Exploring Deep Learning for In-Field Fault Detection in Microprocessors

Simone Dutto, Alessandro Savino and Stefano Di Carlo
Politecnico di Torino, Control and Computer Engineering Department, Torino, Italy
Contact: stefano.dicarlo@polito.it

***Abstract*—Nowadays, due to technology enhancement, faults are increasingly compromising all kinds of computing machines, from servers to embedded systems. Recent advances in machine learning are opening new opportunities to achieve fault detection exploiting hardware metrics inspection, thus avoiding the use of heavy software techniques or product-specific errors reporting mechanisms. This paper investigates the capability of different deep learning models trained on data collected through simulation-based fault injection to generalize over different software applications.**

***Index Terms*—fault detection, deep learning, monitoring tool, hardware metrics**

I. Introduction

Rapid innovation in strategic industrial fields such as medical, automotive, IoT, and HPC, is pushing for the adoption of high-scaled multi-core processors (i.e., 10nm technology nodes and below) in several domains [1]. This puts pressure on systems' designers and manufacturers to deliver improved availability and reliability starting from the early design phases [2]. Recent advances in machine learning and deep learning may provide new powerful instruments to build advanced fault detection systems supported by the development of fault injection frameworks (e.g., [3; 4; 5; 6; 7]) able to collect huge amount of simulated data.

Two applications of machine learning to support fault injection were presented in [8] and [1]. The first paper aims at reducing the number of injected faults, while the second aims at correlating the results of the injection campaigns with application/platform characteristics. In the fault detection domain, authors in [9] evaluated the impact of multi-bit memory errors (both permanent and transient) in HPC applications using machine learning models. The analysis is performed at a high software level rather than exploiting low-level features to perform hardware-level fault detection.

The main contribution of this paper is investigating how selected deep learning models can detect permanent and transient faults in microprocessors executing software applications and how they are able to (i) generalize over several domains, thus minimizing retraining for different applications, and (ii) work with a set of low-level features that can be easily collected at the microprocessor architectural level. To achieve this goal, a full automated flow, from micro-architecture based fault injection, used to collect data, to the training of the detection model, was created and used to validate the approach using MiBench applications [10] executed over a real Linux Kernel.

II. Machine Learning Based Fault Detection Framework

Figure 1 summarizes the full framework presented in this paper. It covers three main challenges of every machine learning based system: (a) data collection, (b) data analysis, and (c) in-field monitoring.

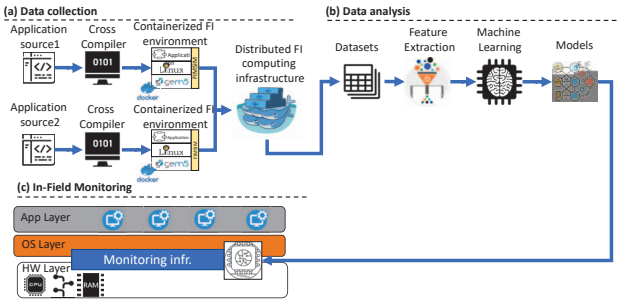

Fig. 1: Machine learning based fault detection framework.

A. Data collection

Fault injection is a powerful tool to investigate the behavior of a system affected by hardware faults [11]. We exploit the combination of two open source tools: gem5 [12] and FIMSIM [5]. gem5 is a micro-architecture simulator able to simulate and profile a full microprocessor architecture running the operating system (OS) and the application software. FIMSIM is an add-on to gem5 that enables fault injection of both permanent faults (stack-at-0, stack-at-1) and single bit upsets (SBU) in the register file of the microprocessor. Using these tools, several data-set entries can be collected by profiling a large set of micro-architectural features at periodic checkpoints, with each entry labeled according to the observed fault effect as (i) *benign*, (ii) *error*, or (iii) *crash*.

To collect large amount of data, the full simulation environment was packed into a Docker container [13] that can be deployed and replicated to a Docker Swarm composed of several distributed computing nodes, thus allowing extremely parallelized injection experiments.

B. Data analysis

gem5 is able to profile about 600 micro-architectural features. However, collected data are noisy and require cleaning. Features containing non-numeric values were removed, dropping columns containing not well-collected data. Moreover,

features with less than 1% variance, i.e., not containing enough informative content, were removed as well. Overall, this process removed half of the available features. Not all features extracted by the simulator are already available and/or can be monitored in real hardware, nonetheless, implementing monitoring facilities for these metrics is feasible but expensive, since it implies changing the hardware architecture. Therefore, a model dependent feature selection phase able to identify a minimum set of features that guarantees good learning performance is required and describe later.

Once data have been collected, the best machine learning model must be selected. Since the goal is to implement the model at the hardware/OS level, it is important to look for simple and fast models. Moreover, the considered data-set has a domain issue to solve: different faults on different binaries produce different features. A single model must be able to either consider the difference or allow easy and fast generalization. These reasons motivated us to consider three models described in the following sections. A detailed description of each model is available in [14].

1) Feed Forward Neural Networks with Transfer Learning: Feed-Forward Neural Network is the simplest deep neural network model [15]. This paper considers a model inspired by the work proposed in [16]. The idea is simple yet powerful: train the model on a data-set obtained by profiling an application for *N* epochs and then specialize the model to work with another data-set, associated to a different application, by training it for *N/5* epochs, thus reducing the training complexity when new applications need to be included. This model requires a feature selection to reduce the number of available features. The solution exploited in this paper is to resort to a Random Forest model [17] to calculate the *feature importance factors*, implementing the procedure described in [18], usually referred to as "Gini importance" or "mean decrease impurity". This approach avoids searching only for linear relationships, as with PCA or correlation coefficients.

2) Domain Adversarial Neural Networks: Domain Adversarial Neural Network is a model to cope with domain adaptation problems, where data at train and test time are structurally similar but distributed differently [19]. The approach consists of training on a so-called *source domain* labeled data and *target domain* unlabeled data. The resulting model should be able to perform on the task independently of the domain, fitting our problem in which we want to generalize the model to unknown applications. Because this model requires working with a high number of features, a DANN with 1D Convolutional layer was evaluated with the total available 300 features.

3) Sparse Stacked Autoencoders with FFNN and TL: An alternative to the first model (FFNN with TL) is using unsupervised models to perform feature selection before submitting information to the classifier. With unsupervised models data do not have to be labeled, i.e., they can be collected with simple profiling, without need of performing fault injection. An Autoencoder is composed of an encoder (mapping the input to a lower dimension), and a decoder (reconstructing the output of the layer before back to its dimension). The loss is the difference between the input and the reconstructed input (also called *reconstruction error*). It is possible to have multiple encoder layers followed by multiple decoders, this model is called Sparse Stacked AutoEncoder (SS-AE). SS-AEs have been used for fault detection to *transform* data before entering the classifier [20]. The idea exploited in this paper is to train the AE (encoder+decoder) to reconstruct the input correctly, then, discarding the decoder and use the encoder as a feature extractor for a fault classifier based on a FFNN with TL, as before.

C. Monitoring infrastructure

Integrating the proposed machine learning models into a real system requires two different infrastructures. First the microprocessor must provide facilities to monitor the identified features. This infrastructure is already available in modern microprocessors through the Performance Monitoring Counters (PMC), which can be accessed at the OS level through dedicated libraries such as PMCTrack [21]. Some features identified by our models (e.g., cache-hit and cache-miss) are already monitored in most commercial microprocessors. For other more specific features the architecture of the microprocessor requires modifications that are out of the scope of this paper. Second, an in-field monitor implementing the considered models is required. A preliminary version of this monitor, implemented at the operating system level as a kernel task, is currently under development. However, it is worth to note that a software implementation of this monitor suffers from the fact that the monitor itself could be affected by faults in the hardware, requiring it to be secured.

III. Experiments

A. Experimental setup

gem5 and FIMSIM were packaged inside a Docker container based on Ubuntu 14.04.6 LTS image. All fault injections were executed on an 32 Intel(R) Xeon(R) Silver 4110 CPU @2.10GHz, 93GB RAM and 5.5T disk space. Each injection required approximately 80 seconds producing 6.4MB of data.

Both permanent (either stuck-at-0 and stuck-at-1) and Single Bit Upsets (SBU) were injected (10,000 injections per fault type and per application) in the register file of the microprocessor with gem5 configured using the AtomicSimpleCpu model with 2-level cache with x86 ISA [22], running a Vanilla Linux kernel. Permanent faults were injected from the beginning of the binary execution in random selected bits, and features collected from the last checkpoint. SBUs were injected similarly, only making the injection time pseudo-random. Four MiBench [10] applications were considered: *qsort*, *basicmath*, *bitcount*, and *ssearch*.

For the training procedure the classical three-way split was adopted: (i) *Train set (50%)* is used to train the model; (ii) *Validation set (20%)* is used to fairly tune the model ; (iii) *Test set (30%)* is the ultimate test to evaluate the performance of the models on unseen data.

Domain adaptation is a key element of the performed analysis. All considered Transfer Learning models were trained/validated on a domain and trained/tested again on another domain. Domain adaptation models were cross-domain validated by training the model on two domains (source and target), then validating on other domains and testing on the target domain.

B. Results

Since different models have different characteristics, results are first analyzed for each model and then compared across models.

1) FFNN with TL: This model requires first to perform feature selection (see Section II-B). After computing the importance score for all features, 19 features with importance significantly higher w.r.t. the full set were selected [1].

Let us now focus on the process to build the baseline model to be used for transfer learning. In our case the baseline was trained on the *basicmath* binary only.

First it is essential to make sure the network is learning properly. When a model is learning correctly the loss on train and validation set converge to the same values with the shape of an elbow as reported in Figure 2 where LR denotes the learning rate, i.e., the amount of weights updated during training. This specific shape expresses that the loss of the network is decreasing to a point of stability and the gap between the train set and validation set means how much the train set is representative of the validation set.

After ensuring that the model is set up to learn correctly, it is time to fine-tune it. The parameters that affect the model are: the LR, other two hyper-parameters β_1 and β_2 that are usually not changed and set to 0.9 and 0.999, the learning algorithm (i.e., Adam or SGD+Momentum), and the scheduler (i.e., StepLR or CycleLR). The reader may refer to [14] for further details on each parameter's meaning. As reported in Table I, among the three models with the best performance, the one using Adam, StepLR and the lowest LR was selected.

At this point the best model, with the last 2 layers frozen, is used as a baseline to perform the transfer learning task. Table II shows that the model can work with different binaries, sometimes even increasing the precision, with few epochs of training after having built the baseline.

It is worth to note here that a high recall is not always a good sign: if the model predicts always fault it would have recall equal to 1 which is clearly not enough. High precision indeed is a good measure of how the model is able to spot the faulty runs.

[1](0) system.cpu.icache.ReadReq_hits::total, (1) system.cpu.itb_walker_cache.tags.occ_blocks::cpu.itb.walker, (2) system.iocache.WriteInvalidateReq_hits::total, (3) system.cpu.icache.overall_hits::cpu.inst, (4) system.cpu.dcache.SoftPFReq_misses::total, (5) system.cpu.itb_walker_cache.ReadReq_accesses::total, (6) system.membus.pkt_count_system.cpu.dtb_walker_cache.mem_side::system.mem_ctrls.port, (7) system.cpu.itb_walker_cache.overall_misses::total, (8) system.cpu.itb_walker_cache.ReadReq_accesses::cpu.itb.walker, (9) system.cpu.dtb_walker_cache.demand_accesses::total, (10) system.cpu.dtb_walker_cache.demand_misses::cpu.dtb.walker, (11) system.mem_ctrls.num_reads::cpu.inst, (12) system.iobus.trans_dist::ReadReq, (13) system.cpu.icache.ReadReq_misses::cpu.inst, (14) system.cpu.itb_walker_cache.tags.avg_refs, (15) system.cpu.itb_walker_cache.ReadReq_hits::total, (16) system.membus.pkt_size_system.cpu.icache.mem_side::system.mem_ctrls.port, (17) system.cpu.icache.demand_accesses::cpu.inst, (18) system.cpu.icache.tags.age_task_id_blocks_1024

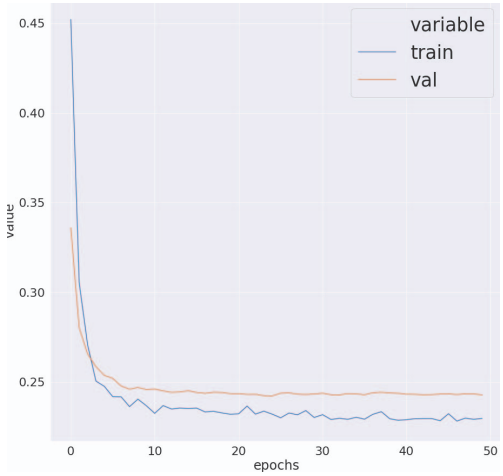

Fig. 2: FFNN+TL: loss over epochs, with LR=0.001 and 50 epochs.

TABLE I: FFNN+TL: validation scores to select the baseline model

Optimizer	Scheduler	textbfEpochs	F1 score
Adam: LR=0.1, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.01, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.001, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8693
Adam: LR=0.0001, Beta1=0.9, Beta2=0.999	*StepLR: N=30 Epochs, gamma=0,1*	50	0.8655
SGD+Momentum: LR=0.1, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8692
SGD+Momentum: LR=0.01, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8651
SGD+Momentum: LR=0.001, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8693
SGD+Momentum: LR=0.0001, Beta1=0.9	*CycleLR: LR_f=LR, LR_i=LR*0.1*	50	0.8195

2) DANN: This model is trained based on a totally different approach. The concept here is to have a model trained on two different domains, one labeled and the other not, and see how the model performs on both domains after the training. The validation set will be on the other two domains.

The parameters to tune in this model are related to the Adam optimizer, i.e., the LR, and the α parameter, which represents how much reverse gradient is flowing into the features' extractor. Table III shows how these parameters were optimized. It is important to remember that the model is not trained on domains in the validation set.

Having selected the best hyper-parameters, the obtained model can be tested using as target all domains and see how it performs. As can be noted in Table IV, the model is not able to generalize from unlabeled data.

3) SS-AE with FFNN and TL: In this case, a SS-AE is used as alternative feature selection process. The SS-AE is trained using MSE and L1 sparsity using only non-faulty runs. The two hyper-parameters to tune are the LR and the α L1 sparsity coefficient (see Table V).

The encoder with the best model from Table V is then trained on the whole (faulty and non-faulty) data-set on a single domain. This model is then used as a baseline to compare performance on other domains before and after some epochs of training on them.

It is interesting to notice in the first column of Table VI that this model has the best precision on unseen domains.

TABLE II: FFNN+TL: performance on test set after 10 epochs of TL

Binary	Precision: baseline	Precision: after	Recall: baseline	Recall: after
basicmath	1.00	/	0.74	/
qsort	0.38	1.00 (+0.62)	1.00	0.82(-0.18)
search	0.46	0.82 (+0.36)	1.00	0.33(-0.67)
bitcount	1.00	1.00(+0.0)	0.73	0.74(+0.01)

TABLE III: DNN: F1 score on validation set

LR/ALPHA	0.1	0.01	0.001
0.01	0.59	0.75	0.59
0.001	0.59	0.62	0.59
0.0001	0.59	0.76	0.75

4) Comparison: Due to the limited space, all results discussed in this section are related to *stuck-at-0* fault datasets. However, comparable performance have been obtained considering *stuck-at-1* and *SBU*.

To summarize, as reported in Figure 3, based on the available experiments, the best performing model is the FFNN with TL for each domain. Not only it reaches the best performances on each domain, but it is also the simplest and it works with 19 features. The small features selection is crucial considering the idea of implementing an in-field monitor architecture. However, it is very interesting to notice the precision of the SS-AE architecture before transfer learning. This could underline the ability of the model of generalizing for many domains. However, further investigation in this direction are required.

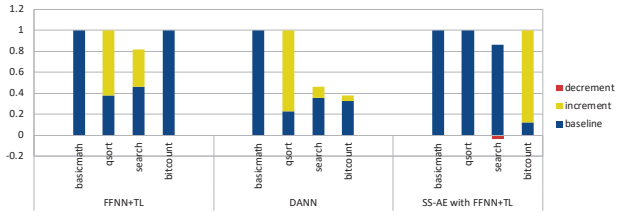

Fig. 3: Test set precision scores on different binaries from different models before and after the training on the specific domain. Basicmath is used to build the baseline.

IV. Conclusions

This paper presented a broad analysis of how deep learning models can be used to build a hardware fault detection framework in microprocessor based systems. The main ideas that guided the analysis were: working with hardware-level metrics that could be monitored in a microprocessor, and working with models able to be generalized and transferred to monitor several applications. Overall, by testing three different models, we showed that FFNN with TL are a potential candidate for the proposed problem.

TABLE IV: DNN: Performance on all domains

Binary	Precision: before	Precision: after	Recall: before	Recall: after
basicmath	1	\	0.80	\
qsort	0.23	1	1	0.77
search	0.36	0.46	0.8	1
bitcount	0.33	0.38	0.8	1

TABLE V: SS-AE: Best (lowest) loss on non-faulty data-set on a single domain

LR/α	0.01	0.001	0.00001
0.01	0.0704	0.0427	0.0704
0.001	0.0355	0.0704	0.0462
0.0001	0.0355	0.0413	0.0704

TABLE VI: SS-AE with FFNN and TL: scores on test set before and after 10 epochs of training on the binary.

Binary	Precision: before	Precision: after	Recall: before	Recall: after
basicmath	1.00	\	0.80	\
qsort	1.00	1.00	0.20	0.78
search	0.86	0.83	0.11	0.32
bitcount	0.12	1.00	0.23	0.77

References

[1] F. R. da Rosa, R. Garibotti, L. Ost, and R. Reis, "Using machine learning techniques to evaluate multicore soft error reliability," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 6, pp. 2151–2164, 2019.

[2] A. Vallero, A. Savino, A. Chatzidimitriou, M. Kaliorakis, M. Kooli, M. Riera, M. Anglada, G. Di Natale, A. Bosio, R. Canal, A. Gonzalez, D. Gizopoulos, R. Mariani, and S. Di Carlo, "Syra: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 765–783, 2019.

[3] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, "Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 313–322, 2012.

[4] A. Benso, A. Bosio, S. Di Carlo, and R. Mariani, "A functional verification based fault injection environment," in *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, 2007, pp. 114–122.

[5] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero, "Fimsim: A fault injection infrastructure for microarchitectural simulators," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*. IEEE, 2011, pp. 431–432.

[6] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 172–182.

[7] A. Vallero, D. Gizopoulos, and S. Di Carlo, "Sifi: Amd southern islands gpu microarchitectural level fault injector," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 138–144.

[8] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Virtualized-fault injection testing: A machine learning approach," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 297–308.

[9] A. Vishnu, H. van Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie, "Fault modeling of extreme scale applications using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 222–231.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. USA: IEEE Computer Society, 2001, pp. 3–14.

[11] M. Eslami, B. Ghavami, M. Raji, and A. Mahani, "A survey on fault injection methods of digital integrated circuits," *Integration*, vol. 71, pp. 154 – 163, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016792601930402X

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[13] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[14] S. Dutto, "Virtual tool-boxing for robust management of cross-layer heterogeneity in complex cyber-physical systems," Master's thesis, Politenico di Torino, 2020.

[15] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.

[16] C. Zhang, L. Xu, X. Li, and H. Wang, "A method of fault diagnosis for rotary equipment based on deep learning," in *2018 Prognostics and System Health Management Conference (PHM-Chongqing)*. IEEE, 2018, pp. 958–962.

[17] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[18] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[19] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 2096–2030, 2016.

[20] A. Borghesi, A. Libri, L. Benini, and A. Bartolini, "Online anomaly detection in hpc systems," in *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE, 2019, pp. 229–233.

[21] J. C. Saez, A. Pousa, R. Rodríguez-Rodríguez, F. Castro, and M. Prieto-Matias, "PMCTrack: Delivering Performance Monitoring Counter Support to the OS Scheduler," *The Computer Journal*, vol. 60, no. 1, pp. 60–85, 01 2017. [Online]. Available: https://doi.org/10.1093/comjnl/bxw065

[22] L. Lowe-Power, "gem5 documentation," [Online] https://www.gem5.org/documentation/, 2020.