Synthesis of Randomized Accuracy-Aware Map-Fold Programs

Sasa Misailovic Martin Rinard
MIT CSAIL
misailo@csail.mit.edu rinard@csail.mit.edu

Abstract

We present Syndy, a technique for automatically synthesizing randomized map/fold computations that trade accuracy for performance. Given a specification of a fully accurate computation, Syndy generates approximate implementations of map and fold tasks, explores the space of approximate computations that these approximations induce, and derives an accuracy versus performance tradeoff curve that characterizes this space. Each point on the tradeoff curve is associated with an approximate program configuration that satisfies the probabilistic error and time bounds of that point.

1. Introduction

Many computations exhibit inherent tradeoffs between the accuracy of the results they produce and the time required to produce these results. Examples include audio and video processing applications, numerical computations, machine learning, and search applications. A key aspect in tuning these applications is finding alternative implementations of the program's subcomputations and configurations of these subcomputations that yield profitable tradeoffs between accuracy and performance.

Researchers have previously proposed several language-based techniques for navigating the tradeoff space and identifying or synthesizing alternative versions of subcomputations that help deliver desired tradeoffs. Many of these approaches are empirical in nature – to find profitable tradeoffs they execute programs on a set of representative training inputs and use this information to make decisions when faced with (previously unseen) production inputs [1, 2, 11, 14, 16, 17, 19]. In contrast to these dynamic approaches, static techniques [5, 8, 13, 21] allow a user to specify the properties of program inputs (e.g., specific ranges or probability distributions of inputs) and computations (e.g., continuity), which can be used to guide the optimization of the program. Out of these static approaches, only the analysis presented in [21] is used to reason about accuracy versus performance tradeoffs.

Syndy. This paper presents Syndy, a technique for accuracy-aware optimization of approximate map-fold computations. Syndy takes as input a program that implements a fully accurate map-fold computation, a set of intervals of the inputs of this computation, and a set of alternative implementations of the program's subcomputations. Syndy's output is a transformed, randomized program that can execute at a number of points in the underlying accuracy versus performance tradeoff space. This tradeoff space is induced by the transformations – subcomputation substitution and sampling of operands of fold operators – that Syndy applies on the original program.

Syndy uses an optimization algorithm we presented previously in [21] to explore the tradeoff space and construct an optimal *tradeoff curve* of the approximate program. This optimization algorithm represents map-fold programs as graphs in the abstract model of computation defined in [21]. The algorithm then constructs the tradeoff curve by defining and solving mathematical optimization problems that characterize the program's expected execution time and expected absolute error. Each point on the tradeoff curve contains a set of randomized program configurations that deliver the specified accuracy/performance tradeoff.

Contributions. This paper presents a programming language for map-fold computations and an optimization algorithm that operates on programs written in this language. Specifically, in comparison with the previous research [21], this paper makes the following contributions:

- **Language.** It presents a language for expressing map-fold computations ([21], in contrast, worked with an abstract model of computation based on graphs).
- **Synthesis of Alternate Computations.** It presents program transformations within the language of map-fold computations that induce the underlying accuracy versus performance tradeoff space ([21], in contrast, worked directly on the graphs from the abstract model of computation).
- **Mapping.** It presents a mapping from the language of map-fold computations to the abstract model of computation from [21]. This mapping enables the use of the optimization framework to find optimal tradeoffs for map-fold programs.

2. Example

Figure 1 presents an example map-fold program. The program takes as input features of images (such as edges of objects in the image and the foreground/background data) and a set of models that represent the position of a person in the

```
function ImageEdge(image, model);
function ImageInside(image, model);
function Exp(val);
function Max(val1, val2);

function Score(image, model) :=
    Exp(-1*((ImageEdge(image, model)
        + ImageInside(image, model)))

program (images, models) :=
    fold(0, Max,
        map (Score,
            zip(images, models)))
```

Figure 1: Example Program

```
SpecF(ImageEdge) := (
    { (IEdge, 0, T_e0), (IEdge1, E_e1, T_e1),
      (IEdge2, E_e2, T_e2), (IEdge3, E_e3, T_e3)
    }, (NA, NA))
SpecF(ImageInside) := (
    { (IInside, 0, T_i0), (IInside1, E_i1, T_i1),
      (IInside2, E_i2, T_i2), (IInside3, E_i3, T_i3)
    }, (NA, NA))
SpecF(Exp) := ({ (Exp, 0, T_exp) }, 1)
SpecF(Max) := ({ (Max, 0, T_max) }, (1,1))

SpecI(images) := (400)
SpecI(models) := (400)
```

Figure 2: Function And Input Specifications

image. For each model the program computes a score that quantifies how well the model represents the person's position. The program returns the maximum computed score. This computation was derived from a part of the Bodytrack motion tracking application [4].

Functions. Syndy allows specifying two kinds of functions – primitive and composite. *Primitive functions* are defined in an external programming language and can use arbitrary complex language constructs. Syndy treats the primitive functions as black box computation. *Composite functions* are defined within Syndy's language. The body of a composite function is a Syndy expression. Composite functions can call the primitive functions and and operate on their results. Both primitive and composite functions produce numerical results.

The functions `ImageEdge` and `ImageInside` are primitive. These functions take two input parameters, `image` (image pixel information) and `model` (the model of the person's location), and produce numerical values as the outputs. The function `ImageEdge` compares the prediction of the person's location from the model with the sharpness of image object edges. The function `ImageInside` compares the body position predicted by the model with the foreground/background surfaces from the image. The result of each function is a value between 0 and 1.

The function `Score` is composite – it computes the arithmetic expression that calls three primitive functions, `Exp` (exponentiation), `ImageEdge`, and `ImageInside`. `Score` produces a result between 0 and 1. Higher value of the result indicates that the model better matches the image data.

Program Inputs. The program takes as input the list containing pointers to the raw image (`images`) and the list of models (`models`). Both lists contain *structurally complex* data structures. The map-fold computation operates on numerical data and the only operation it supports for complex data structures is passing them to primitive functions.

Main Computation. The computation uses the helper *zip* operator to combine the two input lists and construct a list of pairs of the images and models. This list is passed as input of the map operator. The map operator applies the function `Score` to each element of the input list. Since each element of the input list is a pair of values, the map operator unpacks the pair elements and passes them as arguments of the function `Score`. The map operator uses a *lazy evaluation* strategy: its output list contains the expression terms that evaluate to the score of each input parameter. These expressions are evaluated to their numerical values only when required by the subsequent computation, such as fold operators.

The fold operator computes the maximum value of the scores of all models in the input list using the built-in `Max` function. Since the input list of the fold operator contains the expressions that compute scores, these expressions are evaluated before executing the `Max` operation. `Max` then performs comparisons on numerical values. The result of the fold operator is the maximum computed score; this is also the result of the program.

2.1 Approximating Computations

This computation has several opportunities for trading accuracy for additional performance. First, the functions `ImageEdge` and `ImageInside` in the fully accurate implementation compare all pixels of the input image with the corresponding model location. Approximate implementations of these functions can sample only a subset of pixels when computing the image/model difference. Second, the maximization fold operator may skip some of its inputs, effectively searching for the maximum score of only a subset of models. If the fold operator does not aggregate scores of some models, the previous map operator's computation for these models can also be skipped. Syndy uses primitive functions and input specifications to and exploit the approximation opportunities and generate alternative program implementations.

Accuracy/Performance Specifications. The developer provides specifications of accuracy and performance of alternative function implementations. Figure 2 presents the specification of the example functions. The function **SpecF** returns the specification for each function. The first element of the specification is a set of alternative function implementation specifications. Each alternative implementation specification contains the name of function, the expected absolute error incurred by the execution of this implementation, and the expected execution time.

The function `ImageEdge` has four implementations: the original implementation (denoted as `IEdge`) does not incur any error, and it executes in time T_{e0}. The other three implementations are approximate; each of these approximate

$$
\begin{array}{llll}
n \in \text{Numeric} & \text{D} \in \text{Decl} & \to & f(x_1,\ldots,x_n) := e \mid \\
o \in \text{Opaque} & & & f(x_1,\ldots,x_n); \\
x \in \text{Vars} & \text{L} \in \text{LExp} & \to & a \mid [t_1,\ldots,t_k] \mid \mathbf{map}(f,\text{L}) \mid \\
a \in \text{ListVars} & & & [t\langle x\rangle : x \ \mathbf{in}\ 1 : n] \mid \mathbf{zip}(L_1,L_2) \\
p \in [0,1] & t & \to & \bar{e} \mid (\bar{e},t) \\
 & \mathcal{P} \in \text{Prog} & \to & \text{D}^* \ \mathbf{program}(a^*) := [\bar{e} \mid \text{L}]
\end{array}
\qquad
\begin{array}{lll}
e \in \text{Exp} & \to & x \mid e_1 \ op\ e_2 \mid e_1 \ {}_p\!\oplus\ e_2 \mid \\
 & & \mathbf{let}\ x = e_1 \ \mathbf{in}\ e_2 \mid \\
 & & f(e_1,\ldots,e_k) \mid \bar{e} \\
\bar{e} & \to & v \mid \mathbf{fold}(n,f,\text{L}) \\
v & \to & n \mid o
\end{array}
$$

Figure 3: Syndy's Target Language Syntax

implementations performs a regular sampling of the image pixels. The error that the approximate implementations incur is greater than 0, but their execution time is smaller. Both error and time specifications are numerical constants – for instance, they may have the following values: `(IEdge, 0.000, 0.174)`, `(IEdge1, 0.004, 0.097)`, `(IEdge2, 0.012, 0.059)` and `(IEdge3, 0.016, 0.051)`. The specification for `ImageInside` is similar. The functions `Exp` and `Max` have only a single, fully accurate implementation.

The second part of the accuracy/performance specification of the function is the *sensitivity* of the function's result to the changes in each of its numerical input parameters.[1] Consider the function `Max`. Its sensitivity vector has two elements (since the function has two parameters). Both sensitivity coefficients are 1, which indicates that the function `Max` does not amplify the error of its arguments. Therefore, the noise of the output of the maximum operation is proportional to the noise introduced in each of the function's arguments. The developer has specified that the sensitivity of the function `Exp` is 1. For this the developer uses the additional information that the functions `ImageEdge` and `ImageInside` produce a result between 0 and 1. Finally, the functions `ImageEdge` and `ImageInside` take as input complex data structures. Since the sensitivity is defined only for numerical parameters, the developer uses the keyword *NA* to denote that the sensitivity information is not applicable for these parameters.

Input Property Specification. The function **SpecI** specifies the properties of the inputs of the computation, such as the size of the input lists and the intervals of the input values (if applicable). The specification in Figure 2 specifies that the lists `images` and `models` have 400 elements each. For lists of numerical data, a user can also specify the intervals of the inputs. But, as the input lists in the example contain complex data structures, this part of the specification is not applicable.

3. Language

Figure 3 presents the syntax of the language of map/fold computations. A program is a set of function declarations and definitions and a program expression that consists of *map* and *fold* operators.

Data Types. The program operates on three kinds of inputs: 1) *numerical values*, 2) *opaque values*, which may be of an unspecified complex data type, but they can only be passed as arguments and processed by primitive functions, and 3) *lists* of numerical or opaque elements, which have a finite size known at the analysis time.

Probabilistic Choice Operation. The language supports probabilistic choice operator ${}_p\!\oplus$, which computes the result of the expression e_1 with probability p; it computes the result of the expression e_2 with probability $1-p$. The choice is controlled by the numerical constant $p \in [0,1]$.

Arithmetic Operations. The language supports the standard arithmetic operations, including addition, subtraction, and multiplication.

Tuples. The language defines tuples as an auxiliary data structure that allows the developer to use functions with multiple arguments within the *map* operator. It also defines the auxiliary **zip** operator, which produces a list of pairs from two lists of the same size. We define an additional syntax construct $(\bar{e}_1,\bar{e}_2,\ldots,\bar{e}_n)$ to succinctly represent the tuple $(\bar{e}_1,(\bar{e}_2,\ldots,\bar{e}_n))$ and $\mathbf{zip}_n(L_1,L_2,\ldots,L_n)$ to represent $\mathbf{zip}(L_1,\mathbf{zip}(L_2,\ldots,L_n))$.

Map Operator. A *map* operator takes as input a function f and a list of inputs. It applies the function f independently on each element of the input list L to produce the list of outputs. An input list of a map operator is either a program input or an enumerated list of the results of the previous subcomputations, $[t_1,\ldots,t_n]$.

We define a *parameterized enumerated list*, an additional language construct $[t\langle x\rangle : x \ \mathbf{in}\ 1 : n]$, to succinctly represent the enumerated list $[t'_1,\ldots,t'_n]$, in which all free occurrences of the variable x in t'_i are substituted by a constant i.

Fold Operator. A *fold* operator takes as input 1) a starting numerical value n, 1) a function f, which takes as input a temporary value of the accumulator and a single element from the list and produces the results of aggregation, and 3) a list of numerical inputs. In each step, the function f reads a single input from the list and computes the intermediate aggregate value. The output of the fold operator is a single numerical value that aggregates the contributions of all elements of the input list.

Well-Formed Program. The root expression of a well-formed program is a map or a fold operator. The program produces a numerical value. All tuples within an enumerated list in a well-formed program have the same number of elements and all lists referenced by a zip operator have the same length.

[1] Specifically, the sensitivity coefficients are Lipschitz constants of a function. For instance, for a function with one argument, S is a Lipschitz constant if $\forall x \ . \ |f(x+\delta x) - f(x)| \le S \cdot |\delta x|$. The definition of sensitivity can also be restricted for x that belongs to a closed subinterval of numbers.

3.1 Language Semantics

We note several points that have influenced the design of the language:

- ***Probabilistic Randomized Execution.*** The language supports a probabilistic choice operator, ${}_p\oplus$, which executes one of its argument expressions. We use this operator as a foundation for expressing randomized transformations – function implementation substitution and fold operator sampling – within the language.
- ***Lazy Evaluation of Map Operators.*** To support sampling of fold operators, the semantics of the language delays the execution of the computation of the map operators until their results are needed. Therefore, if some of the results of the map operator are not used by the subsequent fold computation, they are not computed.
- ***Isolation of External Computation.*** To support a broader set of computations, the language enables calling an arbitrary complex computation (written in an external programming language) that operates on individual pieces of data. However, the external computation does not have visible side effects; it only affects the program through its return value.

Transition Relation. We define the big step operational semantics for the map-fold computations. We present the details of the semantics in the extended technical report [12].

We define the probabilistic big-step evaluation relation $\mathbf{E} \overset{p}{\Longrightarrow} \mathbf{v}$, which states that the expression $\mathbf{E}$ evaluates in one or multiple steps to the final canonical expression $\mathbf{v}$ with probability p. A canonical expression $\mathbf{v}$ can be a numerical or opaque value, a tuple of values, or a list of expressions.

Expected Value of Expressions. We are specifically interested in the case when an expression e evaluates to a numerical value n: $e \overset{p}{\Longrightarrow} n$. To define the probability distribution of the results produced by the map-fold computation, we first define the set of final values and probabilities to which a closed expression term evaluates, $\mathrm{K}(e) = \{(n, p) : e \overset{p}{\Longrightarrow} n\}$. The induced probability mass function is then $\mathbb{P}_{\mathrm{K}(e)}(n) = p$. Note that the distribution is discrete under the condition that the set Numeric and the set of the opaque values are countable. We define the expected value of a numerical expression e as $\mathbb{E}[\, e \,] = \sum_{(n,p)\in \mathrm{K}(e)} p \cdot n$. It is the weighted sum of all the values that the expression can evaluate to.

Computation Error. The expectation of an expression serves as a basis for defining the error functions that represent the expected difference between the results of the original and transformed expressions. The error functions, computed for each transformation we propose, are later related to the error functions used in the optimization algorithm from [21] (see [12, Section 4]).

The error of an alternative expression $\hat{e}$ is an expected absolute difference, $\mathrm{Err}(e, \hat{e}) := \mathbb{E}[\, |e - \hat{e}| \,]$, for all inputs of the expressions e and $\hat{a}$ that fall within the ranges specified by the user. As a special case, the expected absolute error in the specification of the primitive function f with alternative implementation f' is by definition $E_{f'} := \mathbb{E}[\, |f - \hat{f}'| \,]$. The error of the alternative result of fold operators (except for maximization/minimization) is defined the same way. The error function of two numerical lists L and $\hat{L}$ (which are typically the results of map operators) is defined as the maximum absolute error of its elements, $\mathrm{Err}(L, \hat{L}) := \max_i(\mathbb{E}[\, |L_i - \hat{L}_i| \,])$.

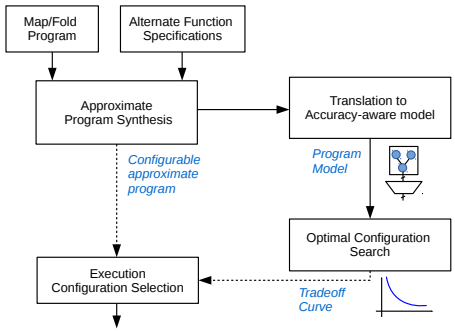

Figure 4: Synthesis Framework Overview

4. Accuracy-Aware Transformations

Figure 4 presents an overview of the framework that generates approximate map-fold programs. The framework takes as input the original map-fold computation and the function and input specifications. The framework consists of 1) the program synthesis component, which constructs an approximate configurable program from the original one, and 2) the optimal configuration search component, which computes configurations that deliver optimal tradeoffs between accuracy and performance.

Approximate Program Synthesis. The semantics of the Syndy's language of map-fold computations enables using the probabilistic choice expression to implement the choice between multiple function implementations and sampling of inputs of fold operations. The probabilistic choice expression, $e_1 \,{}_q\oplus\, e_2$, is controlled by the *configuration variable* q: the expression e_1 is executed with probability q and the expression e_2 is executed with probability $1 - q$. The value of q is obtained from the *concrete configuration* vector and set at the beginning of the program's execution.

To synthesize the body of approximate composite functions, Syndy first searches for the calls to primitive functions that can be approximated – these are the functions for which the developer has specified alternative implementations in the accuracy specification. Syndy replaces each such call with an expression that consists of probabilistic choice operators between the implementations of the primitive function.

```
ImageEdge'(image, model)    :=    IEdge(image, model) q_e1⊕ IEdge1(image, model) q_e2⊕
                                  IEdge2(image, model) q_e3⊕ IEdge3(image, model)
ImageInside'(image, model)  :=    IInside(image, model) q_i1⊕ IInside1(image, model) q_i2⊕
                                  IInside2(image, model) q_i3⊕ IInside3(image, model)

Score'(image, model)        :=    Exp(-1*(ImageEdge'(image, model) + ImageInside'(image, model))).
```

Figure 5: Transformed Score Functions

Figure 5 presents the configurable implementation of the function `Score`. This alternative computation randomly calls the original or approximate implementations of the functions `ImageEdge` and `ImageInside`. For clarity, we extracted the probabilistic choice expressions to auxiliary functions `ImageEdge'` and `ImageInside'`. Probabilistic choice operations execute from left to right. The variables q_{e1}, q_{e2}, q_{e3} and q_{i1}, q_{i2}, q_{i3} control the execution of the function `Score'`. Syndy transforms the map operator to call the function `Score'` instead of the original function `Score`.

Syndy similarly transforms the maximization fold operator. The function `Max`(a, b), which computes the maximum between the temporary aggregation variable a (which contains the maximum value seen so far) and the new input b, is replaced with the function that skips computing the input b with probability $1 - q_m$:

```
Max'(a,b)    :=    Max(a,b) q_m⊕ a.
```

Therefore, the final approximate configurable program that Syndy generates is:

```
program (images, models) :=
    fold(0, Max', map(Score', zip(images, models)))
```

The variables q_{e1}, q_{e2}, q_{e3}, q_{i1}, q_{i2}, q_{i3}, and q_m represent the *configuration* of the synthesized approximate program. This program can execute at many points in the tradeoff space by selecting the values of the configuration variables. We next present how to compute the values of the configurations parameters that deliver profitable tradeoffs.

5. Optimal Configuration Synthesis

To synthesize the configurations of the approximate map-fold program, Syndy 1) translates the map-fold computation into an equivalent graph model from the accuracy-aware model of computation and 2) runs the optimization algorithm from [21] on this graph model to obtain the program configurations.

Accuracy-aware Model of Computation. The accuracy-aware model of computation presented in [21] is a tree of computation and reduction nodes. A *computation node* represents the computation that executes independently on m inputs in parallel. The number of inputs m represents *multiplicity* of the node. The computation performed on each input is represented as a dataflow *directed acyclic graph*, where each node represents a computation (called *function nodes*), and each edge between two nodes represents a data flow between the nodes. A *reduction node* represents aggregate operations (summation, averaging, minimization, and maximization) on m inputs.

Translation to Accuracy-aware Model. Syndy translates the map-fold program to a tree of computation and reduction nodes. The functions with alternative implementations in map-fold programs are translated to function nodes. The map operators are translated to computation nodes. The fold operators with appropriate function (summation, averaging, minimization, or maximization) are translated to the corresponding reduction nodes. The translation component also represents each alternate synthesized computation as a transformation within the accuracy-aware model. We present the details of the translation in [21, Sections 4 and 5].

Figure 6 presents the generated computation model for the example program. The map operator that calls the function `Score` is translated to a computation node. The body of the computation node is the graph that represents the data flow within the function `Score`. The multiplicity of the computation node is equal to the number of elements of the map operator specified by the input specification (400). The calls to the primitive functions `Exp`, `ImageEdge`, and `ImageInside` are translated to function nodes. The functions `ImageEdge` and `ImageInside` have alternative implementations – the model uses the accuracy/performance specification from Figure 2. The fold operator that performs maximization is translated to the maximization reduction node.

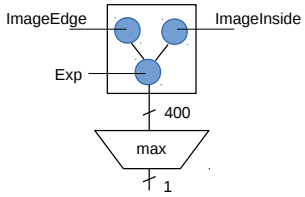

Figure 6: Accuracy-aware Model of the Example Program

Optimal Configuration Search Algorithm. The configuration optimization algorithm uses the graph model to search the tradeoff space induced by the program configurations. The optimization algorithm produces a $(1+\varepsilon)$-optimal accuracy/performance *tradeoff curve* [21]. The parameter ε is a small constant that specifies the relaxation of the optimality of the tradeoff curve.

Each point on the generated tradeoff curve contains 1) the upper bound on the expected absolute error, 2) the upper bound on the execution time, and 3) the configuration of the transformed program that delivers the specified tradeoff. Figure 7 presents the optimal tradeoff curve between the error and the execution time that Syndy produces for the example computation. The execution time is normalized to represent the fraction of the time of the fully accurate program.

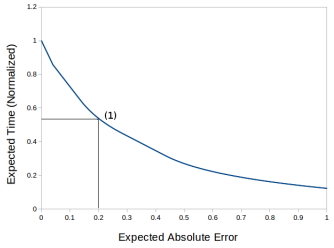

Figure 7: Tradeoff Curve of the Example Program

5.1 Execution Configuration Selection

At the start of the execution of the synthesized randomized program, the runtime system selects the program's configuration based on the user-provided error tolerance bound Δ. The program's runtime module reads the concrete configuration that guarantees that the execution error is smaller than Δ from the tradeoff curve. The runtime assigns the values from the concrete configuration to the configuration variable of the approximate program.

If the user of the example computation selects tolerance bound $\Delta = 0.2$, the runtime will select the configuration from the point *(1)* in Figure 7. This configuration executes in time at most 55% of the execution time of the original (non-approximate) program. The runtime system replaces each configuration parameter q with the corresponding numerical value from the obtained configuration vector. Then the executions of such program are guaranteed to produce the results whose errors are bounded by Δ in expectation.

6. Related Work

Quantitative Program Synthesis. Researchers have recently explored numerical optimization techniques for generation of computations that satisfy quantitative constraints. Smooth interpretation uses a gradient descent based method to synthesize program's control parameters for given representative inputs [9] and probability distributions of inputs [7]. It requires the developer to manually select missing control parameters; it does not automatically transform the fully accurate computation. [6] and [20] present quantitative model checking frameworks for expressing and solving tradeoffs between quantitative properties of computations for reactive systems. These techniques are applicable for computations modeled using Markov decision processes.

Accuracy Analysis of Program Transformations. Researchers have presented several papers on static analysis of program transformations that affect accuracy of results. These techniques were used to justify the application of transformations such as loop perforation [8, 13], reason about differential privacy mechanisms [3, 15], or check the satisfiability of probabilistic assertions in the presence of external noise [10, 18]. However, the main focus of these techniques is the analysis of error propagation and not the search of profitable accuracy/performance tradeoffs.

7. Conclusion

The field of program optimization has focused, almost exclusively since the inception of the field, on transformations that do not change the result that the computation produces. The recent emergence of approximate program transformations promises to dramatically increase the scope and relevance of program optimization techniques in a world increasingly dominated by computations that can profitably trade off accuracy in return for increased performance. Syndy provides an opportunity to exploit such profitable tradeoffs by automatically synthesizing approximate map-fold programs, while providing probabilistic accuracy guarantees.

References

[1] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: a language and compiler for algorithmic choice. PLDI, 2009.

[2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. PLDI, 2010.

[3] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic reasoning for differential privacy. POPL, 2012.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*.

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

[6] K. Chatterjee, T. Henzinger, B. Jobstmann, and R. Singh. Quasy: Quantitative synthesis tool. In *TACAS*, 2011.

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

[8] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving Programs Robust. FSE, 2011.

[9] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. PLDI '10.

[10] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. ICSE, 2013.

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

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

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

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

[15] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ICFP*, 2010.

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

[17] M. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. OOPSLA, 2007.

[18] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, 2013.

[19] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. FSE '11.

[20] C. Von Essen and B. Jobstmann. Synthesizing efficient controllers. In *VMCAI*, 2012.

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