Towards More Precision in Approximate Computing [*]

Radha Venkatagiri Abdulrahman Mahmoud Sarita V. Adve

University of Illinois at Urbana-Champaign

{venktgr2,amahmou2,sadve}@illinois.edu

Imperfect Moore's law scaling has led to recent trends that consider systems that generate incorrect outputs. The emergent field of approximate computing considers deliberate, but controlled, relaxation of correctness for better performance, energy or reliability. While the fast growing popularity of the field indicates the general acceptance, even eagerness, for the idea, there are many hurdles to the practical application of approximate computing in today's systems that are preventing it from realizing its full potential. Many of these hurdles will naturally be surmounted by the evolution of the field, but it is imperative that a strong fundamental base be established upon which future research directions can be built. In this paper we highlight some of the challenges faced by approximate computing, namely, standardized benchmarks, output error evaluation and reducing programmer burden through automation.

Standardized Benchmarks and Output Error Evaluations

Key among the challenges are the unavailability of standardized benchmark suites and standards for evaluating the error in their outputs. While benchmarks used in different approximation techniques are emerging [1, 2] and can guide application selection, there is a need for formalizing standard benchmark suites that can be consistently used for future works in approximate computing and enable an apples to apples comparison of techniques. There is also a crucial need to standardize the methodology for evaluating the errors introduced in the output of these benchmarks due to approximation.

Error Metrics: Approximation techniques are often measured based on how much accuracy is lost in the final output. There are many error metrics that may be used to measure this accuracy loss [3]. The choice of the error metric is application-specific and even within a single application, there may be different applicable metrics.

For instance, let us consider the PARSEC benchmark Blackscholes which produces the pricing for a portfolio of European options. Depending on the input size, the portfolio can consist of anywhere between 4,096 and 65,536 options. The first question that arises is how to calculate the error (introduced by the approximation technique) in a given option price? Should we calculate the absolute dollar value difference between the approximately calculated output (*approximate output*) and the precise output (*golden output*)? Or should we calculate the error relative to the golden output? While relative error may be appropriate in some cases, it may be unreasonable in cases where even small relative errors lead to tangible losses in dollar amounts. For example, in certain scenarios where only errors up to a few cents on the dollar amount may be acceptable, using the absolute error difference as the metric is the right choice. Perhaps, a combination of the two maybe the most appropriate metric in certain contexts.

Aggregating Error: Once an error metric is chosen for part of the entire output (the price of an individual option for Blackscholes), we then need to determine how to combine the errors for all parts of the output (all option prices in the considered portfolio for Blackscholes) to give the aggregate error for the considered approximation technique. For Blackscholes, with a large portfolio of options, calculating the average error across all option prices can undermine significant and often unacceptable errors in individual option prices. In this case, considering both the average and the maximum individual error may be the best strategy in determining the acceptability of the final (approximate) output.

Error Threshold: The final step is choosing an acceptable threshold to bound errors in the benchmark outputs. Often, an error threshold of around 10% is indiscriminately used [1, 2, 4, 5, 6], but this threshold may be too large to be practical in many real world scenarios. In Blackscholes, for example, 10% error in a $20 option is $2, which may be an unacceptable error margin for many financial transactions. Even a 3-5% error threshold may be too much for some types of applications. While there are techniques that use lower error bounds [7, 8], we need to be much more aggressive as a community in defining what is an acceptable error. Erring on the side of caution and tightening error margins as much as possible might be a good strategy for many applications but it might be needlessly conservative for others. Selection of these parameters is currently largely ad-hoc. We need systematic domain studies to standardize acceptable error thresholds.

In summary, there is little consensus in what is the right approach to evaluate the accuracy loss for a given application. Blackscholes has simple numerical outputs and a relatively straightforward usage model; it still poses several questions on which error metrics, error aggregation tech-

[*]This work was supported in part by the National Science Foundation under Grant CCF-1320941 and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

niques, and error thresholds to use. These questions are harder to answer in other applications that may utilize sensory inputs and outputs such as audio and video. While these applications have a lot of potential for approximate computing, gauging and bounding errors in these is even more difficult without understanding how the final outputs are used. Hence, it is imperative that an agreement is reached by the community in defining benchmarks and standardizing the methodology to evaluate the errors in their output.

Programmer Expertise and Automation

Widespread adoption of approximate computing is hampered by its lack of application to a wide variety of programs and users. A significant reason for this is the need for expert programmer knowledge to distinguish locations in the application which may be amenable to approximation versus those that need precise computation [1, 9, 10]. As a result, many potential opportunities for approximation may be overlooked due to insufficient or inaccurate application information. While minimal programmer input such as end-to-end error metrics for the final application output and the acceptable error thresholds are necessary, it is unreasonable to expect the programmer to provide details at the instruction and data level for approximation. Relieving the programmer of this burden will open up many new programs to the potential benefits of approximations. There is a need for automated end-to-end frameworks which can be used by even a novice programmer to find and exploit approximation opportunities (if any) in an application.

Recently, there has been an emergence of approximate computing frameworks [11, 12, 13] that allow programmers to specify high level constraints on kernel outputs and automatically generate approximate programs that can run on given approximate hardware with known hardware reliability specifications while providing guarantees on error bounds [11, 12]. These end-to-end techniques are a promising step towards generating automated frameworks but require some expertise from the programmer in identifying approximate functions/kernels in a large application and burden them with a new programming model.

We are developing a tool called Approxilyzer that can be used by novice programmers to gauge the *first order approximation potential* of their application with no program annotations or modifications. Approxilyzer analyzes a program to provide the programmer with a set of instructions that *may* be candidates for approximation. Approxilyzer does this by eliminating instructions that generate unacceptable errors (either catastrophic or with error magnitude larger than what is deemed acceptable by the user) in the presence of single bit errors. The underlying argument that Approxilyzer makes is that if an instruction produces an unacceptable quality output in the presence of single bit corruptions, then it is highly unlikely to generate an output of acceptable accuracy with more vigorous perturbations introduced by approximation. After eliminating the instructions that produce unacceptable outputs, the remaining instructions are identified as potential first order candidates for approximation that can be further analyzed by the programmer using other software or hardware techniques.

The Approxilyzer framework extends the resiliency tool Relyzer [14] towards application in approximate computing. Relyzer uses a combination of fault injections and program analysis to determine the outcome when a single-bit fault is injected in a given bit pertaining to a given dynamic instruction (referred to as a fault site) in an execution. Using program analysis and some heuristics, Relyzer identifies fault sites that behave similarly in the presence of faults and groups these together into an equivalence class. It then performs a fault injection experiment on just one representative fault site and uses its result to predict the outcome for all the fault sites in the equivalence class. Hence, Relyzer is able to predict the resiliency characteristics of virtually all the fault sites in the application with relatively few fault injection experiments. The outcomes predicted by Relyzer include whether the fault will be masked, detected, or produce a Silent Data Corruption (SDC). Approxilyzer expands upon Relyzer by introducing the notion of *quality* to the SDCs based on how far the corrupted output deviates from the fault free execution's output. It estimates, with high confidence, the range of program output errors caused by each individual instruction in the application. Armed with the information, Approxilyzer can now automatically identify instructions that may be amenable to approximation based on their effect on the program output. Using Approxilyzer, the exploration space of instructions to consider for approximation can be significantly reduced and it can help programmers identify code segments for more targeted experiments.

We envision a tool like Approxilyzer being an intrinsic part of the front-end of a larger systematic and automated framework for approximate computing that will incorporate a single work-flow – from analyzing an unknown application for approximation opportunities to exploiting them by using the most optimal technique (software or hardware) to satisfy given accuracy constraints. There are many questions that will arise in the process of building this workflow, some of which are as follows – What role will the compiler or runtime play in such a framework? What is the best axis of approximation – instruction or data – and what is the right granularity at which to consider instructions and/or data? How do we deal with input dependency? How do we make individual techniques modular so they can be integrated into a single workflow? We urge the community to strive towards building such systematic end-to-end frameworks while working on individual techniques that we hope will serve as essential building blocks in this larger goal.

Conclusion

As the field of approximate computing matures, we as a community need to address some key concerns that are plaguing it. Only by standardizing benchmarks, refining error evaluation methodologies, and building systematic frameworks that make it easy for programmers of all skill levels to automatically apply approximation techniques can we further the field to have widespread and lasting impact. Without these we are doing this much promising field a disservice that will prevent approximate computing from truly achieving its potential. One thing is clear – while our techniques to achieve performance and energy efficiency can be approximate, the tools and methodology used to apply and evaluate them cannot.

REFERENCES

[1] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, “Enerj: Approximate data types for safe and general low-power computation,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, 2011.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Neural acceleration for general-purpose approximate programs,” in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 449–460, 2012.

[3] U. R. K. Ismail Akturk, Karen Khatamifard, “On quantification of accuracy loss in approximate computing,” in *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2015.

[4] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, “Sage: Self-tuning approximation for graphics engines,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[5] J. S. Miguel, M. Badr, and N. E. Jerger, “Load value approximation,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[6] J. San Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, “Doppelganger: A cache for approximate computing,” in *International Symposium on Microarchitecture*, 2015.

[7] D. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, “Rumba: An online quality management system for approximate computing,” in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pp. 554–566, June 2015.

[8] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, “Neural acceleration for gpu throughput processors,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[9] M. Carbin, S. Misailovic, and M. C. Rinard, “Verifying quantitative reliability for programs that execute on unreliable hardware,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA ’13, 2013.

[10] J. Bornholt, T. Mytkowicz, and K. S. McKinley, “Uncertain: A first-order type for uncertain data,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’14, 2014.

[11] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, “Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels,” *SIGPLAN Not.*, vol. 49, pp. 309–328, Oct. 2014.

[12] J. Park, X. Zhang, K. Ni, H. Esmaeilzadeh, and M. Naik, “Expax: A framework for automating approximate programming,” in *Technical Report, Georgia Institute of Technology*, 2014.

[13] B. Boston, A. Sampson, D. Grossman, and L. Ceze, “Probability type inference for flexible approximate programming,” vol. 50, (New York, NY, USA), pp. 470–487, ACM, Oct. 2015.

[14] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, “Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults,” in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.