Approximate Program Synthesis

James Bornholt Emina Torlak Luis Ceze Dan Grossman
University of Washington
{bornholt, emina, luisceze, djg}@cs.washington.edu

Abstract

Existing programming models for approximate computing apply to only restricted classes of programs and approximations, or require special-purpose hardware. A programming model that lifts these restrictions while being accessible to programmers is a central unsolved challenge in the field.

This paper proposes the use of *program synthesis* to automate approximate programming. Program synthesis automatically produces a program that meets a desired correctness specification, and has seen success in a number of application domains. We formalize the *approximate synthesis* (AppSyn) problem as the task of finding an optimal program that is approximately correct, and show how existing approximation techniques fit into this framework. We show how to solve AppSyn problems efficiently with sketch-based program synthesis. The results, which show novel approximations to benchmarks from the literature, suggest that synthesis is a promising approach to approximate computing.

1. Program Synthesis

Program synthesis is the task of automatically producing a program that meets a desired correctness specification. Search-based synthesis [1, 5, 6, 11, 15, 16] addresses this problem by searching for a correct program in a space of candidate implementations defined by a syntactic template (e.g., a context-free grammar [1]). Search-based synthesis has seen success as a programming model in a variety of domains, including low-power spatial computing [10], bulk-synchronous distributed programming [18], browser layout engines [7], and cache coherence protocols [16].

A program synthesizer takes as input a specification, which could be a complete formal specification in some logic, a set of input-output examples, or even a complete reference implementation. It then searches a space of candidate implementations for a program that meets this specification. For example, if the input specification is a logical formula, a synthesizer might use an SMT solver to try to solve for a program implementation that satisfies the specification over all inputs [5]. Many different techniques exist to explore the candidate space effectively. The key productivity benefit of program synthesis is that it abstracts the *what* from the *how*: the programmer specifies what the program should do, and the synthesizer discovers how the program can do it.

2. Approximate Program Synthesis

Approximate computing presents a difficult programming problem. Fresh from having implemented a complex program under familiar, precise semantics, approximate computing asks the programmer to now relax the program's behavior to make it less accurate but more efficient.

Program synthesis is a potential solution to this programming challenge. Synthesis is a good fit for approximate computing because the programmer's precise implementation can serve as the specification for synthesis. In principle, the programmer need only provide the reference and a desired accuracy bound, and the synthesizer will discover an approximate implementation that satisfies this bound. However, the synthesizer must also be able to validate that the resulting program will be more efficient than the reference.

We formalize this notion as the *approximate synthesis* (AppSyn) problem. Approximate synthesis (Def. 1) searches a space of candidate programs L_P for the lowest-cost implementation P that acceptably approximates the behavior of a reference program S. The search is performed with respect to a correctness relation ϕ and a cost function κ.

DEFINITION 1 (Approximate Synthesis). *Let L be a programming language; $[\![P]\!]$ denote the function defined by a program $P \in L$; and $[\![L]\!]$ the set $\{[\![P]\!] \mid P \in L\}$ of all functions representable by L. Given a reference program $S \in L$, a correctness relation $\phi \subseteq [\![L]\!] \times [\![L]\!]$, a cost function $\kappa : L \to [\![L]\!] \to \mathbb{R}$, and a set of candidate programs $L_P \subseteq L$, the approximate synthesis (AppSyn) problem is to find a program $P \in L_P$ such that $\phi([\![P]\!], [\![S]\!])$ holds and $\kappa(P, [\![P]\!])$ is minimal.*

The correctness relation ϕ constrains the semantics of the synthesized program P with respect to the reference implementation S. It therefore controls how the program is approximate. For example, a programmer could require that all approximate outputs are within 5% of the corresponding precise output, or that outputs of the approximate program are precise with some probability p. The cost function κ ensures that approximate synthesis produces an efficient program.

Existing Techniques as AppSyn Problems. Chisel [8] approximates computational kernels by replacing instructions with more efficient approximate versions, while satisfying a programmer-specified accuracy and reliability constraint.

The programmer-provided constraint form the correctness relation ϕ, and Chisel's energy model is the cost function κ. Chisel finds the most efficient program satisfying the quality constraints by solving an integer linear programming problem, which can be seen as a form of synthesis.

Similarly, loop perforation [12] transforms computationally intensive loops to skip some iterations. A loop perforator takes a reference implementation and explores a space of candidate programs defined by trying different loop strides and bounds. The search obeys some desired quality constraint ϕ and generally measures the cost κ by executing the candidate program on a test suite. Many other approximation techniques, including coarse-grained approximations such as neural acceleration [4], also fit the AppSyn framework.

3. Feasibility of Approximate Synthesis

We conducted a series of experiments to determine if existing program synthesizers can solve approximate synthesis problems. We used three state-of-the-art solvers from the *syntax-guided synthesis* (SyGuS) [1, 2] competition: a brute-force enumeration of candidate programs [16], a stochastic search [11], and an SMT solver-based synthesizer [5, 6].

We defined the HD benchmark suite, consisting of 15 problems from the "Hacker's Delight" problem set [17], a standard program synthesis benchmark. We combined these 15 problems with several relaxed correctness specifications.

We found that approximation does not make program synthesis less tractable: the SyGuS solvers were able to solve approximate and precise versions of these benchmarks in similar time. In fact, we found that even though the approximate versions allowed relaxed correctness, the SyGuS solvers often returned *exact* solutions *faster* than exact solvers. While surprising, we validated that this result was due to interactions with the heuristics of the underlying solver, rather than a fundamental result about approximate synthesis.

4. Real-World Approximate Programs

We defined the PARROT benchmark suite, consisting of seven programs from the neural acceleration work of Esmaeilzadeh et al. [4]. Unfortunately, SyGuS solvers were unable to solve any of these approximate synthesis problems, timing out after one hour of work. The PARROT problems are substantially larger than the HD problems, and so the search space of candidate programs is too large to easily explore.

To make approximate synthesis tractable for real problems, we turned to *sketch-based synthesis* [13, 14], which is the problem of completing a partial implementation—a *sketch*—to satisfy a correctness specification. A sketch is a program with missing expressions, called *holes*, to be discovered by the synthesizer. The sketch restricts the search space of candidate programs by conveying domain knowledge to the synthesizer. Sketch-based synthesis is known in the synthesis community to improve the scalability of program synthesis.

```
1  static int kx[3][3] =
2    {{ −1, −2, −1},
3     { 0, 0, 0},
4     { 1, 2, 1}};
5  int sobel_x(int w[3][3]) {
6    int i, j, r = 0;
7    for (j = 0; j < 3; j++)
8      for (i = 0; i < 3; i++)
9        r += w[i][j] * kx[j][i];
10   return r;
11 }
```

(a) sobel_x reference implementation

```
1  ??op ∈ {+, −, <<}
2
3  int sobel_x_sk(int w[9]) {
4    int c0 = ??op(??var, ??var);
5    int c1 = ??op(??var, ??var);
6    int c2 = ??op(??var, ??var);
7    int c3 = ??op(??var, ??var);
8    int c4 = ??op(??var, ??var);
9    int c5 = ??op(??var, ??var);
10   return c5;
11 }
```

(b) sobel_x RIS sketch

Figure 1. Reduced Instruction Set (RIS) sketch for the sobel_x problem.

Problem	RIS		BDF		PF	
	Speedup	Error	Speedup	Error	Speedup	Error
fft_s	–	–	–	–	11.4×	21.3%
fft_c	–	–	–	–	12.0×	28.9%
dist3	–	–	1.6×	14.9%	–	–
sobel_x	10.6×	0%	–	–	–	–
sobel_y	10.7×	0%	–	–	–	–
$\mathsf{inversek2j}_1$	–	–	–	–	34.8×	16.3%
$\mathsf{inversek2j}_2$	–	–	–	–	10.0×	18.5%

Figure 2. Sketch-based synthesis successfully solves real-world AppSyn problems.

We defined a class of sketches called *reduced instruction set* (RIS) sketches. RIS sketches are SSA-form programs, in which each assignment applies an unspecified operator to unspecified operands, as in Figure 1. The operator holes $??_{op}$ are constrained to a reduced instruction set necessary to implement the reference program (as determined automatically). RIS sketches are parameterized by their length and by how many additional operations are added to the reduced instruction set (which might allow novel approximations). We also define bounded data-flow (BDF) sketches: RIS sketches that restrict argument holes to only read from variables read by that instruction in the reference implementation. Finally, we define piecewise function (PF) sketches that approximate a function by a piecewise combination of polynomials.

Figure 2 shows the results of using these sketches to solve PARROT problems, with a correctness specification that approximate outputs are within 50% of precise outputs. We successfully solve all seven problems, discovering more efficient programs with reasonable accuracies.

5. Conclusion and Future Work

Programming models for approximate computing require reasoning about delicate trade-offs between quality and accuracy. Our results show that program synthesis is a promising technique to automatically explore this trade-off. In future work, we intend to automate the solving process, by searching for the optimal sketch for a given reference program. We also intend to explore richer cost functions and optimization criteria, motivated by recent work on synthesis with real-valued optimization functions [3] and of probabilistic programs [9].

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[2] R. Alur, R. Bodik, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Marktoberdorf*, 2014.

[3] S. Chaudhuri, M. Clochard, and A. Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In *POPL*, 2014.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[5] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[6] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[7] L. Meyerovich. *Parallel Layout Engines: Synthesis and Optimization of Tree Traversals*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2013. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-242.html`.

[8] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

[9] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. In *PLDI*, 2015.

[10] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[11] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

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

[13] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[14] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, 2007.

[15] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[16] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. Transit: Specifying protocols with concolic snippets. In *PLDI*, 2013.

[17] H. S. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2007.

[18] Z. Xu, S. Kamil, and A. Solar-Lezama. Msl: A synthesis enabled language for distributed implementations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 311–322, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.31. URL `http://dx.doi.org/10.1109/SC.2014.31`.