Accuracy-Aware Program Transformations

Sasa Misailovic
MIT
misailo@csail.mit.edu

1. Introduction

Program approximation has been accepted as a necessary and useful technique for solving time-consuming computational problems since the inception of computer science. However, for a long time it has been treated exclusively as an *algorithmic problem* – a domain expert, computer scientist, or a developer has been responsible for proposing, implementing, and evaluating approximate algorithms. Up to recently, most programming languages did not expose special abstractions to support approximate program development. Moreover, automated program optimizations have been exclusively *semantics-preserving* transformations – the compiler guarantees that the optimized and original programs always produce identical results.

Many emerging applications operate on noisy inputs or solve problems for which multiple solutions are possible (while some may be more accurate than the others). In addition, recent hardware design proposals provide components that can save energy in return for occasionally producing incorrect results. Traditional, semantics-preserving program optimizations are too rigid to exploit the full optimization potential of these applications. Instead, we need a novel program optimization approach that relaxes the notion of binary correctness of program transformations.

We propose *accuracy-aware* program transformations that change semantics of a program to trade accuracy for performance by exploiting the properties of program's inputs, structure, and execution environment. Automatically applying accuracy-aware transformations provides an additional opportunity to reduce developers' engineering effort, reduce resource consumption, and increase program's functionality.

However, these opportunities come at a price – the transformations introduce uncertainty into the computation and can generate many alternative programs with different tradeoffs. A key prerequisite to utilizing this potential is responding to the two challenges: 1) to characterize the effects that a transformation has on a program and 2) to automatically discover transformations that provide maximum performance gains for an acceptable accuracy loss.

The goal of our research is to provide developers and users with optimization techniques to automatically generate approximate programs that deliver profitable, safe, and predictable tradeoffs between accuracy and performance. This paper presents several transformation and reasoning techniques that are starting points toward that goal.

2. Accuracy-aware Program Transformations

Accuracy-aware program transformations intentionally change the semantics of programs to trade accuracy for performance. Each of these transformations targets a specific language construct or a computational pattern that can be found in many programs. Many accuracy-aware transformations are configurable – their parameters can control how aggressive the approximation should be.

Discarding Inputs and Computation. These transformations cause the programs to do less work by skipping some of its inputs or interrupting the computation. Sampling selects only a subset of inputs from typical reduction operator like summation or maximization [6, 13]. Loop perforation [9, 12] skips iterations of a `for`-style loop. This transformation sometimes corresponds to sampling, but in general case it generates a new implementation of the target function with variable accuracy. Task skipping [10, 11] relaxes the requirement to wait for all threads at synchronization barriers. Once the majority of threads finish their computation, the transformed program interrupts the remaining threads.

Variable-Accuracy Computation. Randomized function substitution replaces a call to a fully accurate function to a random choice between the calls to one of the less accurate function implementations [6, 13]. A deterministic version of function substitution and precision scaling have also been explored by other researchers.

Exploiting Execution Environment. Some transformations exploit the inherent randomness or unreliability of the program's execution environment. Placement of unreliable arithmetic operations [4] in programs written in Rely [2] depends on the probability that the underlying hardware produces a correct result for an operation such as addition or multiplication. Lock elision (which removes lock synchronization) trades the accuracy of shared variables for time saved by avoiding synchronization [10]. The effect of this transformation depends both on the load of the system and the operating system scheduler.

3. Reasoning About Transformed Programs

In general, program optimization with accuracy-aware transformations becomes a decision problem under uncertainty. A key prerequisite to successfully applying the accuracy-aware transformations is understanding the effect that a transformation has on the program's result. Program analyses for these transformations should provide a developer with arguments to answer the question:

> *"Will the benefits of improving the program's performance and energy consumption outweigh the cost of decreasing the accuracy of its results?"*

One way to answer this question is to design 1) accuracy analyses that express accuracy cost and 2) search algorithms that explore the tradeoff space for programs that yield maximum performance or energy benefits for acceptable accuracy costs.

3.1 Accuracy Analysis

Accuracy analyses quantify the deviation of the result of the transformed computation from the result of the original computation. Accuracy analyses may have different precision and generality, which are often related to the kinds of computations they analyze.

Empirical Reasoning. Empirical testing executes an approximate program on a set of representative inputs. While it gives the most precise result (the exact accuracy loss) and works for large programs, it does not generalize to any input beyond the tested inputs. In our previous work, empirical evaluation of accuracy-aware transformations provided us with an initial understanding of the phenomena and helped us assess the usefulness of the transformations [3, 9, 12].

Worst-Case Reasoning. Calculating worst-case error bound is on the opposite side of the analysis expressiveness spectrum. The result of this analysis generalizes for all inputs, but for many inputs its prediction is overly pessimistic. Yet, the worst case error gives us one useful way of understanding the uncertainty of the computation: the error can (and sometimes will) be quite large; a developer should be comfortable with program infrequently producing large, even the worst case errors.

Probabilistic Reasoning. Probabilistic bounds complement the worst case bounds by providing limits on both the *frequency* and *magnitude* of errors. In our previous work, we have studied error bounds of the form $\Pr[\,|D| < B\,] > 1 - \delta$, where B is the acceptable bound on the error magnitude D, and δ is the frequency of unacceptable errors. B and δ are provided by the user. We studied the cases when the randomness may come from inputs [8] or be inherently present in the computation [6, 13]. Presently, these analyses require a specific form of an approximate computation.

Statistical Reasoning. In some situations, it may be necessary to execute programs on real inputs, but still produce confidence bounds on the result of the analysis (e.g., to quantify randomness from the computation or environment). For instance, we have used statistical hypothesis testing to characterize the frequency of large errors [10]. We have also used distribution fitting to estimate the magnitude of error of perforated computations [7].

3.2 Search for Approximate Programs

The optimization algorithm searches for the transformed approximate programs that can be executed in a minimum amount of time or consume minimal energy. It starts from a fully accurate program. To automatically discover approximate programs, we define a three stage *find-analyze-navigate* optimization approach:

- **Find.** In this stage, the optimization algorithm identifies subcomputations that may be good approximation candidates through profiling.
- **Analyze.** In this stage, the optimization algorithm analyzes performance, accuracy loss, and safety of a transformed program location. To test accuracy, a developer specifies an accuracy loss metric, which is a relational accuracy predicate (the expected value of an error metric or probability of large errors) and acceptable error bound. To test safety of the transformed program, the algorithm uses either dynamic criticality testing [3, 9, 11, 12], which checks whether the computation had unwanted behavior on test inputs, or static verification of computation's integrity properties [1].
- **Navigate.** In this stage, the optimization algorithm navigates the accuracy/performance tradeoff space to 1) find combinations of transformations that yield maximum performance savings and 2) construct a tradeoff curve containing the programs that deliver most profitable tradeoffs for different error bounds.

The optimization produces a set of transformed programs together with their accuracy and performance characteristics. These transformed programs allow a developer or a user to tune the application's accuracy to different levels.

We have proposed two instances of the find-analyze-navigate approach for exploring the tradeoff space:

Explicit Search. Explicit search finds acceptable transformations by executing the transformed program on a set of user-provided representative inputs [3, 5, 9, 10, 12]. It uses a user defined error metric and empirical or statistical analysis to characterize the output's accuracy losses. While the result of the analysis is valid only for those inputs and outputs, in practice, the results of these analyses and search typically carry over to similar inputs.

Mathematical Optimization. This approach statically analyzes the program and constructs error and performance expressions parameterized by variables that denote whether the program transformation should be applied. The examples of mathematical optimization approaches are [6, 13] and [4].

These techniques use probabilistic error analyses to construct the error expressions. Therefore, unlike the explicit search, the results of this search are valid for a whole class of inputs – typically all possible inputs or inputs that fall within a certain range. However, this more powerful accuracy property comes at a price that a computation must have a special structure [6, 13] or check for a weaker probabilistic property, e.g. error frequency [4].

4. Conclusion

Automated accuracy-aware optimizations are going to change many phases of software development process, including software specification, profiling, testing, and evolution. To make any of these approaches succeed, we need to empower a developer – through education, more powerful reasoning techniques, and tools – with understanding or good intuition of the ways in which input, computation, and environment randomness can affect their programs.

Accuracy-aware optimizations also have the potential to change the software reuse practices. A standard development goal for many libraries (and other reusable components) is to prepare the libraries to operate under most adverse conditions, which may be overly conservative. A developer of a client application, who may not have an expertise or interest to manually modify library code, may use the automated optimization with accuracy-aware transformations to generate less accurate library code implementations and thus specialize the library code for their purposes.

Acknowledgements. The parts of the work presented in this paper are done in collaboration with Martin Rinard and Michael Carbin, Henry Hoffmann, Jonathan Kelner, Deokhwan Kim, Daniel Roy, Stelios Sidiroglou, and Zeyuan Allan Zhu.

References

[1] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. PLDI, 2012.

[2] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. OOPSLA'13.

[3] H. Hoffman, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS, 2011.

[4] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Reliability-aware optimization of approximate computational kernels with rely. Technical Report MIT-CSAIL-TR-2014-001, MIT, 2014.

[5] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM TECS PEC*, 2013.

[6] S. Misailovic and M. Rinard. Synthesis of randomized accuracy-aware map-fold programs. Technical Report MIT-CSAIL-TR-2013-031, MIT, 2013.

[7] S. Misailovic, D. Roy, and M. Rinard. Probabilistic and Statistical Analysis of Perforated Patterns. Technical Report MIT-CSAIL-TR-2011-003, MIT, 2011.

[8] S. Misailovic, D. Roy, and M. Rinard. Probabilistically Accurate Program Transformations. SAS, 2011.

[9] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. ICSE, 2010.

[10] S. Misailovic, S. Sidiroglou, and M. Rinard. Dancing with uncertainty. In *RACES*, 2012.

[11] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. ICS, 2006.

[12] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE 2011.

[13] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. POPL, 2012.