View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

Randomized Accuracy-Aware Program Transformations For Efficient Approximate Computations

Zeyuan Allen Zhu   Sasa Misailovic   Jonathan A. Kelner   Martin Rinard

MIT CSAIL

zeyuan@csail.mit.edu   misailo@mit.edu   kelner@mit.edu   rinard@mit.edu

Abstract

Despite the fact that approximate computations have come to dominate many areas of computer science, the field of program transformations has focused almost exclusively on traditional semantics-preserving transformations that do not attempt to exploit the opportunity, available in many computations, to acceptably trade off accuracy for benefits such as increased performance and reduced resource consumption.

We present a model of computation for approximate computations and an algorithm for optimizing these computations. The algorithm works with two classes of transformations: *substitution transformations* (which select one of a number of available implementations for a given function, with each implementation offering a different combination of accuracy and resource consumption) and *sampling transformations* (which randomly discard some of the inputs to a given reduction). The algorithm produces a $(1 + \varepsilon)$ randomized approximation to the optimal randomized computation (which minimizes resource consumption subject to a probabilistic accuracy specification in the form of a maximum expected error or maximum error variance).

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—*optimization*; G.3 [*Probability and Statistics*]: Probabilistic Algorithms; F.2.1 [*Analysis of Algorithms and Problem Complexity*]: Numerical Algorithms and Problems

General Terms Algorithms, Design, Performance, Theory

Keywords Optimization, Error-Time Tradeoff, Discretization, Probabilistic

1. Introduction

Computer science was founded on *exact computations* with discrete logical correctness requirements (examples include compilers and traditional relational databases). But over the last decade, *approximate computations* have come to dominate many fields. In contrast to exact computations, approximate computations aspire only to produce an acceptably accurate approximation to an exact (but in many cases inherently unrealizable) output. Examples include machine learning, unstructured information analysis and retrieval, and lossy video, audio and image processing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

Copyright © 2012 ACM 978-1-4503-1083-3/12/01...$10.00

Despite the prominence of approximate computations, the field of program transformations has remained focused on techniques that are guaranteed not to change the output (and therefore do not affect the accuracy of the approximation). This situation leaves the developer solely responsible for managing the approximation. The result is inflexible computations with hard-coded approximation choices directly embedded in the implementation.

1.1 Accuracy-Aware Transformations

We investigate a new class of transformations, *accuracy-aware transformations*, for approximate computations. Given a computation and a probabilistic accuracy specification, our transformations change the computation so that it operates more efficiently while satisfying the specification. Because accuracy-aware transformations have the freedom to change the output (within the bounds of the accuracy specification), they have a much broader scope and are therefore able to deliver a much broader range of benefits.

The field of accuracy-aware transformations is today in its infancy. Only very recently have researchers developed general transformations that are designed to manipulate the accuracy of the computation. Examples include task skipping [27, 28], loop perforation [14, 23, 24, 31], approximate function memoization [6], and substitution of multiple alternate implementations [2, 3, 12, 33]. When successful, these transformations deliver programs that can operate at multiple points in an underlying accuracy-resource consumption tradeoff space. Users may select points that minimize resource consumption while satisfying the specified accuracy constraints, maximize accuracy while satisfying specified resource consumption constraints, or dynamically change the computation to adapt to changes (such as load or clock rate) in the underlying computational platform [12, 14].

Standard approaches to understanding the structure of the tradeoff spaces that accuracy-aware transformations induce use training executions to derive empirical models [2, 3, 14, 24, 27, 28, 31, 33]. Potential pitfalls include models that may not accurately capture the characteristics of the transformed computation, poor correlations between the behaviors of the computation on training and production inputs, a resulting inability to find optimal points in the tradeoff space for production inputs, and an absence of guaranteed bounds on the magnitude of potential accuracy losses.

1.2 Our Result

We present a novel analysis and optimization algorithm for a class of approximate computations. These computations are expressed as a tree of *computation nodes* and *reduction nodes*. Each computation node is a directed acyclic graph of nested *function nodes*, each of which applies an arbitrary function to its inputs. A reduction node applies an aggregation function (such as min, max, or mean) to its inputs.

We consider two classes of accuracy-aware transformations. *Substitution transformations* replace one implementation of a function node with another implementation. Each function has a *propagation* specification that characterizes the sensitivity of the function to perturbations in its inputs. Each implementation has *resource consumption* and *accuracy* specifications. Resource consumption specifications characterize the resources (such as time, energy, or cost) each implementation consumes to compute the function. Accuracy specifications characterize the error that the implementation introduces.

Sampling transformations cause the transformed reduction node to operate on a randomly selected subset of its inputs, simultaneously eliminating the computations that produce the discarded inputs. Each sampling transformation has a *sampling rate*, which is the ratio between the size of the selected subset of its inputs and the original number of inputs.

Together, these transformations induce a space of *program configurations*. Each configuration identifies an implementation for every function node and a sampling rate for every reduction node. In this paper we work with randomized transformations that specify a probabilistic choice over configurations. Our approach focuses on understanding the following technical question:

What is the optimal accuracy-resource consumption tradeoff curve available via our randomized transformations?

Understanding this question makes it possible to realize a variety of optimization goals, for example minimizing resource consumption subject to an accuracy specification or maximizing accuracy subject to a resource consumption specification. The primary technical result in this paper is an optimization algorithm that produces a $(1 + \varepsilon)$-approximation to the optimal randomized computation (which minimizes resource consumption subject to a probabilistic accuracy specification in the form of a maximum expected error or maximum error variance). We also discuss how to realize a variety of other optimization goals.

1.3 Challenges and Solutions

Finding optimal program configurations presents several algorithmic challenges. In particular:

- **Exponential Configurations:** The number of program configurations is exponential in the size of the computation graph, so a brute-force search for the best configuration is computationally intractable.
- **Randomized Combinations of Configurations:** A transformed program that randomizes over multiple configurations may substantially outperform one that chooses any single fixed configuration. We thus optimize over an even larger space—the space of probability distributions over the configuration space.
- **Global Error Propagation Effects:** Local error allocation decisions propagate globally throughout the program. The optimization algorithm must therefore work with global accuracy effects and interactions between errors introduced at the nodes of the computation graph.
- **Nonlinear, Nonconvex Optimization Problem:** The running time and accuracy of the program depend nonlinearly on the optimization variables. The resulting optimization problem is nonlinear and nonconvex.

We show that, in the absence of reduction nodes, one can formulate the optimization problem as a linear program, which allows us to obtain an exact optimization over the space of probability distributions of configurations in polynomial time.

The question becomes much more involved when reduction nodes come to the picture. In this case, we approximate the optimal tradeoff curve, but to a $(1 + \varepsilon)$ precision for an arbitrarily small constant $\varepsilon > 0$. Our algorithm has a running time that is polynomially dependent on $\frac{1}{\varepsilon}$. It is therefore a fully polynomial-time approximation scheme (FPTAS).

Our algorithm tackles reduction nodes one by one. For each reduction node, it discretizes the tradeoff curve achieved by the subprogram that generates the inputs to the reduction node. This discretization uses a special *bi-dimensional discretization* technique that is specifically designed for such tradeoff problems. We next show how to extend this discretization to obtain a corresponding discretized tradeoff curve that includes the reduction node. The final step is to recursively combine the discretizations to obtain a dynamic programming algorithm that approximates the optimal tradeoff curve for the entire program.

We note that the optimization algorithm produces a weighted combination of program configurations. We call such a weighted combination a *randomized configuration*. Each execution of the final randomized program chooses one of these configurations with probability proportional to its weight.

Randomizing the transformed program provides several benefits. In comparison with a deterministic program, the randomized program may be able to deliver substantially reduced resource consumption for the same accuracy specification. Furthermore, randomization also simplifies the optimization problem by replacing the discrete search space with a continuous search space. We can therefore use linear programs (which can be solved efficiently) to model regions of the optimization space instead of integer programs (which are, in general, intractable).

1.4 Potential Applications

A precise understanding of the consequences of accuracy-aware transformations will enable the field to mature beyond its current focus on transformations that do not change the output. This increased scope will enable researchers in the field to attack a much broader range of problems. Some potential examples include:

- **Sublinear Computations On Big Data:** Sampling transformations enable the optimization algorithm to automatically find *sublinear computations* that process only a subset of the inputs to provide an acceptably accurate output. Over the past decade, researchers have developed many sublinear algorithms [29]. Accuracy-aware transformations hold out the promise of automating the development of many of these algorithms.
- **Incrementalized and Online Computations:** Many algorithms can be viewed as converging towards an optimal exact solution as they process more inputs. Because our model of computation supports such computations, our techniques make it possible to characterize the accuracy of the current result as the computation incrementally processes inputs. This capability opens the door to the automatic development of *incrementalized* computations (which incrementally sample available inputs until the computation produces an acceptably accurate result) and *online* computations (which characterize the accuracy of the current result as the computation incrementally processes dynamically arriving inputs).
- **Sensor Selection:** Sensor networks require low power, low cost sensors [32]. Accuracy-aware transformations may allow developers to specify a sensor network computation with idealized lossless sensors as the initial function nodes in the computation. An optimization algorithm can then select sensors that minimize power consumption or cost while still providing acceptable accuracy.
- **Data Representation Choices:** Data representation choices can have dramatic consequences on the amount of resources (time, silicon area, power) required to manipulate that data [10]. Giving an optimization algorithm the freedom to adjust the ac-

curacy (within specified bounds) may enable an informed automatic selection of less accurate but more appropriate data representations. For example, a compiler may automatically replace an expensive floating point representation with a more efficient but less accurate fixed point representation. We anticipate the application of this technology in both standard compilers for microprocessors as well as hardware synthesis systems.

- **Dynamic Adaptation In Large Data Centers:** The amount of computing power that a large data center is able to deliver to individual hosted computations can vary dynamically depending on factors such as load, available power, and the operating temperature within the data center (a rise in temperature may force reductions in power consumption via clock rate drops). By delivering computations that can operate at multiple points in the underlying accuracy-resource consumption tradeoff space, accuracy-aware transformations open up new strategies for adapting to fluctuations. For example, a data center may respond to load or temperature spikes by running applications at less accurate but more efficient operating points [12].
- **Successful Use of Mostly Correct Components:** Many faulty components operate correctly for almost all inputs. By perturbing inputs and computations with small amounts of random noise, it is possible to ensure that, with very high probability, no two executions of the computation operate on the same values. Given a way to check if a fault occurred during the execution, it is possible to rerun the computation until all components happen to operate on values that elicit no faults. Understanding the accuracy consequences of these perturbations can make it possible to employ this approach successfully.[1]

The scope of traditional program transformations has been largely confined to standard compiler optimizations. As the above examples illustrate, appropriately ambitious accuracy-aware transformations that exploit the opportunity to manipulate accuracy within specified bounds can dramatically increase the impact and relevance of the field of program analysis and transformation.

1.5 Contributions

This paper makes the following contributions:

- **Model of Computation:** We present a model of computation for approximate computations. This model supports arbitrary compositions of individual function nodes into computation nodes and computation nodes and reduction nodes into computation trees. This model exposes enough computational structure to enable approximate optimization via our two transformations.
- **Accuracy-Aware Transformations:** We consider two classes of accuracy-aware transformations: function substitutions and reduction sampling. Together, these transformations induce a space of transformed programs that provide different combinations of accuracy and resource consumption.
- **Tradeoff Curves:** It shows how to use linear programming, dynamic programming, and a special bi-dimensional discretization technique to obtain a $(1+\epsilon)$-approximation to the underlying optimal accuracy-resource consumption tradeoff curve available via the accuracy-aware transformations. If the program contains no reduction nodes, the tradeoff curve is exact.
- **Optimization Algorithm:** It presents an optimization algorithm that uses the tradeoff curve to produce randomized programs that satisfy specified probabilistic accuracy and resource consumption constraints. In comparison with approaches that attempt to deliver a deterministic program, randomization enables our optimization algorithm to 1) deliver programs with better combinations of accuracy and resource consumption, and 2) avoid a variety of intractability issues.
- **Accuracy Bounds:** We show how to obtain statically guaranteed probabilistic accuracy bounds for a general class of approximate computations. The only previous static accuracy bounds for accuracy-aware transformations exploited the structure present in a set of computational patterns [6, 22, 23].

[1] The last author would like to thank Pat Lincoln for an interesting discussion on this topic.

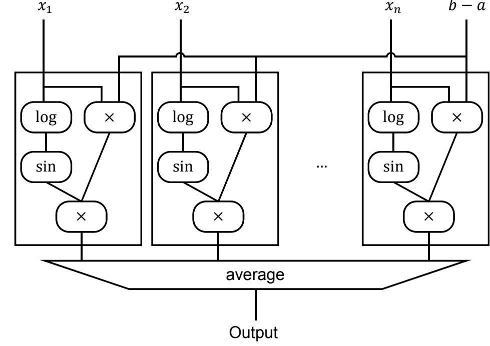

Figure 1: A numerical integration program.

2. Example

We next present an example computation that numerically integrates a univariate function $f(x)$ over a fixed interval $[a, b]$. The computation divides $[a, b]$ into n equal-sized subintervals, each of length $\Delta x = \frac{b-a}{n}$. Let $\boldsymbol{x} = (x_1, \ldots x_n)$, where $x_i = a + i \cdot \frac{\Delta x}{2}$. The value of the numerical integral I is equal to

$$I = \Delta x \cdot \sum_{i=1}^{n} f(x_i) = \frac{1}{n} \sum_{i=1}^{n} (b-a) \cdot f(x_i) \ .$$

Say, for instance, $f(x) = x \cdot \sin\big(\log(x)\big)$ is the function that we want to integrate and $[a, b] = [1, 11]$.

Our Model of Computation. As illustrated in Figure 1, in our model of computation, we have n input edges that carry the values of x_i's into the computation and an additional edge that carries the value of $b-a$. For each x_i, a computation node calculates the value of $(b-a) \cdot f(x_i)$. The output edges of these nodes are connected to a reduction node that computes the average of these values (we call such a node an averaging node), as the final integral I.

Program Transformations. The above numerical integration program presents multiple opportunities to trade end-to-end accuracy of the result I in return for increased performance. Specifically, we identify the following two transformations that may improve the performance:

- **Substitution.** It is possible to substitute the original implementations of the $\sin(\cdot)$ and $\log(\cdot)$ functions that comprise $f(x)$ with alternate implementations that may compute a less accurate output in less time.
- **Sampling.** It is possible to discard some of the n inputs of the averaging node (and the computations that produce these inputs) by taking a random sample of $s \leq n$ inputs (here we call s the reduction factor). Roughly speaking, this transformation introduces an error proportional to $\frac{1}{\sqrt{s}}$, but decreases the running time of the program proportionally to $\frac{s}{n}$.

Tradeoff Space. In this numerical integration problem, a *program configuration* specifies which implementation to pick for each of the functions $\sin(\cdot)$, $\log(\cdot)$, and (in principle, although we do not

Configuration.	Weight	$x_{\log,0}$	$x_{\log,1}$	$x_{\log,2}$	$x_{\sin,0}$	$x_{\sin,1}$	$x_{\sin,2}$	s/n	Error	Speedup
C_1	0.608	0	0	1	0	0	1	100%	0.024	1.39
C_2	0.392	0	1	0	0	1	0	47.5%	0.090	2.63

Table 1: The $(1+\varepsilon)$-optimal randomized program configuration for $\Delta = 0.05$ and $\varepsilon = 0.01$.

do so in this example) $\times$. The configuration also specifies the reduction factor s for the averaging node. If we assume that we have two alternate implementations of $\sin(\cdot)$ and $\log(\cdot)$, each program configuration provides the following information: 1) $x_{u,i} \in \{0,1\}$ indicating whether we choose the i-th implementation of the function $u \in \{\log, \sin\}$, and $i \in \{0,1,2\}$, and 2) s indicating the reduction factor for the averaging node we choose. A *randomized* program configuration is a probabilistic choice over program configurations.

Function Specifications. We impose two basic requirements on the implementations of all functions that comprise $f(x)$.

The first requirement is that we have an error bound and time complexity specification for each implementation of each function. In this example we will use the following model: the original implementation of $\log(\cdot)$ executes in time $T_{\log,0}$ with error $E_{\log,0} = 0$; the original implementation of $\sin(\cdot)$ executes in time $T_{\sin,0}$ with error $E_{\sin,0} = 0$. We have two alternate implementations of $\log(\cdot)$ and $\sin(\cdot)$, where the i-th implementation of a given function $u \in \{\log(\cdot), \sin(\cdot)\}$ runs in time $T_{u,i} = \left(1 - \frac{\sqrt{i}}{5}\right)T_{u,0}$ with error $E_{\log,i} = i \cdot 0.008$, and $E_{\sin,i} = i \cdot 0.004$ ($i \in \{1,2\}$).

The second requirement is that the error propagation of the entire computation is bounded by a linear function. This requirement is satisfied if the functions that comprise the computation are *Lipschitz continuous*[2]. In our example, the function $\sin(\cdot)$ is 1-Lipschitz continuous, since its derivative is bounded by 1. The function $\log(x)$ is also Lipschitz continuous, when $x \geq 1$. Finally, the product function $\times$ is Lipschitz continuous, when the two inputs are bounded. We remark here that this second requirement ensures that an error introduced by an approximate implementation propagates to cause at most a linear change in the final output.

Finding the $(1+\varepsilon)$-Optimal Program Configuration. Given performance and accuracy specifications for each function, we can run our optimization algorithm to $(1+\varepsilon)$-approximately calculate the optimal accuracy-performance tradeoff curve. For each point on the curve our algorithm can also produce a randomized program configuration that achieves this tradeoff.

Given a target expected error bound Δ, we use the tradeoff curve to find a randomized program configuration that executes in expected time τ. The $(1+\varepsilon)$-approximation ensures that this expected running time τ is at most $(1+\varepsilon)$ times the optimal expected running time for the expected error bound Δ. In this example we use $\varepsilon = 0.01$ so that our optimized program will produce a 1.01-approximation. In addition, we define:

- the number of inputs $n = 10000$,
- the overall expected error tolerance $\Delta = 0.05$, and
- the running times $T_{\sin,0} = 0.08\ \mu$s and $T_{\log,0} = 0.07\ \mu$s.

For this example our optimization algorithm identifies the point $(\Delta, T_0/1.71)$ on the tradeoff curve, where T_0 is the running time of the original program. This indicates that the optimized program achieves a speedup of 1.71 over the original program while keeping the expected value below the bound Δ. Table 1 presents the randomized program configuration that achieves this tradeoff. This randomized program configuration consists of two program configurations C_1 and C_2. Each configuration has an associated weight which is the probability with which the randomized program will execute that configuration. The table also presents the error and speedup that each configuration produces.

The configuration C_1 selects the less accurate approximate versions of the functions $\log(\cdot)$ and $\sin(\cdot)$, and uses all inputs to the averaging reduction node. The configuration C_2, on the other hand, selects more accurate approximate versions of the functions $\log(\cdot)$ and $\sin(\cdot)$, and at the same time samples 4750 of the 10,000 original inputs.

Note that individually neither C_1 nor C_2 can achieve the desired tradeoff. The configuration C_1 produces a more accurate output but also executes significantly slower than the optimal program. The configuration C_2 executes much faster than the optimal program, but with expected error greater than the desired bound Δ. The randomized program selects configuration C_1 with probability 60.8% and C_2 with probability 39.2%. The randomized program has expected error Δ and expected running time $T_0/1.71$.

We can use the same tradeoff curve to obtain a randomized program that minimizes the expected error Δ' subject to the execution time constraint τ'. In our example, if the time bound $\tau' = T_0/1.71$ the optimization algorithm will produce the program configuration from Table 1 with expected error $\Delta' = \Delta$.

More generally, our optimization algorithm will produce an efficient representation of a probability distribution over program configurations along with an efficient procedure to sample this distribution to obtain a program configuration for each execution.

3. Model of Approximate Computation

We next define the graph model of computation, including the error-propagation constraints for function nodes, and present the accuracy-aware substitution and sampling transformations.

3.1 Definitions

Programs. In our model of computation, programs consist of a directed tree of *computation nodes* and *reduction nodes*. Each edge in the tree transmits a stream of values. The *size* of each edge indicates the number of transmitted values. The multiple values transmitted along an edge can often be understood as a stream of numbers with the same purpose — for example, a million pixels from an image or a thousand samples from a sensor. Figure 2 presents an example of a program under our definition.

Reduction Nodes. Each reduction node has a single input edge and a single output edge. It reduces the size of its input by some multiplicative factor, which we call its *reduction factor*. A node with reduction factor S has an input edge of size $R \cdot S$ and an output edge of size R. The node divides the $R \cdot S$ inputs into blocks of size S. It produces R outputs by applying an S-to-1 aggregation function (such as min, max, or mean) to each of the R blocks.

For clarity of exposition and to avoid a proliferation of notation, we primarily focus on one specific type of reduction node, which we call an *averaging node*. An averaging node with reduction factor S will output the average of the first S values as the first output, the average of the next S values as the second output, and so on.

The techniques that we present are quite general and apply to any reduction operation that can be approximated well by sampling. Section 8 describes how to extend our algorithm to work with other reduction operations.

[2] A univariate function is α-Lipschitz continuous if for any $\delta > 0$, it follows that $|f(x) - f(x+\delta)| < \alpha\delta$. As a special case, a differentiable function is Lipschitz continuous if $|f'(x)| \leq \alpha$. This definition extends to multivariate functions.

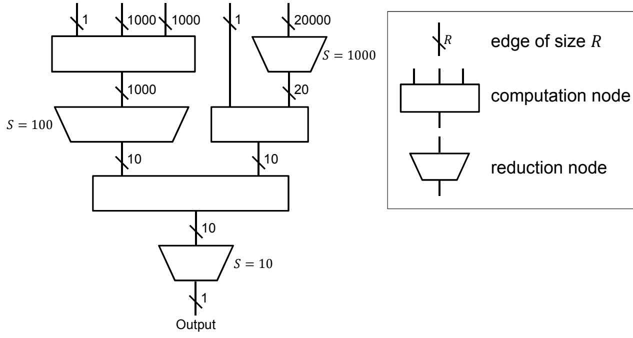

Figure 2: An example program in our model of computation.

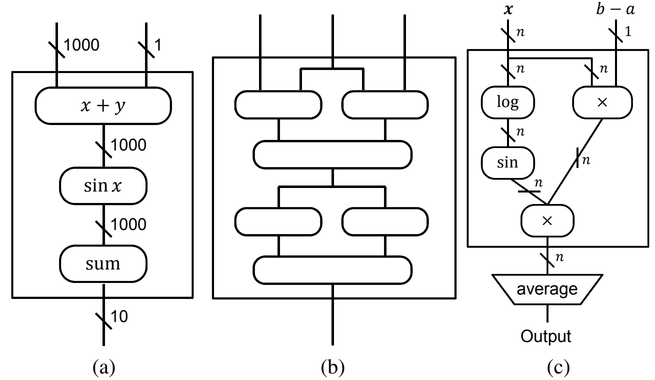

Figure 3: (a)(b): A closer look at two computation nodes, and (c): Numerical integration example, revisited.

Computation Nodes. A computation node has potentially multiple input edges and a single output edge. A computation node of size R has:

- a single output edge of size R;
- a non-negative number of input edges, each of size
  - either 1 (which we call a *control-input* edge),
  - or some multiple tR of R (which we call a *data-input* edge).

Each control-input edge carries a single *global constant*. Data-input edges carry a stream of values which the computation node partitions into R chunks. The computation node executes R times to produce R outputs, with each execution processing the value from each control-input edge and a block of t values from each data-input edge. The executions are independent.

For example, consider a computation node of size 10 with two input edges: one data-input edge of size 1000, denoted by $(a_1, a_2, \ldots, a_{1000})$, and one control-input edge of size 1, denoted by b. Then, the function that outputs the vector

$$\Big(\sum_{i=1}^{100} \sin(a_i, b), \sum_{i=101}^{200} \sin(a_i, b), \ldots, \sum_{i=901}^{1000} \sin(a_i, b)\Big) \quad (1)$$

is a computation node.

We remark here that a reduction node is a special kind of computation node. We treat computation and reduction nodes separately because we optimize computation nodes with substitution transformations and reduction nodes with sampling transformations (see Section 3.2).

Inner Structure of Computation Nodes. A computation node can be further decomposed into one or more *function nodes*, connected via a directed acyclic graph (DAG). Like computation nodes, each function node has potentially multiple input edges and a single output edge. The size of each input edge is either 1 or a multiple of the size of the output edge. The functions can be of arbitrary complexity and can contain language constructs such as conditional statements and loops.

For example, the computation node in Eq.(1) can be further decomposed as shown in Figure 3a. Although we require the computation nodes and edges in each program to form a tree, the function nodes and edges in each computation node can form a DAG (see, for example, Figure 3b).

In principle, any computation node can be represented as a single function node, but its decomposition into multiple function nodes allows for finer granularity and more transformation choices when optimizing entire program.

Example. Section 2 presented a numerical integration program example. Figure 3c presents this example in our model of computation (compare with Figure 1). Note that the multiplicity of computation nodes in Figure 1 corresponds to the edge sizes in Figure 3c. The log function node with input and output edges of size n runs n times. Each run consumes a single input and produces a single output. The $\times$ function node with input edges of size n and 1 runs n times. Each execution produces as output the product of an x_i with the common value $b - a$ from the control edge.

3.2 Transformations

In a *program configuration*, we specify the following two kinds of transformations at function and reduction nodes.

- **Substitution.** For each function node f_u of size R, we have a polynomial number of implementations $f_{u,1}, \ldots, f_{u,k}$. The function runs R times. We require each implementation to have the following properties:
  - each run of $f_{u,i}$ is in expected time $T_{u,i}$, giving a total expected running time of $R \cdot T_{u,i}$, and
  - each run of $f_{u,i}$ produces an expected absolute additive error of at most $E_{u,i}$, i.e., $\forall \boldsymbol{x}, \mathbb{E}[|f_u(\boldsymbol{x}) - f_{u,i}(\boldsymbol{x})|] \leq E_{u,i}$. (The expectation is over the the randomness of $f_{u,i}$ and f_u.)

  We assume that all $(T_{u,i}, E_{u,i})$ pairs are known in advance (they are constants or depend only on control inputs).
- **Sampling.** For each reduction node r with reduction factor S_r, we can decrease this factor S_r to a smaller factor $s_r \in \{1, \ldots, S_r\}$ at the expense of introducing some additive sampling error $E_r(s_r)$. For example, for an averaging node, instead of averaging all S_r inputs, we would randomly select s_r inputs (without replacement) and output the average of the chosen samples. For convenience, we denote the sampling rate of node r as $\eta_r = \frac{s_r}{S_r}$.

  If the output edge is of size R, the computation selects $s_r \cdot R$ inputs, instead of all $S_r \cdot R$ inputs. The values for the reduction node inputs which are not selected need not be computed. Discarding the computations that would otherwise produce these discarded inputs produces a speed-up factor of $\eta_r = \frac{s_r}{S_r}$ for all nodes above r in the computation tree.

The following lemma provides a bound on the sampling error $E_r(s_r) = (B - A)\sqrt{\frac{S_r - s_r}{s_r(S_r - 1)}}$ for an averaging node. The proof is available in the full version of the paper.

Lemma 3.1. *Given m numbers $x_1, x_2, \ldots, x_m \in [A, B]$, randomly sampling s of the numbers $x_{i_1}, \ldots, x_{i_s}$ (without replacement) and computing the sample average gives an approximation to $\frac{x_1+\cdots+x_m}{m}$ with the following expected error guarantee:*

$$\mathbb{E}_{i_1,\ldots,i_s}\Big[\big|\frac{x_{i_1}+\cdots+x_{i_s}}{s}-\frac{x_1+\cdots+x_m}{m}\big|\Big] \leq (B-A)\sqrt{\frac{m-s}{s(m-1)}} \ .$$

3.3 Error Propagation

The errors that the transformations induce in one part of the computation propagate through the rest of the computation and can be amplified or attenuated in the process. We next provide constraints on the form of functions that characterize this error propagation. These constraints hold for all functions in our model of computation (regardless of whether they have alternate implementations or not).

We assume that for each function node $f_u(x_1, \ldots, x_m)$ with m inputs, if each input x_j is replaced by some approximate input $\hat{x}_j$ such that $\mathbb{E}[|x_j - \hat{x}_j|] \leq \delta_j$, the propagation error is bounded by a linear *error propagation function* $\mathcal{E}_u$:

$$\mathbb{E}\big[|f_u(x_1,\ldots,x_m) - f_u(\hat{x}_1,\ldots,\hat{x}_m)|\big] \leq \mathcal{E}_u\big(\delta_1,\ldots,\delta_m\big). \quad (2)$$

We assume that all of the error propagation functions $\mathcal{E}_u$ for the functions f_u are known a priori:

$$\mathbb{E}\big[|f_u(x_1,\ldots,x_m) - f_u(\hat{x}_1,\ldots,\hat{x}_m)|\big] \leq \big(\sum_j \alpha_j\delta_j\big). \quad (3)$$

This condition is satisfied if all functions f_u are *Lipschitz-continuous* with parameters $\boldsymbol{\alpha}$. Furthermore, if $f_u(x_1, \ldots, x_m)$ is differentiable, we can let $\alpha_i = \max_{\boldsymbol{x}} \big|\frac{\partial f_u(x_1,\ldots,x_m)}{\partial x_i}\big|$. If f_u is itself probabilistic, we can take the expected value of such α_i's.

Substitute Implementations. For functions with multiple implementations, the overall error when we choose the i-th implementation $f_{u,i}$ is bounded by an error propagation function $\mathcal{E}_u$ and the local error induced by the i-th implementation $E_{u,i}$ (defined in the previous subsection):

$$\mathbb{E}\big[|f_u(x_1,\ldots,x_m) - f_{u,i}(\hat{x}_1,\ldots,\hat{x}_m)|\big] \leq \mathcal{E}_u\big(\delta_1,\ldots,\delta_m\big) + E_{u,i}. \quad (4)$$

This bound follows immediately from the triangle inequality.

We remark here that the randomness for the expectation in Eq.(4) comes from 1) the randomness of its input $\hat{x}_1, \ldots, \hat{x}_m$ (caused by errors from previous parts of the computation) and 2) random choices in the possibly probabilistic implementation $f_{u,i}$. These two sources of randomness are mutually independent.

Averaging Reduction Node. The averaging function is a Lipschitz-continuous function with all $\alpha_i = \frac{1}{m}$, so in addition to Lemma 3.1 we have:

Corollary 3.2. *Consider an averaging node that selects s random samples from its m inputs, where each input $\hat{x}_j$ has bounded error $\mathbb{E}[|\hat{x}_j - x_j|] \leq \delta_j$. Then:*

$$\mathbb{E}_{i_1,\ldots,i_s,\hat{x}_1,\ldots,\hat{x}_m}\Big[\big|\frac{\hat{x}_{i_1}+\cdots+\hat{x}_{i_s}}{s}-\frac{x_1+\cdots+x_m}{m}\big|\Big] \leq \frac{1}{m}\big(\sum_{j=1}^{m}\delta_j\big) + (B-A)\sqrt{\frac{m-s}{s(m-1)}} \ .$$

If all input values have the same error bound $\mathbb{E}[|\hat{x}_j - x_j|] \leq \delta$, then $\frac{1}{m}\big(\sum_{j=1}^{m}\delta_j\big) = \delta$.

4. Approximation Questions

We focus on the following question:

Question 1. *Given a program P in our model of computation and using randomized configurations, what is the optimal error-time tradeoff curve that our approximate computations induce?*

Here the time and error refer to the expected running time and error of the program. We say that the expected error of program P' is Δ, if for all input x, $\mathbb{E}[|P(x) - P'(x)|] \leq \Delta$. The error-time tradeoff curve is a pair of functions $(E(\cdot), T(\cdot))$, such that $E(t)$ is the optimal expected error of the program if the expected running time is no more than t, and $T(e)$ is the optimal expected running time of the program if the expected error is no more than e.

The substitution and sampling transformations give rise to an exponentially large space of possible program configurations. We optimize over *arbitrary probability distributions* of such configurations. A naive optimization algorithm would therefore run in time at least exponential in the size of the program. We present an algorithm that approximately solves Question 1 within a factor of $(1+\varepsilon)$ in time:[3] 1) polynomial in the size of the computation graph, and 2) polynomial in $\frac{1}{\varepsilon}$. The algorithm uses linear programming and a novel technique called *bi-dimensional discretization*, which we present in Section 5.

A successful answer to the above question leads directly to the following additional consequences:

Consequence 1: Optimizing Time Subject to Error

Question 2. *Given a program P in our model, and an overall error tolerance Δ, what is the optimal (possibly randomized) program P' available within our space of transformations, with expected error no more than Δ?*

We can answer this question approximately using the optimization algorithm for Question 1. This algorithm will produce a randomized program with expected running time no more than $(1+\varepsilon)$ times the optimal running time and expected error no more than Δ. The algorithm can also answer the symmetric question to find a $(1+\varepsilon)$-approximation of the optimal program that minimizes the expected error given a bound on the expected running time.

Consequence 2: From Error to Variance

We say that the overall variance (i.e., expected squared error) of a randomized program P' is Δ^2, if for all input x, $\mathbb{E}[|P(x) - P'(x)|^2] \leq \Delta^2$. A variant of our algorithm for Question 1 $(1+\varepsilon)$-approximately answers the following questions:

Question 3. *Given a program P in our model of computation, what is the optimal error-variance tradeoff curve that our approximate computations induce?*

Question 4. *Given a program P in our model, and an overall variance tolerance Δ^2, what is the optimal (possibly randomized) program P' available within our space of transformations, with variance no more than Δ^2?*

Section 7 presents the algorithm for these questions.

Consequence 3: Probabilities of Large Errors

A bound on the expected error or variance also provides a bound on the probability of observing large errors. In particular, an execution

[3] We say that we approximately obtain the curve within a factor of $(1+\varepsilon)$, if for any given running time t, the difference between the optimal error $E(t)$ and our $\hat{E}(t)$ is at most $\varepsilon E(t)$, and similarly for the time function $T(e)$. Our algorithm is a fully polynomial-time approximation scheme (FPTAS). Section 5 presents a more precise definition in which the error function $\hat{E}(t)$ is also subject to an additive error of some arbitrarily small constant.

of a program with expected error Δ will produce an absolute error greater than $t\Delta$ with probability at most $\frac{1}{t}$ (this bound follows immediately from Markov's inequality). Similarly, an execution of a program with variance Δ^2 will produce an absolute error greater than $t\Delta$ with probability at most $\frac{1}{t^2}$.

5. Optimization Algorithm for Question 1

We next describe a recursive, dynamic programming optimization algorithm which exploits the tree structure of the program. To compute the approximate optimal tradeoff curve for the entire program, the algorithm computes and combines the approximate optimal tradeoff curves for the subprograms. We stage the presentation as follows:

- **Computation Nodes Only:** In Section 5.1, we show how to compute the optimal tradeoff curve exactly when the computation consists only of computation nodes and has no reduction nodes. We reduce the optimization problem to a linear program (which is efficiently solvable).
- **Bi-dimensional Discretization:** In Section 5.2, we introduce our *bi-dimensional discretization* technique, which constructs a piecewise-linear discretization of any tradeoff curve $(E(\cdot), T(\cdot))$, such that 1) there are only $O(\frac{1}{\varepsilon})$ segments on the discretized curve, and 2) at the same time the discretization approximates $(E(\cdot), T(\cdot))$ to within a multiplicative factor of $(1+\varepsilon)$.
- **A Single Reduction Node:** In Section 5.3, we show how to compute the approximate tradeoff curve when the given program consists of computation nodes that produce the input for a *single* reduction node r (see Figure 6).
  We first work with the curve when the reduction factor s at the reduction node r is constrained to be a single integer value. Given an expected error tolerance e for the entire computation, each randomized configuration in the optimal randomized program allocates part of the expected error $E_r(s)$ to the sampling transformation on the reduction node and the remaining expected error $e_{\text{sub}} = e - E_r(s)$ to the substitution transformations on the subprogram with only computation nodes.
  One inefficient way to find the optimal randomized configuration for a given expected error e is to simply search all possible integer values of s to find the optimal allocation that minimizes the running time. This approach is inefficient because the number of choices of s may be large. We therefore discretize the tradeoff curve for the input to the reduction node into a small set of linear pieces. It is straightforward to compute the optimal integer value of s within each linear piece. In this way we obtain an *approximate* optimal tradeoff curve for the output of the reduction node when the reduction factor s is constrained to be a single integer.
  We next use this curve to derive an approximate optimal tradeoff curve when the reduction factor s can be determined by a *probabilistic* choice among multiple integer values. Ideally, we would use the convex envelope of the original curve to obtain this new curve. But because the original curve has an infinite number of points, it is infeasible to work with this convex envelope directly. We therefore perform another discretization to obtain a piecewise-linear curve that we can represent with a small number of points. We work with the convex envelope of this new discretized curve to obtain the final approximation to the optimal tradeoff curve for the output of the reduction node r. This curve incorporates the effect of both the substitution transformations on the computation nodes and the sampling transformation on the reduction node.

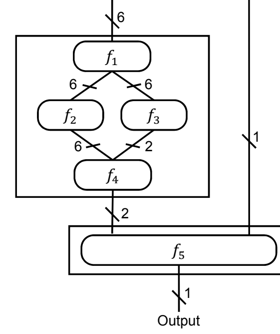

Figure 4: Example to illustrate the computation of time and error.

- **The Final Dynamic Programming Algorithm:** In Section 5.4, we provide an algorithm that computes an approximate error-time tradeoff curve for an *arbitrary* program in our model of computation. Each step uses the algorithm from Section 5.3 to compute the approximate discretized tradeoff curve for a subtree rooted at a topmost reduction node (this subtree includes the computation nodes that produce the input to the reduction node). It then uses this tradeoff curve to replace this subtree with a single function node. It then recursively applies the algorithm to the new program, terminating when it computes the approximate discretized tradeoff curve for the output of the final node in the program.

5.1 Stage 1: Computation Nodes Only

We start with a base case in which the program consists only of computation nodes with no reduction nodes. We show how to use linear programming to compute the optimal error-time tradeoff curve for this case.

Variables $\boldsymbol{x}$. For each function node f_u, the variable $x_{u,i} \in [0, 1]$ indicates the probability of running the i-th implementation $f_{u,i}$. We also have the constraint that $\sum_i x_{u,i} = 1$.

Running Time $\text{TIME}(\boldsymbol{x})$**.** Since there are no reduction nodes in the program, each function node f_u will run R_u times (recall that R_u is the number of values carried on the output edge of f_u). The running time is simply the weighted sum of the running times of the function nodes (where each weight is the probability of selecting each corresponding implementation):

$$\text{TIME}(\boldsymbol{x}) = \sum_u \sum_i (x_{u,i} \cdot T_{u,i} \cdot R_u) \quad . \tag{5}$$

Here the summation u is over all function nodes and i is over all implementations of f_u.

Total Error $\text{ERROR}(\boldsymbol{x})$**.** The total error of the program also admits a linear form. For each function node f_u, the i-th implementation $f_{u,i}$ incurs a local error $E_{u,i}$ on each output value. By the linear error propagation assumption, this $E_{u,i}$ is amplified by a constant factor β_u which depends on the program structure. It is possible to compute the β_u with a traversal of the program backward against the flow of values.

Consider, for example, β_1 for function node f_1 in the program in Figure 4. Let α_2 be the linear error propagation factor for the univariate function $f_2(\cdot)$. The function $f_3(\cdot,\cdot,\cdot)$ is trivariate with 3 propagation factors $(\alpha_{3,1}, \alpha_{3,2}, \alpha_{3,3})$. We similarly define $(\alpha_{4,1}, \ldots, \alpha_{4,4})$ for the quadvariate function f_4, and $(\alpha_{5,1}, \alpha_{5,2}, \alpha_{5,3})$ for f_5. Any error in an output value of f_1 will be amplified by a factor β_1:

$$\beta_1 = \big(\alpha_2(\alpha_{4,1}+\alpha_{4,2}+\alpha_{4,3}) + (\alpha_{3,1}+\alpha_{3,2}+\alpha_{3,3})\alpha_{4,4}\big)(\alpha_{5,1}+\alpha_{5,2}).$$

The total expected error of the program is:

$$\textsc{Error}(\boldsymbol{x}) = \sum_u \sum_i (x_{u,i} \cdot E_{u,i} \cdot \beta_u) \quad . \tag{6}$$

Optimization Given a fixed overall error tolerance Δ, the following *linear program* defines the minimum expected running time:

$$\begin{array}{lll} \text{Variables:} & \boldsymbol{x} & \\ \text{Constraints:} & 0 \le x_{u,i} \le 1, & \forall u, i \\ & \sum_i x_{u,i} = 1 & \forall u \\ & \textsc{Error}(\boldsymbol{x}) \le \Delta & \\ \text{Minimize:} & \textsc{Time}(\boldsymbol{x}) & \end{array} \tag{7}$$

By swapping the roles of $\textsc{Error}(\boldsymbol{x})$ and $\textsc{Time}(\boldsymbol{x})$, it is possible to obtain a linear program that defines the minimum expected error tolerance for a given expected maximum running time.

5.2 Error-Time Tradeoff Curves

In the previous section, we use linear programming to obtain the optimal *error-time tradeoff curve*. Since there are an infinite number of points on this curve, we define the curve in terms of functions. To avoid unnecessary complication when doing inversions, we define the curve using two related functions $E(\cdot)$ and $T(\cdot)$:

Definition 5.1. *The* **(error-time) tradeoff curve** *of a program is a pair of functions $(E(\cdot), T(\cdot))$ such that $E(t)$ is the optimal expected error of the program if the expected running time is no more than t and $T(e)$ is the optimal expected running time of the program if the expected error is no more than e.*

We say that a tradeoff curve is *efficiently computable* if both functions E and T are efficiently computable.[4] The following property is important to keep in mind:

Lemma 5.2. *In a tradeoff curve (E, T), both E and T are non-increasing convex functions.*

Proof. T is always non-increasing because when the allowed error increases the minimum running time does not increase, and similarly for E.

We prove convexity by contradiction: assume $\alpha E(t_1) + (1 - \alpha)E(t_2) < E(\alpha t_1 + (1-\alpha)t_2)$ for some $\alpha \in (0, 1)$. Then choose the optimal program for $E(t_1)$ with probability α, and the optimal program for $E(t_2)$ with probability $1 - \alpha$. The result is a new program P' in our probabilistic transformation space. This new program P' has an expected running time less than the "optimal" running time $E(\alpha t_1 + (1 - \alpha)t_2)$, contradicting the optimality of E. A similar proof establishes the convexity of T. □

We remark here that, given a running time t, one can compute E and be sure that $(E(t), t)$ is on the curve; but one cannot write down all of the infinite number of points on the curve concisely. We therefore introduce a *bi-dimensional discretization* technique that allows us to approximate (E, T) within a factor of $(1 + \varepsilon)$. This technique uses a piecewise linear function with roughly $O(\frac{1}{\varepsilon})$ segments to approximate the curve.

Our bi-dimensional discretization technique (see Figure 5) approximates E in the bounded range $[0, E_{\max}]$, where $E_{\max}$ is an upper bound on the expected error, and approximates T in the bounded range $[T(E_{\max}), T(0)]$. We assume that we are given the maximum acceptable error $E_{\max}$ (for example, by a user of the program). It is also possible to conservatively compute an $E_{\max}$ by analyzing the least-accurate possible execution of the program.

[4] In the remainder of the paper we refer to the function $E(\cdot)$ simply as E and to the function $T(\cdot)$ as T.

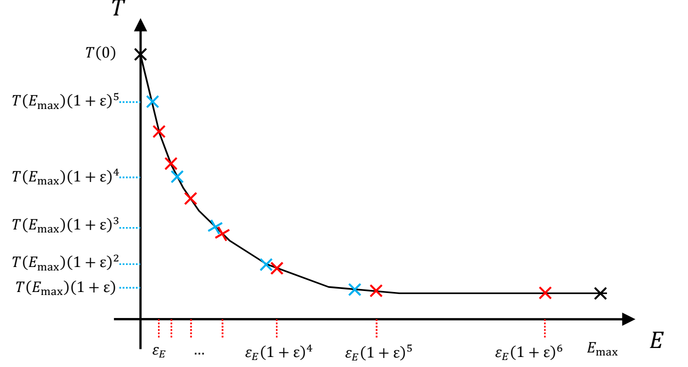

Figure 5: An example of bi-dimensional discretization.

Definition 5.3. *Given a tradeoff curve (E, T) where E and T are both non-increasing, along with constants $\varepsilon \in (0, 1)$ and $\varepsilon_E > 0$, we define the* $(\varepsilon, \varepsilon_E)$**-discretization curve of** (E, T) *to be the piecewise-linear curve defined by the following set of endpoints (see Figure 5):*

- *the two black points $(0, T(0))$, $(E_{\max}, T(E_{\max}))$,*
- *the red points $(e_i, T(e_i))$ where $e_i = \varepsilon_E(1+\varepsilon)^i$ for some $i \ge 0$ and $\varepsilon_E(1+\varepsilon)^i < E_{\max}$, and*
- *the blue points $(E(t_i), t_i)$ where $t_i = T(E_{\max})(1 + \varepsilon)^i$ for some $i \ge 1$ and $T(E_{\max})(1 + \varepsilon)^i < T(0)$.*

Note that there is some asymmetry in the discretization of the two axes. For the vertical time axis we know that the minimum running time of a program is $T(E_{\max}) > 0$, which is always greater than zero since a program always runs in a positive amount of time. However, we discretize the horizontal error axis proportional to powers of $(1 + \varepsilon)^i$ for values above ε_E. This is because the error of a program can indeed reach zero, and we cannot discretize forever.[5]

The following claim follows immediately from the definition:

Claim 5.4. *If the original curve (E, T) is non-increasing and convex, the discretized curve $(\hat{E}, \hat{T})$ is also non-increasing and convex.*

5.2.1 Accuracy of bi-dimensional discretization

We next define notation for the bi-dimensional tradeoff curve discretization:

Definition 5.5. *A curve $(\hat{E}, \hat{T})$ is an* $(\varepsilon, \varepsilon_E)$**-approximation to** (E, T) *if for any error $0 \le e \le E_{\max}$,*

$$0 \le \hat{T}(e) - T(e) \le \varepsilon T(e) \ ,$$

and for any running time $T(E_{\max}) \le t \le T(0)$,

$$0 \le \hat{E}(t) - E(t) \le \varepsilon E(t) + \varepsilon_E \ .$$

We say that such an approximation has a multiplicative error *of ε and an* additive error *of ε_E.*

Lemma 5.6. *If $(\hat{E}, \hat{T})$ is an $(\varepsilon, \varepsilon_E)$-discretization of (E, T), then it is an $(\varepsilon, \varepsilon_E)$-approximation of (E, T).*

Proof Sketch. The idea of the proof is that, since we have discretized the vertical time axis in an exponential manner, if we compute $\hat{T}(e)$ for any value e, the result does not differ from $T(e)$ by

[5] If instead we know that the minimum expected error is greater than zero (i.e., $E(T_{\max}) > 0$) for some maximum possible running time $T_{\max}$, then we can define $\varepsilon_E = E(T_{\max})$ just like our horizontal axis.

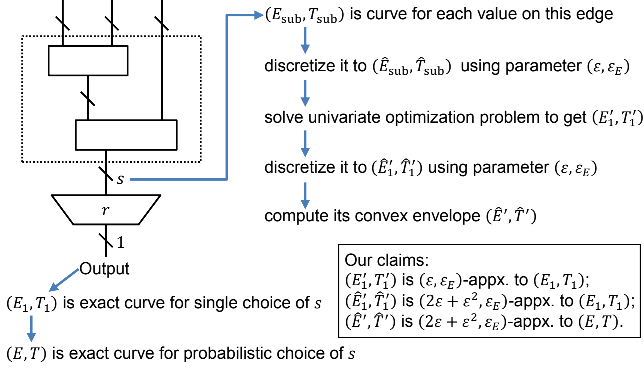

Figure 6: Algorithm for Stage 2.

more than a factor of $(1+\varepsilon)$. Similarly, since we have discretized the horizontal axis in an exponential manner, if we compute $\hat{E}(t)$ for any value t, the result does not differ by more than a factor of $(1+\varepsilon)$, except when $E(t)$ is smaller than ε_E (when we stop the discretization). But even in that case the value $\hat{E}(t) - E(t)$ remains smaller than ε_E.

Because every point on the new piecewise-linear curve $(\hat{E}, \hat{T})$ is a linear combination of some points on the original curve (E, T), $0 \le \hat{T}(e) - T(e)$ and $0 \le \hat{E}(t) - E(t)$, Because (E, T) is convex (recall Lemma 5.2), the approximation will always lie "above" the original curve. $\square$

5.2.2 Complexity of bi-dimensional discretization

The number of segments that the approximate tradeoff curve has in an $(\varepsilon, \varepsilon_E)$-discretization is at most

$$
\begin{aligned}
n_p &\stackrel{\text{def}}{=} 2 + \frac{\log \frac{E_{\max}}{\varepsilon_E}}{\log(1+\varepsilon)} + \frac{\log \frac{T(0)}{T(E_{\max})}}{\log(1+\varepsilon)} \\
&\le O\Big(\frac{1}{\varepsilon}\Big(\log E_{\max} + \log\frac{1}{\varepsilon_E} + \log T_{\max} + \log\frac{1}{T_{\min}}\Big)\Big) \quad , \qquad (8)
\end{aligned}
$$

where $T_{\min}$ is a lower bound on the expected execution time and $T_{\max}$ is an upper bound on the expected execution time. Our discretization algorithm only needs to know $E_{\max}$ in advance, while $T_{\max}$ and $T_{\min}$ are values that we will need later in the complexity analysis.

5.2.3 Discretization on an approximate curve

The above analysis does not rely on the fact that the original tradeoff curve (E, T) is exact. In fact, if the original curve (E, T) is only an $(\varepsilon, \varepsilon_E)$-approximation to the exact error-time tradeoff curve, and if $(\hat{E}, \hat{T})$ is the $(\varepsilon', \varepsilon'_E)$-discretization of (E, T), then one can verify by the triangle inequality that $(\hat{E}, \hat{T})$ is a piecewise linear curve that is an $(\varepsilon + \varepsilon' + \varepsilon\varepsilon', \varepsilon_E + \varepsilon'_E)$ approximation of the exact error-time tradeoff curve.

5.3 Stage 2: A Single Reduction Node

We now consider a program with exactly one reduction node r, with original reduction factor S, at the end of the computation. The example in Figure 3c is such a program. We describe our optimization algorithm for this case step by step as illustrated in Figure 6.

We first define the error-time tradeoff curve for the subprogram without the reduction node r to be $(E_{\text{sub}}, T_{\text{sub}})$ (Section 5.1 describes how to compute this curve; Lemma 5.2 ensures that it is non-increasing and convex). In other words, for *every input value* to the reduction node r, if the allowed running time for computing this value is t, then the optimal expected error is $E_{\text{sub}}(t)$ and similarly for $T_{\text{sub}}(e)$. Note that when computing $(E_{\text{sub}}, T_{\text{sub}})$ as described in Section 5.1, the size of the output edge R_i for each node i must be divided by S, as the curve $(E_{\text{sub}}, T_{\text{sub}})$ characterizes each single input value to the reduction node r.

If at reduction node r we choose an actual reduction factor $s \in \{1, 2, \ldots, S\}$, the total running time and error of this entire program is:[6]

$$
\begin{aligned}
\text{TIME} &= T_{\text{sub}} \times s \\
\text{ERROR} &= E_{\text{sub}} + E_r(s) \ .
\end{aligned} \qquad (9)
$$

This is because, to obtain s values on the input to r, we need to run the subprogram s times with a total time $T_{\text{sub}} \times s$; and by Corollary 3.2, the total error of the output of an averaging reduction node is simply the sum of its input error E_{sub}, and a local error $E_r(s)$ incurred by the sampling.[7]

Let (E_1, T_1) be the exact tradeoff curve (E_1, T_1) of the *entire* program, assuming that we can choose only a single value of s. We start by describing how to compute this (E_1, T_1) approximately.

5.3.1 Approximating (E_1, T_1): single choice of s

By definition, we can write (E_1, T_1) in terms of the following *two* optimization problems:

$$
\begin{aligned}
T_1(e) &= \min_{\substack{s \in \{1,\ldots,S\} \\ e_{\text{sub}} + E_r(s) = e}} \big\{T_{\text{sub}}(e_{\text{sub}}) \times s\big\} \\
\text{and} \quad E_1(t) &= \min_{\substack{s \in \{1,\ldots,S\} \\ t_{\text{sub}} \times s = t}} \big\{E_{\text{sub}}(t_{\text{sub}}) + E_r(s)\big\} \ ,
\end{aligned}
$$

where the first optimization is over variables s and e_{sub}, and the second optimization is over variables s and t_{sub}. We emphasize here that this curve (E_1, T_1) is by definition non-increasing (because $(E_{\text{sub}}, T_{\text{sub}})$ is non-increasing), but may not be convex.

Because these optimization problems may not be convex, they may be difficult to solve in general. But thanks to the piecewise-linear discretization defined in Section 5.2, we can approximately solve these optimization problems efficiently. Specifically, we produce a bi-dimensional discretization $(\hat{E}_{\text{sub}}, \hat{T}_{\text{sub}})$ that $(\varepsilon, \varepsilon_E)$-approximates $(E_{\text{sub}}, T_{\text{sub}})$ (as illustrated in Figure 6). We then solve the following two optimization problems:

$$
\begin{aligned}
T'_1(e) &= \min_{\substack{s \in \{1,\ldots,S\} \\ e_{\text{sub}} + E_r(s) = e}} \big\{\hat{T}_{\text{sub}}(e_{\text{sub}}) \times s\big\} \\
\text{and} \quad E'_1(t) &= \min_{\substack{s \in \{1,\ldots,S\} \\ t_{\text{sub}} \times s = t}} \big\{\hat{E}_{\text{sub}}(t_{\text{sub}}) + E_r(s)\big\} \ .
\end{aligned} \qquad (10)
$$

We remark here that E'_1 and T'_1 are both non-increasing since $\hat{E}_{\text{sub}}$ and $\hat{T}_{\text{sub}}$ are non-increasing using Claim 5.4.

Each of these two problems can be solved by 1) computing the optimal value within each linear segment defined by $(\hat{E}_{\text{sub},k}, \hat{T}_{\text{sub},k})$ and $(\hat{E}_{\text{sub},k+1}, \hat{T}_{\text{sub},k+1})$, and 2) returning the smallest optimal value across all linear segments.

Suppose that we are computing $T'_1(e)$ given an error e. In the linear piece of $\hat{T}_{\text{sub}} = a e_{\text{sub}} + b$ (here a and b are the slope and intercept of the linear segment), we have $e_{\text{sub}} = e - E_r(s)$. The objective that we are minimizing therefore becomes univariate with respect to s:

$$
\hat{T}_{\text{sub}} \times s = (a e_{\text{sub}} + b) \times s = (a(e - E_r(s)) + b) \times s \ . \qquad (11)
$$

[6] Here we have ignored the running time for the sampling procedure in the reduction node, as it is often negligible in comparison to other computations in the program. It is possible to add this sampling time to the formula for TIME in a straightforward manner.

[7] We extend this analysis to other types of reduction nodes in Section 8.

The calculation of s is a simple univariate optimization problem that we can solve quickly using our expression for $E_r(s)$.[8] Comparing the optimal answers from all of the linear pieces gives us an efficient algorithm to determine $T'_1(e)$, and similarly for $E'_1(t)$. This algorithm runs in time linear in the number of pieces n_p (recall Eq.(8)) in our bi-dimensional discretization. This finishes the computation of (E'_1, T'_1) in Figure 6. We next show that (E'_1, T'_1) accurately approximates (E, T):

Claim 5.7. *(E'_1, T'_1) is an $(\varepsilon, \varepsilon_E)$-approximation to (E_1, T_1).*

Proof. Because $\hat{T}_{\rm sub}$ approximates $T_{\rm sub}$, we know that for any $e_{\rm sub}$, $0 \le \hat{T}_{\rm sub}(e_{\rm sub}) - T_{\rm sub}(e_{\rm sub}) \le \varepsilon T_{\rm sub}(e_{\rm sub})$, and this gives:

$$T'_1(e) = \min_{\substack{s\in\{1,\dots,S\}\\ e_{\rm sub}+E_r(s)=e}} \left\{\hat{T}_{\rm sub}(e_{\rm sub}) \times s\right\}$$
$$\le \min_{\substack{s\in\{1,\dots,S\}\\ e_{\rm sub}+E_r(s)=e}} \left\{(1+\varepsilon)T_{\rm sub}(e_{\rm sub}) \times s\right\} = (1+\varepsilon)T_1(e).$$

Similarly, this also gives that $T'_1(e) \ge T_1(e)$ using $0 \le \hat{T}_{\rm sub}(e_{\rm sub}) - T_{\rm sub}(e_{\rm sub})$.

Using a similar technique, we can also prove that $E'_1(t) \ge E_1(t)$ and $E'_1(t) \le (1+\varepsilon)E_1(t) + \varepsilon_E$. Therefore, the (E'_1, T'_1) curve $(\varepsilon, \varepsilon_E)$-approximates the exact tradeoff curve (E_1, T_1). □

We next further bi-dimensionally discretize the curve (E'_1, T'_1) that we obtained into $(\hat{E}'_1, \hat{T}'_1)$ using the same discretization parameter $(\varepsilon, \varepsilon_E)$. A discretization of an approximate curve is still approximate (see Section Section 5.2.3). We therefore conclude that

Claim 5.8. *$(\hat{E}'_1, \hat{T}'_1)$ is a $(2\varepsilon+\varepsilon^2, 2\varepsilon_E)$-approximation to (E_1, T_1).*

5.3.2 Approximating (E, T): probabilistic choice of s

Now, we define (E, T) to be the exact tradeoff curve (E_1, T_1) of the *entire* program, assuming that we can choose s probabilistically. We claim:

Claim 5.9. *(E, T) is the* convex envelope *of (E_1, T_1).*

Proof. We first prove the claim that T is the convex envelope of T_1. The proof for E is similar. One side of the proof is straightforward: every weighted combination of points on T_1 should lie on or above T, because this weighted combination is one candidate randomized configuration that chooses s probabilistically, and T is defined to be the optimal curve that takes into account all such randomized configurations.

For the other side of the proof, we need to show that every point on T is a weighted combination of points on T_1. Let us take an arbitrary point $(e, T(e))$. Suppose that $T(e)$ is achieved when the optimal probabilistic choice of s is $\{(s_i, p_i)\}_{i\ge1}$ at reduction node r, where we choose s_i with probability p_i and when s_i is chosen, the overall error-time incurred is (e_i, t_i). Therefore, we have $e = \sum_i p_i e_i$ and $T(e) = \sum_i p_i t_i$.

Because $T(e)$ is the exact optimal tradeoff curve, each t_i is also minimized with respect to e_i and the fixed choice of s_i. This is equivalent to saying that (e_i, t_i) lies on the curve (E_1, T_1), i.e., $T_1(e_i) = t_i$. This implies that T is the convex envelope of T_1. □

In general, computing the convex envelope of an arbitrary function (E_1, T_1) may be hard, but thanks to our piecewise-linear discretization, we can compute the convex envelope of $(\hat{E}'_1, \hat{T}'_1)$ easily. Let us denote the convex envelope of $(\hat{E}'_1, \hat{T}'_1)$ by $(\hat{E}', \hat{T}')$. In fact, $(\hat{E}', \hat{T}')$ can be computed in time $O(n_p \log n_p)$ because $(\hat{E}'_1, \hat{T}'_1)$ contains only n_p endpoints.

[8] Note that for each different type of reduction code, the optimization procedure for this univariate optimization can be hardcoded.

Since $(\hat{E}'_1, \hat{T}'_1)$ is a $(2\varepsilon + \varepsilon^2, \varepsilon_E)$-approximation to (E_1, T_1) by Claim 5.8, we should expect the same property to hold for their convex envelopes:

Claim 5.10. *$(\hat{E}', \hat{T}')$ is a $(2\varepsilon+\varepsilon^2, 2\varepsilon_E)$-approximation to (E, T).*

Proof. By the definition of convex envelope, for all time t, there exists some $\alpha \in [0, 1]$ such that $E(t) = \alpha E_1(t_1) + (1-\alpha)E_1(t_2)$ and $\alpha t_1 + (1-\alpha)t_2 = t$. Then,

$$\begin{aligned}\hat{E}'(t) &\le \alpha\hat{E}'_1(t_1) + (1-\alpha)\hat{E}'_1(t_2)\\ &\le 2\varepsilon_E + (1+2\varepsilon+\varepsilon^2)(\alpha E_1(t_1) + (1-\alpha)E_1(t_2))\\ &= 2\varepsilon_E + (1+2\varepsilon+\varepsilon^2)E(t) \ ,\end{aligned}$$

where the first inequality uses the fact that $\hat{E}'$ is the convex envelope of $\hat{E}'_1$, and the second uses the fact that $\hat{E}'_1$ approximates E_1.

At the same time, there exists some $\beta \in [0, 1]$ such that $\hat{E}'(t) = \beta\hat{E}'_1(t_3) + (1-\beta)\hat{E}'_1(t_4)$ and $\beta t_3 + (1-\beta)t_4 = t$. Then,

$$\begin{aligned}\hat{E}'(t) &= \beta\hat{E}'_1(t_3) + (1-\beta)\hat{E}'_1(t_4)\\ &\ge \beta E_1(t_3) + (1-\beta)E_1(t_4) \ge E(t) \ ,\end{aligned}$$

where the first inequality uses the fact that $\hat{E}'_1$ approximates E_1, and the second uses the fact that E is the convex envelope of E_1.

We can derive the two similar inequalities for the time function $\hat{T}'$ and conclude that $(\hat{E}', \hat{T}')$ is a $(2\varepsilon + \varepsilon^2, 2\varepsilon_E)$-approximation to (E, T). □

So far, we have finished all steps described in Figure 6. We have ended with a piecewise-linear tradeoff curve $(\hat{E}', \hat{T}')$ that $(2\varepsilon + \varepsilon^2, 2\varepsilon_E)$-approximates the exact tradeoff curve (E, T), taking into account the probabilistic choices at this reduction node as well.

5.4 The Final Dynamic Programming Algorithm

We next show how to compute the approximate error-time tradeoff curve for *any* program in our model of computation. We first present the algorithm, then we discuss how to choose the discretization parameters ε and ε_E and how the discretization errors compose.

If the program has no reduction nodes, we can simply apply the analysis from Section 5.1. Otherwise, there must exist at least one topmost reduction node r whose input is computed from computation nodes only. Assume (E, T) is the exact error-time tradeoff curve for the output of r. Applying Stage 2 (Section 5.3) to the subprogram rooted at r, we can efficiently find some piecewise-linear curve $(\hat{E}', \hat{T}')$ that accurately approximates (E, T).

Recall that this piecewise-linear curve $(\hat{E}', \hat{T}')$ is convex (since it is a convex envelope). If we pick all of its at most n_p endpoints on the curve $\mathcal{P} = \{(E_{r,i}, T_{r,i})\}_{i=1}^{n_p}$, then 1) every point on the curve can be spanned by at most two points in $\mathcal{P}$ because the function is piecewise linear, and 2) all points that can be spanned by $\mathcal{P}$ lie above the curve because the function is convex.

The two observations 1) and 2) above indicate that we can *replace* this reduction node r along with all computation nodes above it, by a single function node f_r such that its i-th substitute implementation $f_{r,i}$ gives an error of $E_{r,i}$ and a running time of $T_{r,i}$. Observation 1) indicates that every error-time tradeoff point on $(\hat{E}', \hat{T}')$ can be implemented by a probabilistic mixture of $\{f_{r,i}\}_{i=1}^{n_p}$, and observation 2) indicates that every error-time tradeoff point that can be implemented, is no better than the original curve $(\hat{E}', \hat{T}')$. In sum, this function node achieves the same error-time tradeoff as the piecewise-linear curve $(\hat{E}', \hat{T}')$.

This completes the description of the algorithm. We can recursively choose a reduction node with only computation nodes above it, replace the subtree rooted at the reduction node with a function node, continue until we have no reduction nodes left, and compute the final error-time tradeoff curve. We next consider the accuracy and running time of this recursive algorithm.

Maximum Error Propagation Factor (*MEPF*). We use a value MEPF to bound the cumulative additive error of the optimized program. Specifically, we choose MEPF so that a local additive error e at some node in the computation will produce at most a $e \times \mathrm{MEPF}$ additive error at the final output. Specifically, we set MEPF to be the maximum β_u over all function nodes f_u in the program.

Accuracy. Our algorithm repeatedly approximates error-time curves with their discretizations. This typically occurs twice during the analysis of each reduction node. If the inputs of a reduction node come from some computation node, the first discretization approximates the tradeoff curve for that computation node (we do not need this step if inputs come from some other reduction node.) The second discretization approximates the tradeoff curve for the outputs of the reduction node. By bounding the total error introduced by all of these approximations, we show that our algorithm produces a good approximation to the exact error-time curve.

We remark that there are two very different types of errors being discussed here: errors that we trade off against time in our approximate computation and errors that our optimization algorithm incurs in computing the optimal error-time tradeoff curve for the approximate computation. For clarity, we shall refer to the former as *computation error* and to the latter as *optimization error*.

If we choose parameters $(\varepsilon, \varepsilon_E)$ for both discretizations of a single reduction node r, the argument in Section 5.2.3 shows that we obtain a $(2\varepsilon+\varepsilon^2, 2\varepsilon_E)$-approximation to its error-time tradeoff curve.

A computation node may amplify the computation error, which will result in a corresponding increase in the optimization error of our approximation to it. However, we can bound the total effect of all computation nodes on our optimization error using our linear error propagation assumption. In particular, we can study the effect of a single reduction node's $(2\varepsilon+\varepsilon^2, 2\varepsilon_E)$-approximation on the program's overall additive and multiplicative optimization error:

- The additive error $2\varepsilon_E$ is multiplied by the computation nodes' Lipschitz parameters. The result is an overall additive optimization error of at most $2\varepsilon_E \mathrm{MEPF}$.
- The multiplicative error is not affected, since both the actual computation error and our approximation to it are multiplied by the same factors. The overall multiplicative optimization error introduced is thus at most $2\varepsilon + \varepsilon^2$.

This bounds the optimization errors introduced by the discretization in our analysis of a single reduction node. We can add these up to bound the total optimization error.

We fix parameters $\varepsilon = \frac{\varepsilon'}{\Omega(n)}$ and $\varepsilon_E = \frac{\varepsilon'_E}{\Omega(n)\times\mathrm{MEPF}}$ for all of the discretizations, where n is the number of nodes in the original program. Since we have at most n reduction nodes, this sums up to a total multiplicative error of ε' and additive error of ε'_E (both of these errors are optimization errors) in the final approximation of the exact error-time tradeoff curve of the whole program. Our overall optimization algorithm thus produces a piecewise-linear curve $(\hat{E}, \hat{T})$, that $(\varepsilon', \varepsilon'_E)$-approximates the exact error-time tradeoff curve.[9]

[9] If we choose ε'_E to be the smallest unit of error that we care about, then this essentially becomes an ε' multiplicative approximation to the actual error-time tradeoff curve.

Time Complexity. The most expensive operation in our optimization algorithm is solving the multiple linear programs that the algorithm generates. We next bound the number of linear programs that the algorithm generates.

Recall by Eq.(8) that n_p is the number of pieces in our bi-dimensional discretizations. By our choices of ε and ε_E:

$$n_p = O\Big(\frac{1}{\varepsilon}\big(\log E_{\max} + \log\frac{1}{\varepsilon_E} + \log T_{\max} + \log\frac{1}{T_{\min}}\big)\Big)$$
$$= O\Big(\frac{n}{\varepsilon'}\big(\log E_{\max} + \log\frac{n\mathrm{MEPF}}{\varepsilon'_E} + \log T_{\max} + \log\frac{1}{T_{\min}}\big)\Big)\ .$$

For each reduction node, we solve a linear program for each of the n_p points in its discretization. We therefore run the LP solver $O(n \times n_p)$ number of times. The number of variables in these linear program is $O(n \times n_p)$, since each node may have n_p implementations (which occurs when we replace a reduction node with its bi-dimensional discretization). This yields:

Theorem 5.11. *Our proposed recursive algorithm calls the LP solver $O(n \times n_p)$ times, each time with $O(n \times n_p)$ variables. The algorithm produces a $(1+\varepsilon')$-approximation to the exact error-time tradeoff curve, and thus approximately solves Question 1.*

Note that in practice we can find constants that appropriately bound $T_{\min}$, $T_{\max}$, $E_{\max}$, MEPF and ε_E.[10] In addition, all those numbers stay *within log functions*. We can therefore assume that in practice:

$$n_p \approx O\big(\frac{n\log n}{\varepsilon'}\big)\ .$$

We also note that 1) the worst-case time complexity of linear programming is polynomial, and 2) efficient linear programming algorithms exist in practice.

6. Optimization Algorithm for Question 2

At this point, we have approximately solved the error-time tradeoff curve problem in Question 1. Given an overall error tolerance Δ, we have an efficient algorithm that approximates the optimal running time $T(\Delta)$ up to a multiplicative factor $(1+\varepsilon)$.

We next show that our algorithm is constructive: it uses substitution and sampling transformations to obtain a randomized program P' with expected error bound Δ that runs in expected time no more than $(1+\varepsilon)T(\Delta)$. The algorithm therefore answers Question 2 approximately. Our proof first considers the two simpler cases that we introduced in Section 5.

Stage 1: Computation Nodes Only. If the program has no reduction nodes, then the solution $\boldsymbol{x}$ to the linear program gives explicitly the probability of choosing each implementation at each function node, which in turn gives us a randomized configuration for the randomized program P'.

Stage 2: A Single Reduction Node. If the program consists of computation nodes that generate the input for a single reduction node (this reduction node is therefore the root of the program), recall that we have perform two discretizations as illustrated in Figure 6.

Specifically, let (E, T) be the exact error-time tradeoff curve of the final output, which we have approximated by a piecewise-linear curve $(\hat{E}', \hat{T}')$ in the last step of Figure 6. We want to obtain a program P' whose error-time pair is $(\Delta, \hat{T}'(\Delta))$, because $\hat{T}'$ is guaranteed to be a $(1+\varepsilon)$-approximation to T.

[10] For example, we may lower bound $T_{\min}$ with the clock cycle time of the machine running the computation, ε_E with the smallest unit of error we care about, upper bound $T_{\max}$ with the estimated lifetime of the machine running the computation, $E_{\max}$ with the largest value representable on the machine running the computation, and MEPF with the ratio between the largest and smallest value representable on the machine running the computation.

By piecewise linearity, the point $(\Delta, \hat{T}'(\Delta))$ lies on some linear segment of $(\hat{E}', \hat{T}')$:

$$\hat{T}'(\Delta) = \lambda \hat{T}'(e_1) + (1-\lambda)\hat{T}'(e_2)$$
$$\Delta = \lambda e_1 + (1-\lambda) e_2 \ ,$$

where $[e_1, e_2]$ are the two endpoints of that segment. To achieve an error-time pair $(\Delta, \hat{T}'(\Delta))$, we can let the final program P' run 1) with probability λ some randomized program P'_1 whose error-time pair is $(e_1, \hat{T}'(e_1))$, and 2) with probability $1-\lambda$ some other randomized program P'_2 whose error-time pair is $(e_2, \hat{T}'(e_2))$. We next verify that both P'_1 and P'_2 can be constructed explicitly. We focus on the construction of P_1.

The goal is to construct P'_1 with error-time pair $p = (e_1, \hat{T}'(e_1))$. Note that p is an endpoint of $(\hat{E}', \hat{T}')$, and thus is also an endpoint of $(\hat{E}'_1, \hat{T}'_1)$. Since $(\hat{E}'_1, \hat{T}'_1)$ is a discretization of (E'_1, T'_1), p is also a point on the curve (E'_1, T'_1). We can therefore write $p = (e_1, T'_1(e_1))$.

Recall that we obtained $T'_1(e_1)$ by exactly solving the univariate optimization problem defined in Eq.(11). Because this solution is constructive, it provides us with the optimal reduction factor s to use at the reduction node. Substituting this optimal value s into Eq.(10), we obtain the error-time pair $(e_{\rm sub}, \hat{T}_{\rm sub}(e_{\rm sub}))$ that we should allocate to the subprogram (without the reduction node). We therefore only need to construct a program for the subprogram whose error-time pair is $(e_{\rm sub}, \hat{T}_{\rm sub}(e_{\rm sub}))$.

Because $\hat{T}_{\rm sub}$ is a piecewise-linear discretization of $T_{\rm sub}$, we can obtain a program whose error-time pair is $(e_{\rm sub}, \hat{T}_{\rm sub}(e_{\rm sub}))$ by combining at most two points on the $(E_{\rm sub}, T_{\rm sub})$ curve. The linear program described above implements this $(E_{\rm sub}, T_{\rm sub})$ curve. This completes the construction of P'.

We have therefore shown that as part of the algorithm for Stage 2 (see Section 5.3), we can indeed construct a program P' with the desired error and time bounds $(\Delta, \hat{T}'(\Delta))$.

Putting It All Together. We prove, by induction, that for a program with an arbitrary number of reduction nodes, we can obtain a randomized program P' with expected error bound Δ and expected running time no more than $(1+\varepsilon)T(\Delta)$. We are done if there is no reduction node (by Stage 1). Otherwise, suppose that we have n_r reduction nodes.

We will substitute one of the reduction nodes r, along with all computations above it, by a function node f_r with piecewise-linear tradeoff $(\hat{E}', \hat{T}')$. This function node has a set of implementations $\{f_{r,i}\}_{i=1}^{n_p}$, and in our Stage 2, we have actually shown that each $f_{r,i}$, being a point on the $(\hat{E}', \hat{T}')$ curve, is achievable by some explicit randomized program P_i.

In other words, this new function node f_r is a "real" function node: every approximate implementation $f_{r,i}$ has a corresponding randomized program P_i that can be constructed by our algorithm. This reduces the problem into a case with $n_r - 1$ reduction nodes. Using our induction hypothesis, we conclude that for any Δ, we can construct a program P' that runs in expected time no more than $(1+\varepsilon)T(\Delta)$.

7. Optimization Algorithm for Question 3

We next describe how to modify our algorithm if one instead is interested in the time-variance tradeoff.

7.1 Changes to Function Nodes

For each function node f_u, the algorithm for Question 1 works with an error-time pair $(T_{u,i}, E_{u,i})$ for each implementation $f_{u,i}$. Now we instead work with a pair $(T_{u,i}, V_{u,i})$, where $V_{u,i}$ is the variance (i.e., the expected squared error):

$$\forall \boldsymbol{x}, \quad \mathbb{E}[|f_u(\boldsymbol{x}) - f_{u,i}(\boldsymbol{x})|^2] \le V_{u,i} \ , \tag{12}$$

where the expectation is over the the randomness of $f_{u,i}$ and f_u.

We also assume that the variance propagation function is linear with coefficients $\alpha_1, \dots, \alpha_m$. In other words, if f_u has an arity of m, and each input x_i is approximated by some input $\hat{x}_i$ such that $\mathbb{E}[|x_i - \hat{x}_i|^2] \le v_i$, then the final variance:

$$\mathbb{E}\big[|f_u(x_1,\dots,x_m) - f_{u,i}(\hat{x}_1,\dots,\hat{x}_m)|^2\big] \le \big(\sum_j \alpha_j v_j\big) + 2V_{u,i} \ . \tag{13}$$

We next check that a large class of meaningful functions satisfies the above definition. Note that if f_u is a deterministic Lipschitz-continuous function (with respect to vector $\boldsymbol{L} = (L_1, \dots, L_m)$):

$$\begin{aligned}
&\mathbb{E}\big[|f_u(\boldsymbol{x}) - f_{u,i}(\hat{\boldsymbol{x}})|^2\big] \\
&= \mathbb{E}\big[|(f_u(\boldsymbol{x}) - f_u(\hat{\boldsymbol{x}})) + (f_u(\hat{\boldsymbol{x}}) - f_{u,i}(\hat{\boldsymbol{x}}))|^2\big] \\
&\le 2\mathbb{E}\big[|f_u(\boldsymbol{x}) - f_u(\hat{\boldsymbol{x}})|^2 + |f_u(\hat{\boldsymbol{x}}) - f_{u,i}(\hat{\boldsymbol{x}})|^2\big] \\
&\le 2\mathbb{E}\big[|f_u(\boldsymbol{x}) - f_u(\hat{\boldsymbol{x}})|^2\big] + 2V_{u,i} \\
&\le 2\mathbb{E}\big[(\textstyle\sum_i L_i |x_i - \hat{x}_i|)^2\big] + 2V_{u,i} \\
&\le 2m\mathbb{E}\big[\textstyle\sum_i L_i^2 (x_i - \hat{x})^2\big] + 2V_{u,i} \\
&\le 2m \textstyle\sum_i L_i^2 v_i + 2V_{u,i} \ .
\end{aligned} \tag{14}$$

So let $\alpha_i = 2mL_i^2$ in Eq.(13). If f_u is probabilistic and each deterministic function in its support is Lipschitz-continuous (perhaps for different $\boldsymbol{L}$'s), then we can similarly set the final $\alpha_i = 2m\mathbb{E}[L_i^2]$ using the expectation.[11]

7.2 Changes to Reduction Nodes

Recall that for a reduction node r with reduction factor S_r, we can decrease this factor to a smaller value $s_r \in \{1, \dots, S_r\}$ at the expense of introducing some additive sampling error $E_r(s_r)$. In our Question 3, we have a different variance bound $V_r(s_r) = (B-A)^2 \frac{S_r - s_r}{s_r(S_r - 1)}$, whose proof can be found in the full version of this paper.

Lemma 7.1. *Given m numbers $x_1, x_2, \dots, x_m \in [A, B]$, randomly sampling s of the numbers $x_{i_1}, \dots, x_{i_s}$ (without replacement) and computing the sample average gives an approximation to $\frac{x_1 + \dots + x_m}{m}$ with the following variance guarantee:*

$$\begin{aligned}
&\mathbb{E}_{i_1,\dots,i_s}\Big[\big|\frac{x_{i_1} + \dots + x_{i_s}}{s} - \frac{x_1 + \dots + x_m}{m}\big|^2\Big] \\
&\le (B-A)^2 \frac{m-s}{s(m-1)} \ .
\end{aligned}$$

Similar to Corollary 3.2, we also verify that an averaging node satisfies our linear variance propagation assumption, due to its Lipschitz continuity We defer the proof to the full version of this paper.

Corollary 7.2. *Consider an averaging node that will pick s random samples among all m inputs where each input $\hat{x}_j$ has bounded variance $\mathbb{E}[|\hat{x}_j - x_j|^2] \le v_j$. Then:*

$$\begin{aligned}
&\mathbb{E}_{i_1,\dots,i_s,\hat{x}_1,\dots,\hat{x}_m}\Big[\big|\frac{\hat{x}_{i_1} + \dots + \hat{x}_{i_s}}{s} - \frac{x_1 + \dots + x_m}{m}\big|^2\Big] \\
&\le 2(B-A)^2 \frac{m-s}{s(m-1)} + \frac{2}{m}\sum_{i=1}^m v_i \ .
\end{aligned}$$

[11] Because the randomness within f_u is independent of the randomness for input error $\hat{x}_i - x_i$, we have $\mathbb{E}[L_i^2 (x_i - \hat{x}_i)^2] = \mathbb{E}[L_i^2]\mathbb{E}[(x_i - \hat{x}_i)^2]$.

7.3 Putting Things Together

As before, if there is no reduction node, we can write the variance in the following linear form:

$$\text{ERROR2}(\boldsymbol{x}) = \sum_u \sum_i (x_{u,i} \cdot V_{u,i} \cdot \beta_u) \quad .$$

The amplification factors β_u can be pre-computed using the linear variance propagation factors α_i.

The expressions for the reduction node are slightly changed. Instead of Eq.(9), we now have

$$\text{TIME} = T_{\text{sub}} \times s$$
$$\text{ERROR2} = 2V_{\text{sub}} + 2V_r(s) \ \ .$$

One can use almost the same arguments as in Section 5 to show that the optimization problem with respect to one reduction node can be reduced to optimizing a univariate function. This yields a fully polynomial-time approximation scheme (FPTAS) that solves Question 3. The solution for Question 4 can be derived similarly if one follows the arguments in Section 6.

8. Other Reduction Nodes

We have stated our main theorems for programs that contain only averaging reduction nodes. More generally, our algorithm supports any reduction node that can be approximated using sampling. This includes, for example, variances, ℓ^p norms, and order statistics functions. In this section we discuss the analysis of some additional reduction nodes.

Summation Reduction Nodes. For a summation node with reduction factor S_r (i.e., S_r numbers to sum up), the sampling procedure randomly selects s_r numbers, then outputs the partial sum multiplied by $\frac{S_r}{s_r}$. The output is therefore an unbiased estimator of the exact sum. The derivation of the expressions for the expected error and the variance of the error closely follows the derivation for averaging nodes. The final expressions are the corresponding averaging expressions multiplied by S_r.

Minimization and Maximization Reduction Nodes. Note that all reasonable sampling techniques may, with some non-negligible probability, discard the smallest input value. We therefore consider *percentile minimization nodes*, which have two parameters $q \in (0,1)$ and B. If the output is any of the smallest $\lfloor qm \rfloor$ numbers in the input (here m is the number of inputs), the error is 0. Otherwise, the error is B. We also define $m' = \lceil (1-q)m \rceil$.

Lemma 8.1. *For a percentile minimization (or maximization) node r with parameters $q \in (0,1)$ and B, sampling s_r of m input elements gives an expected error of*

$$E_r(s_r) = \begin{cases} \frac{\binom{m'}{s_r}}{\binom{m}{s_r}} B & \text{if } s_r \le m' \\ 0 & \text{if } s_r > m', \end{cases}$$

which is a convex function. It is also possible to define a simpler approximate bound if m is large or sampling is done without replacement: $E_r(s_r) \approx (1-q)^{s_r} B$.

Proof. There are a total of $\binom{m}{s_r}$ ways to take samples of size s_r. $\binom{m'}{s_r}$ of these samples will contain no element in the smallest q fraction of the original S_r inputs. The probability that a sample contains no element that lies in this smallest q fraction is therefore $\frac{\binom{m'}{s_r}}{\binom{m}{s_r}}$. The expected error is this probability multiplied by B. One can verify that this is a convex function by taking the second-order discrete derivative. When m is large, the probability can be approximated by $(1-q)^{s_r}$, which is obviously convex. □

Also, the error propagates linearly (with a constant factor of 1) for such nodes. In other words, if all inputs to a minimization (or maximization) node are subject to an error of $\mathbb{E}[|x_i - \hat{x}_i|] \le \delta$, the output will have error $\delta + E'_r(s_r)$. Now, if we replace $E_r(s_r)$ in Eq.(9) with this function we just derived, all of the analysis in Section 5 goes through unchanged.

It is also straightforward to verify that if one uses the variance of the error, $V_r(s_r) = E_r(s_r)B$.

Argmin and Argmax Reduction Nodes. We can similarly define percentile argmin and argmax nodes, where the output value is a pair (i, v) in which i is the index for the smallest/largest element and v is the actual minimum or maximum value. The error depends exclusively on the element v. We can output the index i, but it can not be used as a value in arithmetic expressions.

9. Related Work

Empirical Accuracy-Performance Tradeoffs. Profitably trading accuracy for performance is a well known practice, often done at an algorithmic or system level. For many computationally expensive problems, researchers have developed approximation, randomized, or iterative algorithms which produce an approximate solution with guaranteed error bounds [25]. Researchers have also proposed a number of system-level techniques that specifically aim to trade accuracy for performance, energy, or fault tolerance. The proposed techniques operate at the level of hardware [4, 17, 34], system software [9, 13, 17, 20, 21], and user applications [2, 3, 6, 14, 24, 27, 28, 30, 31, 33].

Previous research has explored the tradeoff space by running and comparing the results of the original and transformed programs on either training inputs [12, 27, 28, 31] or online for chosen production inputs [2, 3, 13, 33]. The result of this exploration is often an empirical approximation of the accuracy-performance tradeoff curve. This approximation may be valid only for inputs similar to those used to construct the empirical curve. In some cases the approximation comes with empirical statistical accuracy bounds [27, 28]. In other cases there are no accuracy bounds at all. This paper, in contrast, statically analyzes the computation to produce a $(1+\epsilon)$-approximation to the exact tradeoff curve. This approximation provides guaranteed probabilistic accuracy bounds that are valid for all legal inputs.

Probabilistic Accuracy Bounds For Single Loops. Researchers have recently developed static analysis techniques for characterizing the accuracy effects of loop perforation (which transforms loops to execute only a subset of their iterations) [22, 23] or loops with approximate function memoization (which transforms functions to return a previously computed value) [6]. These techniques analyze single loops and do not characterize how the accuracy effects propagate through the remaining computation to affect the output. This paper, in contrast, presents a technique that characterizes and exploits the accuracy-performance curve available via substitution and sampling transformations applied to complete computations. Unlike previous research, the techniques presented in this paper capture how global interactions between multiple approximate transformations propagate through the entire computation (instead of just a single loop) and does not require specification of probability distributions of the inputs.

Analytic Properties of Programs. Researchers have developed techniques to identify continuous or Lipschitz-continuous programs [5, 6, 18, 19, 26]. Identified applications include differential privacy [5, 6, 26] and robust functions for embedded systems [6, 18, 19]. This paper, in contrast, presents techniques that apply accuracy-aware transformations to obtain new computations that occupy more desirable points on the underlying accuracy-performance tradeoff curve that the transformations induce.

Smooth interpretation [7, 8] uses a gradient descent based method to synthesize control parameters for imperative computer programs. The analysis returns a set of parameters that minimize the difference between the expected and computed control values for programs that control cyberphysical interactions.

Approximate Queries in Database Systems. Modern databases often enable users to define queries that operate on some subset of the records in a given table. Such queries come with no accuracy or performance guarantees. Researchers have explored multiple directions for supporting approximate queries with probabilistic guarantees. Approximate aggregate queries let a user specify a desired accuracy bound or execution time of a query [1, 15, 16]. The database then generates a sampling strategy that satisfies the specification [15, 16] or uses a cached sample, when applicable [1]. Online queries compute the exact answer for the entire data-set, but provide intermediate results and confidence bounds [11]. Probabilistic databases [35] operate on inherently uncertain data. The accuracy bounds of all queries (including aggregation) depend on the uncertainty of data.

These systems work with specific classes of queries defined in a relational model. They sample data but do not consider multiple function implementations. They also do not provide general mechanisms to achieve optimal accuracy-performance tradeoffs for sampling when processing complex nested queries.

10. Conclusion

Despite the central role that approximate computations play in many areas of computer science, there has been little research into program optimizations that can trade off accuracy in return for other benefits such as reduced resource consumption. We present a model of computation for approximate computations and an algorithm for applying accuracy-aware transformations to optimize these approximate computations. The algorithm produces a randomized program, which randomly selects one of multiple weighted alternative program configurations to maximize performance subject to a specified expected error bound.

Given the growing importance of approximation in computation, we expect to see more approximate optimization algorithms in the future. We anticipate that these algorithms may share many of the key characteristics of the computations and optimizations that we present in this paper: program transformations that trade off accuracy in return for performance, the ability to optimize programs in the presence of errors that propagate globally across multiple composed programming constructs, and randomization to improve performance and tractability.

Acknowledgements

We thank Michael Carbin, Stelios Sidiroglou, Jean Yang, and the anonymous reviewers for useful feedback on previous versions of this paper.

This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0843915, CCF-0905244, CCF-1036241 and IIS-0835652), the United States Department of Energy (Grant DE-SC0005288), and a Sloan Fellowship.

References

[1] S. Acharya, P. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

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

[3] W. Baek and T. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI '10*.

[4] L. Chakrapani, K. Muntimadugu, A. Lingamneni, J. George, and K. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: A mathematical foundation and preliminary experimental validation. In *CASES*, 2008.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, 2010.

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

[7] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *PLDI,* 2010.

[8] S. Chaudhuri and A. Solar-Lezama. Smoothing a program soundly and robustly. In *CAV*, 2011.

[9] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP*, 1999.

[10] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[11] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD/PODS*, 1997.

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

[13] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. Seec: A general and extensible framework for self-aware computing. Technical Report MIT-CSAIL-TR-2011-046, 2011.

[14] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures . Technical Report MIT-CSAIL-TR-2009-042, 2009.

[15] W.-C. Hou, G. Ozsoyoglu, and B. K. Taneja. Processing aggregate relational queries with hard time constraints. SIGMOD, 1989.

[16] Y. Hu, S. Sundara, and J. Srinivasan. Supporting time-constrained SQL queries in Oracle. VLDB, 2007.

[17] S. Liu, K. Pattabiraman, T. Moscibroda, and B. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. ASPLOS, 2011.

[18] R. Majumdar and I. Saha. Symbolic robustness analysis. In *RTSS '09*.

[19] R. Majumdar, I. Saha, and Z. Wang. Systematic testing for control applications. In *MEMOCODE*, 2010.

[20] J. Meng, S. Chakradhar, and A. Raghunathan. Best-Effort Parallel Execution Framework for Recognition and Mining Applications. In *IPDPS*, 2009.

[21] J. Meng, A. Raghunathan, and S. B. S. Chakradhar. Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution. In *IPDPS*, 2010.

[22] S. Misailovic, D. Roy, and M. Rinard. Probabilistic and statistical analysis of perforated patterns. Technical Report MIT-CSAIL-TR-2011-003, January 2011.

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

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

[25] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

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

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

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

[29] R. Rubinfeld. Sublinear time algorithms. In *Intl. Congress of Mathematicians*, 2006.

[30] 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 *PLDI*, 2011.

[31] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Tradeoffs With Loop Perforation. In *FSE, 2011*.

[32] K. Sohraby, D. Minoli, and T. Znati. *Wireless sensor networks: technology, protocols, and applications*. Wiley-Blackwell, 2007.

[33] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys*, 2007.

[34] P. Stanley-Marbell and D. Marculescu. Deviation-Tolerant Computation in Concurrent Failure-Prone Hardware. Technical Report ESR-2008-01, Eindhoven University of Technology, 2008.

[35] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Morgan-Claypool, 2011.