fully connected layers. While SPEC, Parsec, and Rodinia benchmarks show that LLTFI performs FI with high overhead (60x slowdown on average) on heavy workloads, similar overheads are observed in the original LLFI [1]. For ML programs, LLTFI has higher FI overhead (0.25x to 2.5x slower) than ML FI tools. We attribute this to LLTFI's finer FI granularity and likeliness to cause programs to hang (i.e. stuck in loops). In the future, we will explore techniques to reduce the FI overhead.

VI. Related Work

Li et al. [28] were one of the first to characterize the impact of soft hardware errors on deep neural networks (DNN) by performing systematic FI in a DNN simulator. Subsequently, Reagen et al. presented Ares [16], an application-level fault injector that optimizes fault injection speed by using tensor operations on the GPU. In contrast to LLTFI, Ares changes the Keras inference computation.

TABLE I: Profiling and fault injection (FI) overheads by LLTFI for working benchmarks. Dashed inputs indicate defaults.

Benchmark	**Input**	**Profiling (×)**	**FI (×)**
MNIST-CNN-PyTorch	Image of 8	1.5	1.7
MNIST-CNN-TensorFlow	Image of 8	2.5	2.5
MNIST-NN-TensorFlow	Image of 8	0.07	0.25
Mantevo-comd	-x 10 -y 10 -z 10 -N 50	42	51
Mantevo-hpccg	64 64 64	42	58
NPB-BT	-	92	140
NPB-CG	-	49	89
NPB-DC	10000000 ADC.par	1.5	4
NPB-EP	-	8.1	11
NPB-FT	-	56	82
NPB-IS	-	19	39
NPB-LU	-	73	37
NPB-MG	-	19	49
NPB-SP	-	40	37
NPB-UA	-	35	59
Parsec-blackscholes	1 in_16K.txt output.txt	5	13
Parsec-fluidanimates	1 10 in_5K.fluid out.fluid	29	39
Rodinia-backprop	65536	370	450
Rodinia-bfs	1 graph1MW_6.txt	0.33	0.78
Rodinia-kmeans	-i 819200.txt -k 1	13	17
Rodinia-lud	-v -i 512.dat	3.8	7
Cesar-xsbench	-s small	9	10
Spec-bzip2	-1kvv image.jpg	19	34
Spec-hmmer	−seed 10000000 ig.hmm	21	50
Spec-libquantum	33 5	34	74
Spec-mcf	inp.in	15	16
Splash2-ocean	-p1 -o	36	76

The closest ML FI tools to LLTFI are PyTorchFI [10] and TensorFI [9], developed for PyTorch and TensorFlow respectively, whereas, LLTFI works with both frameworks.

PyTorchFI [10] enables single and multiple FI into neurons and weights at the application level, using PyTorch hooks. PyTorch hooks provide a way to perturb model parameters without directly modifying the source code. Similar to PyTorchFI, LLTFI can inject both single and multiple faults in a single run. However, PyTorchFI is unable to perform register-level FI due to its high-level abstraction. Additionally, while PyTorchFI offers support for injection during both inference and training, LLTFI only supports FI at inference time due to the input requirement for trained saved models.

TensorFI [9] can inject faults into the outputs of TensorFlow operators (i.e., add, multlply, convolution). LLTFI can inject faults into operations like add and multiply, which also exist in LLVM IR. Since LLTFI operates at a lower level, it can inject into intrinsics e.g., *fmuladd*, floating point multiply-add. However, LLTFI does not support FI of high level operators like convolution, which do not exist in IR.

VII. Conclusions and Future Work

We show that LLTFI performs low-level fault injection on both C/C++ programs and ML models. Once LLTFI is fully developed, we plan to expand our evaluation to higher complexity ML models and launch large scale fault injection experiments to understand the advantages and disadvantages of fault injection at different levels. We also plan to bolster LLTFI's debugging capabilities for ML models by generating layer-wise saliency maps.

References

[1] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," in *Proc. of QRS '15*, 2015.

[2] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *Proc. of DSN'14*, 2014.

[3] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: an integrated software fault injection environment for distributed real-time systems," in *Proc. of IPDS'95*, 1995.

[4] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, 1995.

[5] J. Duraes and H. Madeira, "Emulation of Software Faults: A Field Data Study and a Practical Approach," vol. 32, 2006.

[6] M. Innes *et al.*, "On Machine Learning and Programming Languages," 02 2018.

[7] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019.

[9] G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Configurable Fault Injector for TensorFlow Applications," in *Proc. of ISSREW'18*, 2018.

[10] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "PyTorchFI: A Runtime Perturbation Tool for DNNs," in *Proc. of DSN-W'20*, 2020.

[11] N. Narayanan and K. Pattabiraman, "TF-DM: Tool for Studying ML Model Resilience to Data Faults," in *Proc. of DeepTest'21*, 2021.

[12] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO'04*.

[13] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[14] C. Lattner *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *Proc. of CGO'21*, 2021.

[15] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Proc. of DSN'12*, 2012.

[16] B. Reagen *et al.*, "Ares: A Framework for Quantifying the Resilience of Deep Neural Networks," in *Proc. of DAC '18*, 2018.

[17] T. Jin *et al.*, "Compiling ONNX Neural Network Models Using MLIR," 2020.

[18] N. Rotem *et al.*, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *CoRR*, 2018.

[19] C. Leary and T. Wang, "XLA," 2017.

[20] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proc. of OSDI'18*, 2018.

[21] Ragan-Kelley *et al.*, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," 2013.

[22] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[23] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, vol. 2, 2010.

[24] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph – static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*, 2003.

[25] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, 2006.

[26] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[27] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of IISWC '09*, 2009.

[28] G. Li *et al.*, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proc. of SC '17*, 2017.