Approximate Checkers

Abdulrahman Mahmoud
amahmou2@illinois.edu
University of Illinois at Urbana-Champaign

Paul Reckamp
paulrr2@illinois.edu
University of Illinois at Urbana-Champaign

Panqiu Tang
panqiut2@illinois.edu
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign

Sarita V. Adve
sadve@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

With the end of conventional CMOS scaling, efficient resiliency solutions are needed to address the increased likelihood of transient hardware errors. Many resiliency solutions consider redundancy in time or space in order to detect errors during deployment. However, full modular redundancy is expensive, introducing large performance overheads in order to improve reliability.

In this work, we propose the use of approximate checkers to detect when errors may occur during execution, and only trigger application re-execution when the approximate checker identifies an error. We implement the approximate checker using a neural network, and show that for many applications, the approximate checker can achieve very high accuracy in detecting errors.

1 Overview

With the end of conventional CMOS scaling, hardware is becoming increasingly susceptible to errors in the field [3–5, 10, 12, 13]. Systems must be able to handle such failures in order to guarantee continuous error-free operation. Hardware error detection mechanisms form a crucial part in devising such reliability solutions. Traditional solutions use heavy amounts of redundancy (in space or time) to detect hardware faults, and can introduce very high overheads. Despite the high overheads, modern systems today commonly rely on full redundancy to meet their resiliency targets even though lower cost techniques have been proposed in the literature [6–9, 13]. For example, as recently as March 2019, Tesla announced its new Full Self-Driving processor for it's autonomous driving fleet, which includes a fully redundant co-processor on chip to guard against hardware errors [11].

To tackle the high costs introduced by full redundancy, we propose augmenting certain applications with a small neural network (NN) which raises a flag when an error manifests itself during execution. The neural network acts as a fast, approximate checker, quickly making a decision whether or not it detected an error, in order to avoid an indiscriminate re-execution for all applications.

Figure 1 depicts the general overview of how an approximate checker functions. An approximate checker takes as input the input and output of an application, and predicts whether an error occurred or not. Based on the prediction by the approximate checker, the runtime system can make a decision to rerun the application or not, based on the confidence of the approximate checker in identifying the occurrence of an error during execution.

Intuitively, an approximate checker looks for a correlation between the input of an application, and the output observed by the application. By learning the relationship between the input and the output, the approximate checker can run relatively quickly while providing a "sanity check" for the user without requiring a full rerun of the application. Since not all applications require precise outcomes, a small error which does not affect the final output would be learned by the approximate checker as not requiring re-execution.

2 Design of Approximate Checker

One primary objective in the design of the approximate checker is that it needs to be fast in order to offset the cost of redundantly rerunning the application. Thus, we avoid deeper neural networks (DNNs) because although they might be more accurate in general, they also incur a larger performance overhead from the many additional weights and hidden layers. For our design, we target a few hidden layers with a relatively small number of neurons per layer in a fully connected fashion, to address this issue. Further, the approximate checker can be run on a dedicated NN accelerator to minimize the runtime of checking for errors.

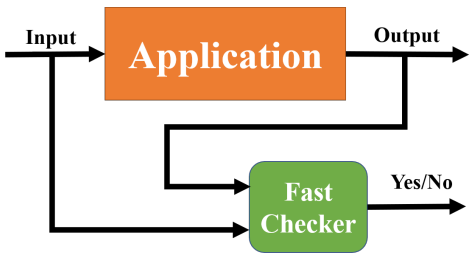

Figure 1. An approximate checker takes as input the input and output of an application, and checks whether an error occurred during execution.

Table 1. Summary of applications explored and results

Application	Domain	Topology	Error Model	Accuracy (%)	Checker	Category
Black Scholes [1]	Financial Analysis	7->128->64->2 7->16->8->2	Random Output	85 77	✓ ✓	2
Inversek2j [1]	Robotics	4->128->64->2 4->8->2	Random Output	98 94	✓ ✓	2
JPEG [1]	Compression	6145->1280->64->2	Random Output	54	✗	1
K-means [1]	Machine Learning	7->128->64->2 7->8->4->2	Random Output	57 54	✗ ✗	1
Sobel [1]	Image Processing	11->128->64->2 11->8->2	Random Output	89 80	✓ ✓	3
Jmeint [1]	3D Gaming	19->128->64->2	Random Shuffle	73	✓	2
FFT [1] [2]	Signal Processing	128->128->64->2	Random Scaling	86	✓	1
AES [2]	Security	64->128->64->2	Random Shuffle	50	✗	1
Backprop [2]	Machine Learning	78->128->64->2	Random Shuffle	99	✓	3
GEMM [2]	Linear Algebra	192->128->64->2 192->16->8->2	Random Scaling	99 99	✓ ✓	1
NW [2]	Bioinformatics	384->128->64->2	Random Shuffle	50	✗	2
Sort [2]	Common Kernel	60->128->64->2	Swap Two Values Random Shuffle	50 99	✗ ✓	1
SPMV [2]	Linear Algebra	384->128->64->2	Random Shuffle	50	✗	1

To train the approximate checker, we generate 80,000 tuples of the form {input, output, label}, where the output is either the correct output as generated by the application without a perturbation, or alternatively an erroneous output that is generated by transforming the legal output. We generate 40,000 correct and 40,000 incorrect outputs, labeling the training entry accordingly. Incorrect outputs are obtained for many applications by substituting output values with random uniform values of the same type. We ensure that the output is legal (i.e., it is not trivially incorrect); however, our corruption model is extreme to test whether an approximate checker can learn anything before fine-tuning to a more rigorous and realistic error model. For testing, we use 20,000 tuples split evenly between correct and incorrect entries and obtained in a similar fashion to the training set, described above.

3 Results

We explored 13 applications from the AxBench [1] and MachSuite [2] benchmark suites, spanning many different domains. Table 1 shows each application studied, along with the NN topology used for the Approximate Checker (Column 3) and the error model used to generate erroneous outputs (Column 4).

We partitioned the applications into 3 classes, based on the potential of employing an approximate checker

1. Outputs can be checked precisely.
2. Outputs can be checked approximately.
3. No known approximate output validation exists.

For example, a precise method to check SORT would be to scan over the output and ensure the final order is sorted correctly, while also checking that all the input values appear in the output. Category 2 applications, such as Blackscholes, are based on a model or a heuristic, and the precise output value has less importance in the grand scheme of things. Finally, for Category 3 applications, no known approximate output validation exists: known validation techniques are either indirect or are variations on original computation (e.g. Sobel edge detection).

Column 5 of Table 1 shows the accuracy measured for the different applications, and Column 6 indicates whether the application has potential for further exploration (based on whether the accuracy is greater than 70%).

We find that the Approximate Checker for some applications such as JPEG, AES, sort, and SPMV do not learn anything (an accuracy near 50%), despite a very egregious error model being used (randomization of the original, correct output). This eliminates these applications from further study, since we would not expect a fine-grained error model to successfully predict errors. To illustrate this point, we can see the example with sort, where we found high accuracy with randomization, but very low accuracy once we changed the error model to swap two values of the originally sorted output.

However, many applications did surprisingly well. For these applications, our preliminary results show that an approximate checker has a lot of promise, with some applications gaining very high accuracy such as Inversek2j, GEMM, and Backprop. This encourages further exploration of more realistic error models. We also find that some Category 1 applications observe decent accuracy (e.g., FFT). This could be a result of the error model, but also warrants additional exploration.

4 Conclusion

In this work, we present the idea of an approximate checker, a low-cost companion NN model to an application to quickly identify if an error occurring during execution resulted in an output corruption. We provide a taxonomy for applications which may benefit from an approximate checker, and show that for some applications, an approximate checker can be trained to have very high accuracy.

Moving forward, we plan to explore more rigorous error models for training approximate checkers, and expand to additional applications that can benefit from this form of error detection.

Acknowledgements

This material is based upon work supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. We also thank the reviewers for their valuable feedback.

References

[1] P. Lotfi-Kamran H. Esmaeilzadeh A. Yazdanbakhsh, D. Mahajan. 2017. AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing. *IEEE Design and Test* 34, 2 (April 2017), 60–68.

[2] Y. S. Shao G. Wei D. Brooks B. Reagen, R. Adolf. 2014. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Raleigh, North Carolina.

[3] Shekhar Borkar. 2005. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro* 25, 6 (2005).

[4] Franck Cappello, Geist Al, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. 2014. Toward Exascale Resilience: 2014 Update. *Supercomput. Front. Innov.: Int. J.* (2014).

[5] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. 2009. High-end Computing Resilience: Analysis of Issues Facing the HEC Community and Path-forward for Research and Development. *Whitepaper* (2009).

[6] W. Dweik, M. Annavaram, and M. Dubois. 2014. Reliability-Aware Exceptions: Tolerating Intermittent Faults in Microprocessor Array Structures. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.114

[7] Dan Ernst et al. 2003. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *MICRO*.

[8] R. Hegde and N. R. Shanbhag. 2000. Algorithmic noise-tolerance for low-power signal processing in the deep submicron era. In *2000 10th European Signal Processing Conference*. 1–4.

[9] K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson. 2008. Fault-Tolerant Design of the IBM Power6 Microprocessor. *IEEE Micro* (2008).

[10] Philippe Ricoux. 2013. European Exascale Software Initiative EESI2-Towards Exascale Roadmap Implementation. *2nd IS-ENES workshop on high-performance computing for climate models* (2013).

[11] Sean Hollister. 2019. Tesla's new self-driving chip is here, and this is your best look yet. Website. (2019). https://www.theverge.com/2019/4/22/18511594/tesla-new-self-driving-chip-is-here-and-this-is-your-best-look-yet

[12] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A Debardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. 2014. Addressing Failures in Exascale Computing*. *International Journal of High Performance Computing* (2014).

[13] James F Ziegler and Helmut Puchner. 2004. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress.