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. Typically, many resiliency solutions consider redundancy in time or space in order to detect errors during deployment. However, full modular redundancy is expensive, and takes away resources that may be utilized for other performance needs.

In this work, we propose the use of approximate checkers to detect when errors may have occurred during execution, and only trigger re-execution of the program 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 [3–8]. 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. For example, running an application twice and comparing outcomes to ensure no errors occurred could incur a 2× overhead, a performance penalty that may be too high for some time-sensitive domains.

To address the high cost introduced by full application redundancy, we propose augmenting applications with a small neural network (NN) which can provide a "ballpark" estimate on whether an error was detected or not during execution. The neural network acts as a fast, approximate checker, quickly making a decision whether or not a re-execution is required, based on the confidence of the checker as to whether an error occurred or did not occur.

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 (with a confidence associated with the prediction, as typically provided by a NN) whether an error occurred or not. Based on the output of the Approximate Checker, a decision by the system can be made as to whether the application needs to be redundantly executed or not.

Intuitively, the Approximate Checker is looking for a correlation between the input of an application, and the output observed from the application. By learning the relationship between the input and the output, the Approximate Checker can run relatively quickly and provide a "sanity check" for the user without requiring a full rerun of the application.

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 want to avoid deeper neural networks (DNNs) because although they might be more accurate in general, they also incur additional computational and memory overheads as a result of many weights and 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.

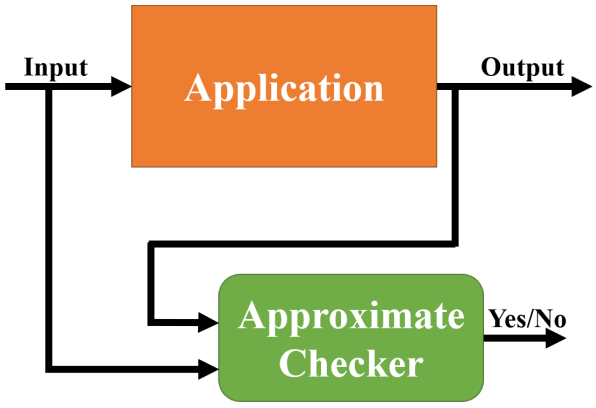

Figure 1. An approximate checker takes as input the input and output of an application, and checks whether an error occurred during execution.

Table 1. Summary of apps and results

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

To train the Approximate Checker, we generate 80,000 tuples of the form {input, output, label}, where the output can be the correct output as generated by the application, or 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. 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. We use 20,000 tuples for testing, with 50% correct and 50% incorrect entries, similar to training the Approximate Checker.

3 Results

We explored 13 applications from the AxBench [1] and MachSuite [2] benchmarking 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 erronous outputs (Column 4).

We find that the concept of an Approximate Checker can be useful for some domains, but not all. 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 K-means, JPEG, AES, NW, sorting, and SPMV do not learn anything (an accuracy near 50%, or random guessing by the NN), 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 bad 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. For applications with mediocre accuracy (such as Blackscholes, Sobel, Jmeint), our results indicate that the Approximate Checker did in fact learn something, and perhaps an exploration of the topology space could improve accuracy for the random error model, before proceeding to more precise error models.

4 Conclusion

In this work, we present the idea of an Approximate Checker, a low-cost companion NN model to an application which can quickly identify if an error occurring during execution resulted in an output corruption. Using very small NN models, we show that for some applications, an Approximate Checker has very high potential to find a correlation between the input and output, while for other applications it is very difficult for a small NN model to differentiate between a correct and erroneous output. Moving forward, we would like to better understand why Approximate Checkers perform well for certain applications (or domains) over others, and also explore whether more realistic error models can be captured using Approximate Checkers.

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

[7] 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).

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