Leveraging Software Testing to Explore Input Dependence for Approximate Computing[*]

Abdulrahman Mahmoud Radha Venkatagiri Khalique Ahmed Sarita V. Adve
Darko Marinov Sasa Misailovic
University of Illinois at Urbana-Champaign
{amahmou2, venktgr2, kahmed10, sadve, marinov, misailo}@illinois.edu

With the end of Moore's law and the shift towards alternative paradigms for energy and performance gains, approximate computing has gained considerable traction in recent years. Approximate computing environments obtain gains in performance and/or energy by trading off computational accuracy. The gains afforded by many approximation techniques are intimately coupled with the inputs provided to the program. Hence, reasoning about inputs in a systematic way is an important and open research problem. In this paper, we discuss some concerns associated with input dependence and the need to elevate program inputs to first-class citizens within the domain of approximate computing. We leverage the mature field of software testing for mechanisms and techniques to analyze program inputs and measure their influence on approximations. We believe that, going forward, software testing methodologies will become an integral part of the workflow for approximate computing-based systems.

1. THE ISSUE OF INPUT DEPENDENCE

Recent projects have illustrated a range of promising techniques in both hardware and software for utilizing approximate computing across the stack [1, 2, 3, 4, 5, 6, 7, 8]. Many of these techniques, however, use empirical evaluation methodology with a specific set of inputs that may not be fully representative of the entire application domain. Consequently, it may be difficult to generalize the effectiveness of such techniques across the whole application domain. As a result, a technique may overfit the specific input data, or alternatively, be too conservative and fail to benefit from additional approximation opportunities.

At the other end of the spectrum, when data is prevalent, one can employ machine learning algorithms to learn about an application's behavior and amenability to approximation [9, 10, 11]. Again, this can be extremely beneficial, but also suffers from a variety of issues, including obtaining access to huge data sets, time-consuming computations in the learning phase, as well as various challenges in understanding the features and characteristics of an input set.

Some past projects have explored input-aware approximation techniques which elevate the reasoning about approximation to the programmer; Others have tried tuning their choice of approximation based on the current program input [2, 12, 13]. Although adapting to inputs is a good step towards input-independent approximations, understanding why certain inputs lend themselves to specific approximations would open the door for new techniques that do not need to reactively be tuned to the input.

The difficulty with input dependence arises in the fact that it is prohibitively expensive (if not intractable) to explore the entire input space of an application in order to provide guarantees. Fortunately, this is not a new problem in computer science, and has been explored in depth by the software testing community [14]. Software testing literature has extensively dealt with providing quality assurance, verification and validation, and reliability estimation for software. Thus, the approximate computing community can leverage decades of research in this field to better reason about input dependence and quality guarantees.

In this paper, we leverage software testing techniques for analyzing input-dependence in an approximate-computing framework. The presented parallels from software testing to approximate computing are just an example corollary, and we invite further exploration down this path to help propel current approximate computing research.

2. SOFTWARE TESTING AND APPROXIMATION

One facet of the software testing framework is to decide whether code is "good enough" to deploy. Many bugs exist in the tools and applications a programmer or client uses. However, at some point in the pipeline, a software developer must decide that the code is good enough for release. The analogy to approximate computing naturally follows, in that we desire to also provide strong enough guarantees on approximation techniques and tools, while understanding that the user has a variety of knobs at her disposal.

The software testing community has developed different metrics, techniques, and tools to decide what "good

[*]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.

enough" means quantitatively and to improve various other aspects of testing. Mutation testing [15], regression testing [16], test prioritization, test selection, test minimization, and coverage analysis [17] are just some of the approaches aimed at improving testing. We explore a few techniques in this paper targeting input dependence, and how these concepts relate to approximate computing.

2.1 Test-Input Generation

Identifying which inputs trigger certain scenarios in the code can be extremely beneficial in the approximate computing realm. Different code segments may exhibit a certain affinity to some approximation tactics, while others are averse to any approximation, and both these segments can swap roles based on a different input.

Given a piece of code, test-input generation automatically finds inputs that lead to some execution point. One technique that can be of value to approximate computing researchers is that of symbolic execution. Rather then executing the program with concrete values, symbolic execution executes the program with symbolic expressions, resulting in a general and input-insensitive analysis of the software.

A major benefit of symbolic execution is that it can find concrete inputs that together span a large part of the input domain. Hence, it can provide a software engineer with a minimal set of inputs needed to stress a program, solving the issue of a large input space.

Unfortunately, a symbolic execution tree usually grows exponentially in size, and is very computationally expensive to generate. Further, most tools available today (such as KLEE [18]) are not widely used in real-world practice, and are only used in research domains.

Generating representative inputs is imperative in assessing an approximation technique and providing quality guarantees. Using tools that automatically generate and reason about inputs would greatly alleviate the issues of input dependence in approximate computing.

2.2 Coverage Criteria

Whether an input set was generated automatically (e.g. via symbolic execution) or manually, it is important to assess how representative the input set really is of the input space. Coverage analysis can be used in tandem with test-input generation to measure the coverage of a given program and its test inputs, provided a coverage criterion.

Formally, coverage criteria are defined as a rule or requirement which a test suite needs to satisfy, with many coverage criteria existing in the literature [14]. Some coverage criteria, such as branch or statement coverage, are more readily used due to their simplicity and feasibility in implementation. Common tools such as gcov and lcov are widely available for use today [19]. Others, such as path coverage, are less common due to cost or intractability despite being more comprehensive in studying inputs.

By incorporating coverage criteria in the analysis of an approximation technique, one can reason more about the quality guarantees needed by an application. For example, in real-life scenarios, most commercial code can simply be vetted with statement coverage. However, in the situation of mission-critical code, such as airliner-related software, the coverage criterion may differ (to be more stringent than simple statement coverage), and the actual coverage percentage would need to be higher. Similarly, there is a range of applications between these two extremes where an appropriate coverage criterion may be utilized with acceptable results.

3. BRIDGING THE GAP WITH TOOLS

For true wide-scale adoption of approximate computing across the stack, there is a need for automated tools that provide quality assurance to the end user. Incorporating the software testing tools already in use into the approximation work flow would only strengthen the gains shown by approximate computing in performance and energy savings. This gap needs to be tightened to ensure guarantees and allow for full acceptance by software developers.

For example, we recently released an open-source tool called Approxilyzer[1] [20] which can quantify the quality impact of a single-bit error in virtually all dynamic instructions of an application. Approxilyzer functions at the instruction level, and requires very little programmer involvement to profile an application and extract instructions that exhibit first-order approximation potential for a given input.

As instructions are the abstraction level a machine understands, we are currently exploring a new coverage criterion called *PC coverage*, which is analogous to statement coverage at the source code level. As a result, rather than using Approxilyzer with a large input set as typically provided in benchmark suites, we can target just a small set of inputs which together provide us with close to 100% PC coverage.

Preliminary results show that running smaller input sets with Approxilyzer, we can get up to a 7x reduction in total error injections necessary when Approxilyzer is used for resiliency analysis. Additionally, the smaller input set can match the classification of error injections for the larger set with 99% accuracy for some applications. Further, it can do this faster due to the reduction in simulation run-time per injection, and the fewer number of total injections.

Most importantly however, this provides us with a systematic methodology as to how to choose our inputs, and how we can provide guarantees with our tool with respect to the input we explored.

This simple example illustrates how software testing methodology can be an additional piece of the puzzle towards an automated end-to-end framework. We demonstrate here just one example of how software testing can be applied to approximate computing. Going forward, we would like to explore additional techniques from software testing for approximate computing.

[1]https://cs.illinois.edu/approxilyzer

4. REFERENCES

[1] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan 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, pages 164–174, New York, NY, USA, 2011. ACM.

[2] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[3] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[4] D.S. 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*, pages 554–566, June 2015.

[5] 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*, pages 449–460, 2012.

[6] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelganger: A cache for approximate computing. In *International Symposium on Microarchitecture*, 2015.

[7] James Bornholt, Todd Mytkowicz, and Kathryn 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.

[8] John Sartori and Rakesh Kumar. Architecting processors to allow voltage/reliability tradeoffs. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 115–124, New York, NY, USA, 2011. ACM.

[9] Adrian Sampson. The case for compulsory approximation. In *Workshop on Approximate Computing Across the Stack*, 2016.

[10] Xin Sui, Andrew Lenharth, Donald S. Fussell, and Keshav Pingali. Proactive control of approximate programs. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 607–621, New York, NY, USA, 2016. ACM.

[11] Subrata Mitra, Manish K. Gupta, Sasa Misailovic, and Saurabh Bagchi. Phase-aware optimization in approximate computing. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO 2017, pages 185–196, Piscataway, NJ, USA, 2017. IEEE Press.

[12] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[13] Michael A. Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

[14] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[15] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[16] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[17] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[19] Qian Yang, J. Jenny Li, and David Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 99–103, New York, NY, USA, 2006. ACM.

[20] Radha Venkatagiri, Abdulrahman Mahmoud, Siva Kumar Sastry Hari, and Sarita V. Adve. Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 00:1–14, 2016.