Statistical Algorithmic Profiling for Randomized Approximate Programs

Keyur Joshi, Vimuth Fernando, Sasa Misailovic
University of Illinois at Urbana-Champaign
{kpjoshi2, wvf2, misailo}@illinois.edu

***Abstract*—Many modern applications require low-latency processing of large data sets, often by using approximate algorithms that trade accuracy of the results for faster execution or reduced memory consumption. Although the algorithms provide probabilistic accuracy and performance guarantees, a software developer who implements these algorithms has little support from existing tools. Standard profilers do not consider accuracy of the computation and do not check whether the outputs of these programs satisfy their accuracy specifications.**

We present AXPROF, an algorithmic profiling framework for analyzing randomized approximate programs. The developer provides the accuracy specification as a formula in a mathematical notation, using probability or expected value predicates. AXPROF automatically generates statistical reasoning code. It first constructs the empirical models of accuracy, time, and memory consumption. It then selects and runs appropriate statistical tests that can, with high confidence, determine if the implementation satisfies the specification.

We used AXPROF to profile 15 approximate applications from three domains – data analytics, numerical linear algebra, and approximate computing. AXPROF was effective in finding bugs and identifying various performance optimizations. In particular, we discovered five previously unknown bugs in the implementations of the algorithms and created fixes, guided by AXPROF.

I. INTRODUCTION

Modern applications, such as machine learning, data analytics, computer vision, financial forecasting, and content search require low-latency processing of massive data sets. To meet such demands, researchers have developed various approximate algorithms, data structures, and systems software that trade accuracy for performance and/or memory consumption.

Many emerging approximate algorithms come with analytically derived specifications of accuracy. The specifications are typically probabilistic – e.g., an algorithm will return the desired result with high probability. As an example, locality-sensitive hashing algorithm [1], [2] can find nearest neighbors in a set of points, by using smart hashing to group similar points. It guarantees to return the most similar points with high probability. Probabilistic specifications have been proposed for applications in areas as diverse as theoretical computer science [3], [4], [5], [6], [7], [8], numerical computing [9], [10], [11], [12], databases [13], [14], [15], [16], [17], and compilers and hardware architectures [18], [19], [20], [21], [22], [23].

Despite many rigorous specifications, a software developer who needs to implement, test, and tune these randomized programs and systems has virtually no tool support for this effort. Standard profilers only track and build models of run time and memory consumption for individual inputs [24], [25], [26] or build performance models for multiple input sizes in the case of *algorithmic profiling* [27], [28]. Researchers have also given guidelines for how to rigorously apply statistical testing in software engineering, e.g., [29], but the process is manual, and the developer may end up with inflexible and overly conservative choices of test parameters.

There are numerous tasks that the developer needs to perform manually: infer the properties of the mathematical (probabilistic) specification, write code to check this specification, decide on the appropriate statistical test and its parameters (e.g., confidence or power), provide appropriate inputs, and interpret obtained statistical metrics. Frustrated by such manual effort, developers often resort to ad-hoc testing. Moreover, manually written test code can have various subtle errors that prevent the discovery of errors in the implementation. A more promising alternative is to automate these tasks with profiling and testing frameworks.

Our Work. We present AXPROF, an algorithmic profiling framework for analyzing accuracy, execution time, and memory consumption of approximate programs. AXPROF constructs statistical models of accuracy, time, and memory, checks if any of them deviate from the algorithm specification, and if so, warns the developer. AXPROF is available at www.axprof.org.

The key novelty of AXPROF is the automatic generation of the accuracy checking code from a high-level probabilistic specification (Section IV). The developer-written specification highly resembles the mathematical specification that algorithm designers provide as a part of their theoretical study. AXPROF supports two general probabilistic predicates:

- ***Probability Predicate:*** It specifies that the probability that the output returned by the approximate program satisfies a condition is below, above, or equal to a certain threshold.
- ***Expectation Predicate:*** It specifies that the output's expected value is below, above, or equal to a certain threshold.

These two predicates can capture the key properties of many representative randomized and approximate computations. For instance, they are expressive enough to capture the majority of the accuracy specifications of the randomized algorithms from [30]. They can also model common accuracy specifications from other domains, such as numerical linear algebra and approximate computing.

An important concern of AxProf's language design is to present the specifications in an unambiguous manner. In general, probabilistic specifications can be defined over different sets of events (e.g., runs or inputs), which may require different sampling procedures. In AxProf, a developer explicitly writes if a probabilistic specification is over inputs, items within an input, or runs. For each specification, AxProf automatically 1) selects a proper statistical test, 2) generates checking code that aggregates the outputs and applies the selected test, and 3) determines the number of samples to achieve a desired level of statistical confidence.

Testing randomized programs often requires a large number of (concrete) inputs. To automatically produce representative inputs, we provide several input generators for scalars, vectors, and matrices, that allow for various input properties to be modified: the difference in the frequency of values, order of data, or various forms of correlations. We present a dynamic analysis that infers which of these properties have a significant impact on the algorithm's accuracy (Section V).

Results. We evaluated AxProf on a set of 15 programs that implement well-known randomized approximate computations from the domains of data analytics, numerical linear algebra, and approximate computing. Each application has an analytically derived specification of accuracy, performance, and memory consumption. We demonstrate that AxProf can help developers in two scenarios: 1) profiling to understand program behavior and 2) identifying potential implementation errors. Moreover, we demonstrate the effectiveness of the input generation analysis, which discovers the parameters of the input generator that affect output accuracy (Section VII).

AxProf helped us identify and fix previously-unknown problems with five different implementations of these algorithms. Our analysis shows that these problems could not have been identified with standard profilers that track only memory or runtime. The problems were caused by incorrect implementations of the algorithms or their key components like hash functions. We prepared pull requests for each of these problems. The implementation developers already accepted three of our pull requests.

AxProf also identified some implementations that make additional optimizations in their resource consumption. These optimizations may result in a different complexity of resource consumption than specified by the algorithm. For instance, the implementation developers may allocate resources dynamically (only when needed) or they may create polyalgorithms – compose multiple algorithms that work better for different input sizes, and switch algorithms dynamically.

These results demonstrate that AxProf's focus on accuracy analysis opens a new dimension in algorithmic profiling. Previous approaches for algorithmic profiling [25], [28] focused only on deriving models of performance. As such, they miss to characterize important accuracy-related properties of the emerging data-centric applications.

Contributions. The paper makes the following contributions:

- ⋆ ***Concept:*** We present algorithmic profiling for accuracy, performance, and resource consumption of approximate computations. We also present AxProf, a system that automates many algorithmic profiling tasks and pinpoints potential violations of algorithm specifications.
- ⋆ ***Accuracy Analysis:*** We present an approach for automatically generating statistical testing procedures from high-level probabilistic specifications in mathematical notation, as proposed by the algorithm authors. The generated procedures can pinpoint if the algorithm implementations significantly deviate from their specifications.
- ⋆ ***Evaluation:*** We present the evaluation of AxProf on a set of 15 approximate benchmarks from three application domains (data analytics, numerical linear algebra, and approximate computing). Our results show that 1) AxProf can be effectively used to find errors in randomized approximate programs and check for the correctness of the fixes and 2) AxProf can identify polyalgorithms and optimizations of resource usage.

II. Example

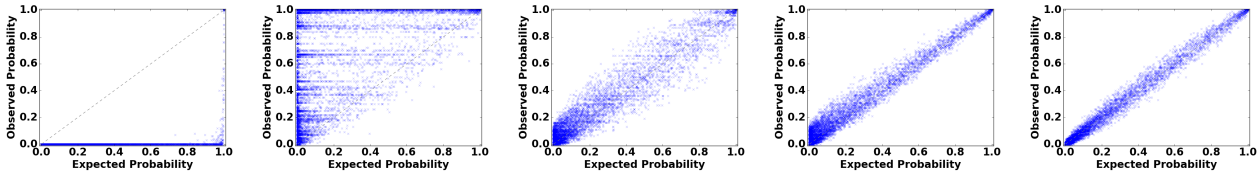
Locality Sensitive Hashing (LSH) [1], [2] is an algorithm for finding points that are near a given query point in multidimensional space. Instead of directly computing the distance of the query point to every other point in the set, LSH maintains a compact representation of the points and their locations using a set of hash maps. The keys of the maps are *hash signatures* and the values are the list of points with that signature.

To obtain a hash signature of a point, LSH calculates a "locality sensitive" hash of that point. Depending on the desired similarity metric between points (such as ℓ_1 distance, ℓ_2 distance, or cosine similarity), different "locality sensitive" hash function families exist which hash similar points to the same hash signatures with high probability.

When it receives a query, LSH calculates the query point's signature and returns all stored points with that signature. LSH can increase the number of similar points found by increasing the number of hash maps (l). LSH can also concatenate signatures from k different hash functions as the keys in each hash map to increase the probability that dissimilar points will be mapped to different bins. Each of these hash functions must be drawn uniformly at random from the same hash family.

AxProf Specification. We assign each point in the dataset an index. One representation of LSH output is a list of pairs of indices. The first index is the index of the query point and the second is the index of the detected neighbor. We use the same set of points as both data points and query points.

Suppose a hash function chosen uniformly at random from the desired hash function family puts a point d in the dataset and the query point q in the same map bin with probability $p_{d,q}$. This probability is calculated from the distance between the dataset point and the query point. Then, for all d and q, the probability that LSH will return d in the output for the query point q is $1 - (1 - p_{d,q}^k)^l$.



(a) Before (b) After Fix 1 (c) After Fix 2 (d) After Fix 3 (e) After Fix 4

Fig. 1: TarsosLSH – Comparison of accuracy of the implementation before and after bug fixes. Each point compares the expected and observed probability for a query-datapoint pair. Ideally all points should lie on the diagonal line.

In AXPROF, we write the full specification as:

```
1 Input list of (list of real);
2 Output list of (list of real);
3 TIME k*l*datasize;
4 SPACE l*datasize;
5 ACC forall di in indices(Input), qi in indices(Input) :
6         Probability over runs [ [qi, di] in Output ] ==
7         L1HashEqProb(Input[di], Input[qi], 10, k, l)
```

Lines 1 and 2 indicate the data types of the input and output, respectively. Lines 3 and 4 give the time and memory specification of LSH. As each point must be stored in each hash table, the memory usage is $O(ln)$, where n is the number of data points. Storing a point requires calculating lk hashes. The total time required to construct the hash tables is $O(lkn)$.

The last line gives the accuracy specification. Informally, it specifies that, for all possible pairs of indices over the input `(di,qi)`, the probability that a particular run of LSH has `[di,qi]` in its output is equal to the return value of `L1HashEqProb`, which is a user-defined function that calculates the expression $1-(1-p_{d,q}^{k})^{l}$.

AXPROF uses this specification to automatically generate code to check that the property holds. The code aggregates the outputs of the implementation over multiple runs. Then, for each pair of indices, it calculates the fraction of runs for which the pair is in the output. It compares this fraction against the return value of `L1HashEqProb` using the binomial test. Finally, it combines the results of the binomial tests for each pair of indices using Fisher's method. For time and memory, AXPROF generates code to perform statistical regression. The full details of code generation are in Section IV.

Testing the Implementation. We tested *TarsosLSH* [31], an implementation of LSH in Java that has its own testing framework and over 100 stars on GitHub. The algorithm can be configured through two parameters `k` and `l`, for which the developer specifies a list of values of interest. The number of points, `datasize`, can also be specified. We instruct AXPROF to uniformly generate random 2-dimensional points with each coordinate in the range $[-10, 10]$.

Identifying and Fixing Bugs. While profiling *TarsosLSH* for the ℓ_1 distance metric, AXPROF indicated errors for several values of k and l. We used the visualization feature of AXPROF and observed that many points were not being considered similar at all, as shown in Figure 1a. Each point represents a query-datapoint pair. The x and y coordinates of the point denote the expected and observed probability respectively. Ideally, all results should lie on the diagonal line.

This prompted us to investigate the hash function used for ℓ_1 distance. We found that there were several inaccuracies in the implementation and use of the hash function. We fixed a bug that occurred due to operator precedence, followed by a bug caused by incorrect assumptions about the rounding of floating point values in Java. Fixing the first bug led to the result shown in Figure 1b and fixing the second bug led to the result in Figure 1c. While the result in Figure 1c seemed to conform with the diagonal line as expected, AXPROF's statistical tests reported that the number of outliers was still too high, indicating the presence of more bugs.

On further investigation we found a bug in the method by which the implementation chose a hash function from the hash function family, and a bug in the method by which the outputs of the k different hash functions for a hash table were combined. Fixing the third bug led to the result shown in Figure 1d and fixing the fourth led to the result in Figure 1e. After fixing the fourth bug, AXPROF indicated that the implementation conformed with the accuracy specification.

An important point to note is that while the results in Figures 1c and 1d seem to be visually correct, AXPROF was able to accurately conclude that there were still unfixed bugs in the implementation via statistical testing.

The implementation included a test method which tested the algorithm with various parameters. However, there was an error in the tester code that miscounted the number of false negatives. This led the tester to overestimate the recall of the implementation, i.e., the percentage of nearby points that are correctly identified. A user that depended on the results of this test would mistakenly believe that the algorithm was implemented correctly. This further illustrates the need for automated tools like AXPROF.

III. AXPROF OVERVIEW

Figure 2 presents the overview of the system.

Inputs. AXPROF takes the following inputs:

- ***Implementation and Parameters:*** AXPROF takes an implementation of the algorithm to test and a set of algorithm configuration parameter ranges to be tested.
- ***Property Specification:*** The user provides an accuracy, time, and memory specification in a high-level language that resembles the mathematical specifications usually provided by the algorithm authors. AXPROF automatically generates code to check these specifications.

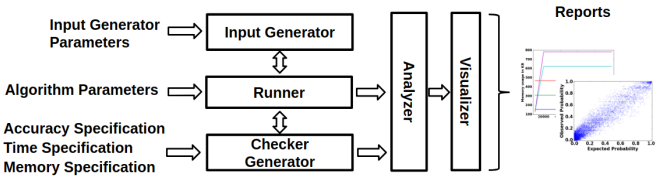


Fig. 2: Overview of AXPROF

- ***Input Generator Properties:*** AXPROF provides several input generators for scalar, vector, and matrix data. Alternatively, the user can provide a custom input generator. The user can allow AXPROF to infer interesting parameters that affect the output accuracy of the algorithm, or fix some or all of the parameters to values that the user wants to test.

Components. AXPROF has multiple components:

- ***Checker Generator*** takes an accuracy, time, and memory usage specification from the user and generates code to aggregate output data, along with time and memory usage data, and check that the data conforms to the specifications.
- ***Input Generator*** generates inputs to test the program. It can experiment with various input parameters to determine which ones affect the accuracy of the output.
- ***Runner*** executes the program on an AXPROF-provided input. It returns the generated output and resource consumption statistics to AXPROF.
- ***Analyzer*** uses the code generated by the *Checker Generator* to test whether the implementation conforms to the provided specifications, and issues a warning otherwise. AXPROF's analyses provide statistical guarantees that a warning may signify a real discrepancy with high probability. The developer can set parameters that influence how sensitive the statistical analyses are.
- ***Visualizer*** plots time, memory, and accuracy statistics for visual inspection by the user.

We present the statistical techniques for constructing models and checking whether they deviate from the ones provided in the specification for both accuracy and resource usage in Section IV. These techniques guide the Checker Generator (before profiling begins) and the Analyzer (as the profiler runs). We describe input generators in Section V.

Specification Language. The user writes specifications in a high-level language. Figure 3 presents its grammar. Specifications consist of type declarations, time expression, memory expression, and accuracy expression. The time and memory expressions are typical arithmetic expressions. Supported types are real numbers, matrices of real numbers, or collections (lists or maps). Knowing the types helps our generators to produce the correct code for the checkers and connect them with the rest of the framework.

The accuracy specification encodes two common predicates: 1) probability comparison (`Probability over` *Qualif*) and 2) expectation comparison (`Expectation over` *Qualif*). Both predicates explicitly define the probability space via *qualifiers*. The qualifier can be a list of items, a set of executions (*runs*), or a set of inputs. If the qualification is

$$
\begin{aligned}
c &\in \textit{Constants}\\
x &\in \textit{Vars} \cup \{\texttt{Input, Output}\}\\
f &\in \textit{Functions}\\
aop &\in \{+, -, *, /, \hat{}\}\\
bop &\in \{\&\&, ||\}\\
rop &\in \{==, >, \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
\textit{Spec} &::= \textit{TDclr}\ \texttt{TIME}\ \textit{DExpr}\ ;\ \texttt{SPACE}\ \textit{DExpr}\ ;\ \texttt{ACC}\ \textit{ASpec}\\
\textit{TDclr} &::= \texttt{Input}\ \textit{Type};\ \texttt{Output}\ \textit{Type};\ (x\ \textit{Type};)^*\ (f\ \textit{Type};)^*\\
\textit{Type} &::= \texttt{real} \mid \texttt{matrix} \mid \texttt{list of}\ \textit{Type} \mid\\
&\quad\ \texttt{map from}\ \textit{Type}\ \texttt{to}\ \textit{Type}\\
\textit{ASpec} &::= \texttt{Probability over}\ \textit{Qualif}\ [\textit{BExpr}]\ \textit{rop}\ \textit{DExpr} \mid\\
&\quad\ \texttt{Expectation over}\ \textit{Qualif}\ [\textit{DExpr}]\ \textit{rop}\ \textit{DExpr} \mid\\
&\quad\ \texttt{forall}\ \textit{Range}^{+} : \textit{ASpec} \mid\\
&\quad\ \texttt{let x} = \textit{DExpr}\ \texttt{in}\ \textit{ASpec}\\
\textit{Qualif} &::= \texttt{runs} \mid \texttt{inputs} \mid \textit{Range}^{+}\\
\textit{Range} &::= \texttt{x in}\ \textit{DExpr} \mid \texttt{x in uniques(}\textit{DExpr}\texttt{)} \mid\\
&\quad\ \texttt{x in indices(}\textit{DExpr}\texttt{)}\\
\textit{BExpr} &::= \textit{BExpr}\ \textit{bop}\ \textit{BExpr} \mid\ !\textit{BExpr} \mid\\
&\quad\ \textit{DExpr}\ \texttt{in}\ \textit{DExpr} \mid \textit{DExpr}\ \textit{rop}\ \textit{DExpr}\\
\textit{DExpr} &::= c \mid x \mid x[\textit{DExpr}] \mid |\textit{DExpr}| \mid \textit{DExpr}\ \textit{aop}\ \textit{DExpr} \mid\\
&\quad\ f(\textit{DExpr}^{+}) \mid [\textit{DExpr}^{+}]
\end{aligned}
$$

Fig. 3: AXPROF Specification Language

over items in the input, each item has equal weight, as used in average-case analysis of algorithms [32]. The accuracy specification also allows quantification over a list of items. We interpret these quantifiers as the requirement that the tests of the predicates inside the quantifiers should be correct for each item in the list. These predicates are translated to corresponding statistical tests using the code generation process we describe in Section IV-B.

The specification can contain the special variables `Input` (the input data) and `Output` (the algorithm output). Additional variables may also be declared in range expressions and `let` expressions.

Finally, the specification can contain standard boolean and arithmetic expressions. The boolean operator `in` checks whether an element is in a collection. Elements within a collection can be accessed using a key using typical array access notation. The operator $|\cdot|$ calculates the size of a collection. We allow a developer to call helper functions written in Python. These functions may implement complicated testing conditions or compute exact solutions through an oracle. Individual parameters from the algorithm configuration can be accessed directly by their name. Multiple expressions of the same type can be composed into a list.

IV. CHECKER CODE GENERATION

We derive statistical hypotheses from AXPROF's accuracy specifications, test them with common statistical tests, and calculate the number of executions or inputs necessary. We also generate code to check resource utilization specifications.

A. Background on Statistical Testing

A statistical hypothesis can be tested by observing samples of one or more random variables. A tester forms two hypotheses: a *null hypothesis* and an *alternative hypothesis*. Then they use an appropriate statistical test to calculate a p-value: the probability of obtaining a test statistic at least as extreme as the one observed, assuming the null hypothesis is true. If the p-value is too low, the null hypothesis can be rejected.

Several statistical tests are available for various use cases. These tests are either parametric (they make some assumption about the population from which the data is drawn) or non-parametric (they make no such assumptions). Parametric tests are generally more powerful at detecting statistical anomalies, while nonparametric tests can handle more types of data and small sample sizes. We use several statistical tests in AXPROF.

The binomial test [33] is an exact nonparametric test used to compare the observed probability (p_1) of an event against an expected or threshold probability (p_0). For example, to test specifications that state $p_1 \leq p_0$ we formulate a null hypothesis $H_0 : p_1 \leq p_0$ and test it against the alternative hypothesis $H_1 : p_1 > p_0$ using the binomial test. The *greater one-tailed*, *lesser one-tailed*, and *two-tailed* variants of this test respectively check if the observed probability is too high, too low, or different to the expected probability.

The *one*-sample t-test [34] is a parametric test used to compare the mean of *one* set of samples (μ) against an expected or threshold mean (μ_0). For example, to test specifications that state $\mu = \mu_0$ we formulate the null hypothesis $H_0 : \mu = \mu_0$ and test it against $H_1 : \mu \neq \mu_0$ using the t-test. This test too has one and two-tailed variants. Although the t-test requires that the sample mean is normally distributed, when the sample size is large enough, this requirement can be assumed to hold.

Another approach to perform hypothesis testing is to use sequential testing, where hypothesis testing is performed as samples are collected. Sequential Probability Ratio Test (SPRT) [35] is often proposed as a technique to select between two hypotheses with a minimum number of samples. SPRT maintains a likelihood ratio, updated after each sample, that rates two hypotheses based on the observed samples. Based on this ratio one hypothesis is accepted, or more samples are gathered until enough evidence is available to pick one.

Fisher's method [36] is a technique for combining the results of multiple statistical tests for the same null hypothesis. Each individual test produces a p-value for the hypothesis. Fisher's method can then be used to calculate a single p-value for the entire set of tests. If this combined p-value is too low, then the null hypothesis can be rejected. Otherwise the null hypothesis cannot be rejected even if some of the individual tests failed. Fisher's method assumes that the results of the individual tests are independent, which is not always true. If the tests are dependent, using Fisher's method may result in a p-value lower than the correct p-value.

B. Generating Accuracy Checker Code

Based on the accuracy specification, AXPROF needs to select what statistical test to use and how many samples are needed for the statistical test based on the required level of confidence. In addition, AXPROF needs to decide how to aggregate output data over multiple executions or inputs and when to use the aggregated data to perform the statistical tests: after every run, after multiple runs for the same input, or after runs on multiple inputs.

AXPROF supports three main types of accuracy specifications and selects the statistical test based on the type of *ASpec* expression from the specification language and the comparison operator used in its predicate.

Probability Predicates. Specifications of the form

```
Probability over Qualif [BExpr] rop DExpr
```

require testing the probability of satisfying the inner boolean expression (`BExpr`) against the probability `DExpr`.

Each element in the space defined by `Qualif` can be treated as one sample drawn from a Bernoulli distribution which is 1 if `BExpr` is satisfied and 0 otherwise. This allows us to use the binomial test to compare `DExpr` with the probability of the Bernoulli variable. Based on the probability space defined by the user in `Qualif`, we need to gather sample outputs of the program over multiple executions or inputs to calculate the fraction of elements that satisfy `BExpr`:

- If `Qualif` is `runs`, the accuracy specification defines a probability over a set of executions of the program on the same input. Therefore, AXPROF executes the program multiple times and calculates the fraction of executions that satisfy `BExpr`. At profiling time, AXPROF sets the number of executions to the number of samples required for the binomial test (Section IV-C). If the developer provides multiple inputs, then the implementation is expected to pass the test for each input separately.
- If `Qualif` is `inputs`, the accuracy specification defines a probability over the inputs of the program. In this case, for each configuration of algorithm parameters, AXPROF executes the program on multiple inputs and calculates the fraction of inputs that satisfy `BExpr`. At profiling time, AXPROF sets the number of inputs to the number of samples required for the binomial test (Section IV-C). AXPROF generates inputs using its input generators (Section V) and the implementation is executed once on each input.
- If `Qualif` is a list of items (*Range*$^+$), after each execution of the program we calculate the fraction of items in the list that satisfy `BExpr`. The test is performed separately for each execution on each input. The specification should be valid for each run of the implementation. While we cannot prove this property with full certainty, we can do so with high confidence. In particular, we formulate the null hypothesis that the program succeeds with very high probability (e.g. greater than 0.999) and use the SPRT to estimate the number of executions sufficient to check this weaker property (Section IV-C).

In all cases, the fraction of samples (executions, inputs, or items) that satisfy `BExpr` is compared against the value of `DExpr` using the binomial test. If AXPROF observes enough evidence to reject the null hypothesis, it issues a warning to the

user. Depending on the comparison operator used (`rop`), we choose one of the three variants of the binomial test (*greater one-tailed*, *lesser one-tailed*, or *two-tailed*).

Expectation Predicates. Specifications of the form

```
Expectation over Qualif [DExpr1] rop DExpr2
```

require comparing the value in expression `DExpr1` against the expected value in expression `DExpr2`. We gather samples of the value of `DExpr1` over `Qualif` and calculate their mean. For large enough sample sizes, this sample mean value is an estimate of the *real* mean value and can be considered to be drawn from a normal distribution centered around the *real* mean value. This allows us to use of the t-test for comparing the sample mean against the expected value. Similar to the probability predicates, depending on the comparison operator used (`rop`), we choose one of the three variants of the t-test.

Based on the sample space defined by the user in `Qualif`, we may have to gather sample outputs of the program over multiple executions or inputs (as in the probability predicate case). We calculate the value of `DExpr1` for each sample and take the mean. This mean is then compared against the expected mean `DExpr2` using the appropriate t-test. The process of gathering samples is similar to the process used for probability predicates, except that AXPROF uses the t-test instead of binomial test for calculating the number of samples.

Universally Quantified Predicates. Specifications of the form

```
forall Range+ : ASpec
```

require that each element in `Range+` satisfy the accuracy predicate `ASpec` (a probability or expectation specification).

We perform the statistical test required for the probability or expectation specification `ASpec` as described in the previous paragraphs for each individual element in `Range+`. The null hypothesis for each test is that the implementation satisfies `ASpec` for all elements, the alternative being that it does not satisfy `ASpec` for that element. This results in multiple p-values, one for each element in `Range+`.

Next, we combine these p-values obtained from the individual tests into a single p-value, to test if the overall specification is satisfied. AXPROF uses Fisher's method for this purpose.

C. Determining the Required Number of Samples

For statistical tests, the required number of samples depends on the test being used, desired significance level α, the statistical power $1-\beta$, and other test-specific parameters. The user can specify these parameters to control false warnings, risk of potentially missing a bug, and run time of AXPROF.

Binomial Test. The number of samples required also depends on the size of the indifference region, δ, which determines the minimum deviation from the expected probability that the test will be able to detect [37]. The minimum number of samples required to achieve the required statistical confidence is $\left(\left(z_{sig}\sqrt{p_0(1-p_0)} + z_{1-\beta}\sqrt{p_a(1-p_a)}\right)/\delta\right)^2$, where for any probability q, z_q is the critical value of the normal distribution for q. Here, z_{sig} is $z_{1-\alpha/2}$ for a two-tailed test and $z_{1-\alpha}$ for a one-tailed test.

One-sample t-test. The number of samples required also depends on the effect size $d = |\mu_0 - \mu_1|/\sigma$, the difference in the means corresponding to the null hypothesis (μ_0) and an alternative hypothesis (μ_1) divided by the standard deviation (σ) of the sample being tested. The minimum number of samples required is $(t_{sig} + t_\beta)^2/d^2$, where for any probability q, t_q is the critical value of the t-distribution for q. Here, t_{sig} is $t_{\alpha/2}$ for a two-tailed test and t_α for a one-tailed test.

SPRT. To calculate the minimum number of runs n, we use SPRT with the minimum success probability H and the maximum failure probability L, as $n \geq \frac{log(\beta)-log(\alpha)}{log(H)-log(L)}$. We ensure that each individual execution passes the test.

D. Analyzing Resource Utilization

To analyze the time and memory consumption of a computation, AXPROF employs curve fitting to build the most likely regression model and checks the quality of the fit. As the first step, AXPROF generalizes specification expressions provided by the developer to capture the hidden constants in asymptotic notations. For example, if the specification of time/memory is the expression $x+y*z$, then AXPROF will generate the general expression: $(p_0 * x + p_1) + (p_2 * y + p_3) * (p_4 * z + p_5) + p_6$. For this expression, AXPROF searches for the values of the free variables $p_{0\ldots6}$ that best fit the data using statistical curve fitting [38]. The curve fitting procedure computes the R^2 metric, which quantifies how well the model fits the observed data. Higher R^2 values indicate better fitting models. AXPROF triggers a warning if it cannot find a model with high R^2.

V. INPUT GENERATION

To generate random inputs to test the algorithms, we developed a set of flexible input generators, each of which can control different aspects of the generated data. Each input generator outputs a required number of data elements. We developed three input generators:

- **Integer/Real Generator.** This generator can be used to generate a list of integers or reals. The data can either be sampled uniformly from a range of valid values or can be drawn from a Zipf distribution with a given skew, which allows for the control of frequency of individual data items. The generator also allows for the control of internal order of data items, and the distance between individual data items.
- **Matrix Generator.** This generator can be used to generate a matrix of given dimensions with random elements. In addition to the controllable parameters from the previous generator, the sparsity of the matrix can be controlled. The sparsity can be between 0 (fully dense) to 1 (only zeros).
- **Vector Generator.** This generator can be used to generate a list of vectors with given number of dimensions. The internal order of elements and the distance between individual vectors (ℓ_2 distance) can be controlled.

Automatically Selecting Input Generator Features. Identifying what input features affect the accuracy of a program is important to selecting a input generator and deciding what inputs to test. Each generator described above has input features that can be used to control the generated inputs.

TABLE I: Summary of the Algorithms. (n is the size of the dataset in all unspecified cases)

Name	Parameters	(Informal) Accuracy Specification	Time	Memory
Locality Sensitive Hashing	k : hash functions per table l : number of hash tables	P[*Neighbor*] $= 1-(1-p^k)^l$ p depends on similarity	Insertion : $O(kl)$ Query : $O(kl+n)$	$O(ln)$
Bloom Filter	p : max. false pos. probability c : capacity	P[*False positive*] $\leq p$ If number of inserted elements $< c$	Insertion: $O(\log(1/p))$ Query: $O(\log(1/p))$	$O(n\log(1/p))$
Count-Min Sketch	ϵ : error factor δ : error probability	P[*error* $> n*\epsilon] < 1-\delta$	$O(\log(1/\delta))$	$O((1/\epsilon^2)\log(1/\delta))$
HyperLogLog	k : Number of hash values	P[*error* $\leq 1.04/\text{sqrt}(k)] >= 0.65$	$O(1)$ for fixed k	$O(k)$
Reservoir Sampling	s : reservoir size	P[*in sample*] $= \min(s/n, 1)$	$O(1)$	$O(s)$
Matrix Multiply	c : sampling rate A : m x n matrix B : n x p matrix	$P[\|\|error\|\|_F < C] > 1-\delta$ $C = \eta/c * \|\|A\|\|_F \|\|B\|\|_F$ where $\eta = 1+\sqrt{8*\log(1/\delta)}$	$O(mcnp + c(m+p))$	$O(mcnp + c(m+p))$
Chisel/blackscholes	r : reliability factor	P[*exact* $=$ *approx*] $> r$	$O(1)$	$O(1)$
Chisel/sor	r : reliability factor m x n : matrix dimensions i : iterations	P[*exact* $=$ *approx*] $> r$	$O(imn)$	$O(mn)$
Chisel/scale	s : scale factor r : reliability factor	E[*PSNR*$(d, d')] \geq -10 \cdot \log_{10}(1-r)$	$O(s^2mn)$	$O(s^2mn)$

TABLE II: Accuracy Specifications Provided to the Checker Generator

Algorithm	Accuracy Specification for AxProf
LSH	`forall i in indices(Input), q in indices(Input) :` `  Probability over runs [ [q,i] in Output ] == L1HashEqProb(Input[i],Input[q],10,k,l)`
Bloom Filter	`Probability over i in excluded(Config,Input) [ i in Output ] < p`
Count-Min Sketch	`Probability over i in uniques(Input)` `  [ (count(i,Input) - Output[i]) > epsilon*|Input| ] < 1 - delta`
HyperLogLog	`Probability over inputs [ abs(datasize-Output) < (datasize*1.04)/sqrt(2^k) ] >= 0.65`
Reservoir Sampling	`forall i in Input : Probability over runs [ i in Output ] == min(ressize/datasize,1)`
Matrix Multiply	`Probability over runs` `  [Output < (eta(delta)/samplingFactor)*(frobenius(Input[0])*frobenius(Input[1]))] > delta`
Chisel/blackscholes	`Probability over runs [ Output == oracle(Config,Input) ] > r`
Chisel/sor	`Probability over runs [ Output == oracle(Config,Input) ] > r`
Chisel/scale	`Expectation over runs [ PSNR(Input, Output) ] >= -10*log10(1-r)`

We adapt a technique from [39] to select important input features that need to be explored. We use *Maximal Information Coefficient* (MIC) [40] to identify relationships between input features and the accuracy of a program. MIC is commonly used to identify associations between a given pair of variables. For each input feature available in a input generator we perform sample runs of the program and gather the accuracy of the runs. We use this data to calculate a MIC value. If the MIC value is beyond a threshold, we accept that input feature as one that affects output.

VI. Methodology

Table I presents the summary of algorithms we analyze in this paper. We chose these algorithms to represent common randomized and approximate tasks. The table lists algorithmic parameters that can be controlled (Column 2), and the accuracy, time, and memory specifications (Columns 3, 4, and 5). Table II presents the accuracy specifications for each algorithm specified using AxProf's language.

A. Tested Algorithms

Locality Sensitive Hashing. We presented it in Section II.

Bloom Filter. Bloom Filter [8] checks if an item was present in a data stream. It starts with k different hash functions and an all-0 bit filter of length m. For each data item, it calculates the k different hash functions and sets the corresponding bits to 1. To check if an item q was in the stream, the algorithm calculates all the hashes of q and checks that each corresponding bit is 1. k and m are calculated from a specified capacity c and a maximum false positive rate p. In the specification in Table II, `excluded` calculates the set of items in the input that were not inserted into the filter.

Count-Min Sketch. Count-Min Sketch [6] counts the frequency of items in a dataset. The algorithm maintains a set of uniform hash functions whose range is divided into a set of bins. For every item in the dataset, the hash functions are calculated and counters in the mapped bins are incremented. The estimated frequency of an item is the minimum of all the counters in the corresponding bins. The accuracy can be improved by increasing the number of hash functions and bins. In the specification in Table II, `uniques` calculates the set of unique items in the input, as some items appear multiple times.

HyperLogLog. HyperLogLog [3] is an algorithm for calculating the number of distinct elements in a large dataset. For each element in a dataset, the algorithm calculates k hash values. The hash value with the maximum number of leading zeros is then used to estimate the cardinality of the dataset. The variance of the result can be reduced by using more hashes.

TABLE III: Controlled parameters

Algorithm	Parameters	Range
LSH	Hash functions per table	2, 4, 8
	Number of hash tables	2, 4, 8, 16
	Input size (performance)	1000 - 10000 step 1000
	(accuracy)	100
Bloom Filter	Max. false positive prob.	0.1, 0.01, 0.001
	Load (fraction of capacity)	0.2 - 0.8 step 0.2
	Capacity (performance)	20000 - 100000 step 20000
	(accuracy)	200 - 1000 step 200
Count-Min	ϵ : error factor	0.1, 0.01, 0.001
	δ : error probability	0.2, 0.1, 0.05
	Input size	10000 - 100000 step 10000
HyperLogLog	Number of hash function	2^8, 2^{10}, 2^{12}, 2^{14}
	Unique items in input	10000 - 100000 step 10000
Reservoir Sampling	Size of reservoir	10000 - 100000 step 10000
	Input: size	10000 - 100000 step 10000
Matrix Multiply	Size of matrices	20×20 - 200×200
	Sampling rate	0.2, 0.4, 0.8
blackscholes	load error rate	0.000048, 0.00024, 0.24
sor	Size of matrices	4×4 - 50×50
	Iterations	1, 5, 10, 20
	omega	0.1, 0.5, 0.75
scale	Input Image	5 Images
	scale factor	1, 2, 4, 8

Reservoir Sampling. Reservoir Sampling [7] uniformly samples a data stream. For a reservoir of size s, the first s elements are directly inserted into the reservoir. Afterwards, for the i^{th} element to be inserted, an integer p is chosen uniformly at random from $[1, i]$. If $p \leq s$ then the item currently in the p^{th} position in the reservoir is replaced with the new item.

Randomized Matrix Multiplication. Randomized matrix multiplication [9] methods reduce computation time and resource consumption of matrix multiplication by randomly sampling the matrices. Guarantees for accuracy are given as an upper bound on the Frobenius norm of the errors. The error can be controlled by changing the sampling rate.

Chisel Kernels. Chisel [20] is a reliability aware optimization framework for programs that run on approximate hardware. For a given reliability specification, Chisel minimizes energy consumption by utilizing approximate computations. Chisel programs run on an approximate hardware simulator that injects errors at runtime. We look at three kernels from Chisel: 1) *scale* scales an image by a specified scale factor; 2) *sor* performs Jacobi SOR for a given matrix; 3) *blackscholes* computes the price of a stock portfolio using the Black-Scholes formula. For *blackscholes* and *sor*, Chisel provides bounds on the probability that the output differs from the exact value. For *scale*, the authors provided the expression for the expected PSNR value between the exact and approximate results.

B. Algorithm Implementations

For most algorithms we selected two implementations from GitHub. We preferred implementations in Java, Python, or C/C++. We took several factors into account when selecting implementations to profile, such as the number of stars on GitHub and how active the repository is. All the selected implementations appear among the top-10 search results in GitHub for the name of the algorithm. Finally, we chose three implementations of kernels from the evaluation of Chisel [20].

C. Experimental Setup

Parameters and Their Ranges. The parameter value choices offer a trade-off between profiling time and the confidence in the algorithm's correctness. We chose parameters across their valid range. For each parameter in the time or memory specifications, we used at least 3 values across the range (for curve fitting). Table III presents the ranges in our experiments.

Statistical Tests. For statistical tests, we used a significance level $\alpha = 0.05$ and a statistical power $1 - \beta = 0.8$ when calculating the number of runs. To get the number of runs for per-run checkers, we used SPRT with an minimum acceptable probability of success of 0.999 and a maximum probability of failure of 0.99. Based on these values, AXPROF calculated that 320 runs were sufficient. For the specifications requiring the binomial test, we chose a probability deviation $\delta = 0.1$. For those that require the t-test, we chose an effect size $d = 0.2$. For both of these, AXPROF calculated (using the formulae from Section IV-C) that slightly less than 200 runs were sufficient. For identifying resource model discrepancies, we used an R^2 threshold of 0.9.

Environment. We ran experiments on a Xeon E5-1650 v4 CPU, 32 GB RAM, Ubuntu 16.04. For time profiling we used a real-time timer around the relevant functions. For memory, we used serialization in Java and Python, and the `time` Linux utility for C/C++ programs. For fitting the models we used the `scipy.optimize` module of SciPy [52].

VII. EVALUATION

This section discusses the three main research questions:

- **RQ1:** How effective is AXPROF in profiling accuracy of applications? (Section VII-A)
- **RQ2:** Can the bugs found by AXPROF be identified by performance-only algorithmic profiling? (Section VII-D)
- **RQ3:** Is AXPROF effective in identifying input features that affect program's accuracy? (Section VII-E)

A. Effectiveness of AXPROF in Profiling Accuracy

Table IV presents a summary of our findings using AXPROF. Column 1 presents the algorithm, and Column 2 the profiled implementation. Column 3 presents the results for accuracy analysis for each benchmark implementation. Columns 4 and 5 present the analysis for time and memory. In each column, a "✓" represents that AXPROF did not find any issues. WARN(X/Y) represents cases where AXPROF issued a warning in X out of Y configurations. A "*" indicates a false warning. For accuracy analysis each configuration was tested independently. For time and memory analysis, data from all configurations are part of a single regression model.

Out of the 15 implementations that we profiled, 4 implementations (Matrix Multiplication in *mscs* and all Chisel kernels) passed all tests for compliance with accuracy, time and memory specifications. AXPROF detected conditions that trigger warnings in the remaining 11 implementations. We manually analyzed the implementations that caused warnings. We found two causes for the 9 *real* warnings:

TABLE IV: Summary of the Results of Profiling

Algorithm	Implementation	Accuracy	Time		Memory
LSH	*TarsosLSH* [31]	WARN (12/12)	✓		✓
LSH	*java-LSH* [41]	WARN (4/4)	✓		N.A.
Bloom Filter	*libbf* [42]	✓	Insertion:✓	Query:WARN	✓
Bloom Filter	*BloomFilter* [43]	✓	Insertion:✓	Query:WARN	✓
Count-Min	*alabid* [44]	WARN (90/90)	✓		✓
Count-Min	*awnystrom* [45]	WARN (81/90)	✓		WARN*
HyperLogLog	*yahoo* [46]	✓	WARN		WARN
HyperLogLog	*ekzhu* [47]	WARN* (2/40)	✓		✓
Reservoir Sampling	*yahoo* [46]	✓	✓		WARN
Reservoir Sampling	*sample* [48]	✓	WARN*		✓
Matrix Multiplication	*RandMatrix* [49]	WARN (30/243)	✓		✓
Matrix Multiplication	*mscs* [50]	✓	✓		✓
blackscholes	Chisel [51]	✓	✓		✓
sor	Chisel [51]	✓	✓		✓
scale	Chisel [51]	✓	✓		✓

- Errors in implementations: four implementations used hash functions with errors that caused higher than expected accuracy loss. One implementation had a misconfigured random number generator that affected sampling (Section VII-B).
- Performance optimizations that caused unexpected execution times or memory usage (Section VII-C).

We observed false warnings (WARN*) in the time specification check for *sample* and the memory check for *awnystrom* due to noise in measurements. In HyperLogLog, when the input-set cardinality is very low and close to a predefined threshold, the estimate error is expected to be high. This led to warnings for two configurations of *ekzhu* (*yahoo* avoided the problem via a polyalgorithm [53]).

B. Errors in Implementations

LSH: *TarsosLSH*. We discuss this benchmark in Section II.

LSH: *java-LSH*. This is a MinHash-LSH implementation in Java for Jaccard similarity metric [54]. We observed that sets were being considered similar to the query set more often than expected, as shown in Figure 4. Each point represents a query-datapoint pair. The x and y coordinates of the point denote the expected and observed probability, respectively. Results before and after the bugfix are shown. The implementation used the simple hash function $h(x) = (a * x + b)\%m$. This hash function is usable only when m is a prime. However, the implementation often chose a composite value for m. We fixed the hash by setting m to a fixed, large prime. After this fix, the observed results matched the expected values.

Count-Min Sketch: *awnystrom*, *alabid*. For each data item, Count-Min Sketch calculates multiple hash functions chosen randomly from a family of hash functions. The family of hash functions used in an implementation should be *pairwise independent*. i.e. if h is a function chosen uniformly at random from the hash family, for two values x and y, $h(x)$ and $h(y)$ are uniformly distributed and pairwise independent.

AXPROF detected that the observed error rate was higher than expected for many configurations in both implementations. By manual inspection we identified that the buggy implementation uses a hash function that does not satisfy the pairwise independence property. Figure 5 presents the observed errors in *awnystrom*. The X axis shows the various input sizes, and the Y axis shows the fraction of runs that had errors beyond the specified threshold. We observe that this fraction dropped significantly in both implementations after we fixed the bugs in the hash functions.

Matrix Multiplication: *RandMatrix*. The rows and columns of the matrices to be multiplied are subsampled to reduce their size. The algorithm [9] provides an optimum sampling method that was not implemented correctly (wrong initialization of `std::discrete_distribution` in C++) in the implementation leading to wrong results.

Developer-Provided Tests. In all cases, tests written by the developers failed to catch the bugs identified through AXPROF. We observed three main reasons: 1) unit tests only partially check the algorithm functionality (*java-LSH*), 2) tests use fixed inputs that do not trigger bugs (*alabid*, *awnystrom*, *RandMatrix*), 3) bugs in the test framework itself (*TarsosLSH*).

C. Performance Optimizations

We also observed situations where warnings were issued in AXPROF due to performance optimizations in implementations that were causing unexpected behavior, which were not necessarily errors.

Reservoir Sampling. The memory usage was unexpectedly low due to the implementation incrementally allocating memory. Figure 6 plots the observed memory usage against the number of inserted elements for various reservoir sizes.

HyperLogLog: *yahoo*. AXPROF was unable to model the runtime of the algorithm against the input size due to the polyalgorithm implementation [53]. Figure 7 shows the runtime of the algorithm (Y axis) against the size of the dataset (X axis) and the best linear model AXPROF found.

Bloom Filter. Instead of checking the entire filter to search for a 0 value, the programs return when the first 0 is found. This property is not encoded in the specification.

D. Effectiveness of Accuracy Profiling

We analyzed all of the accuracy bugs we identified and confirmed that they can not be detected through regular

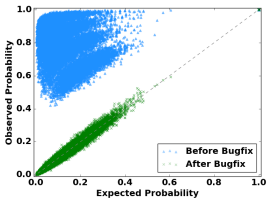


Fig. 4: *java-LSH* Accuracy

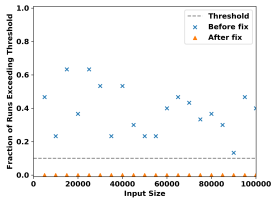


Fig. 5: *awnystrom* Accuracy

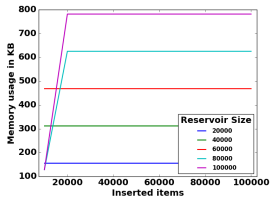


Fig. 6: RS:*yahoo* Memory

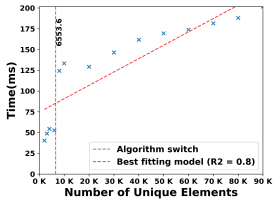


Fig. 7: HLL:*yahoo* runtime

TABLE V: Input feature impact on accuracy

Algorithm	Order	Frequency	Distance	Sparsity
Count-Min	(x, x)	(✓, ✓)	(x, x)	-
HLL	(x, x)	(x, x)	(✓, x)	-
BF	(x, x)	(✓, ✓)	(x, x)	-
Matrix Mul	(✓, ✓)	-	(✓, ✓)	(✓, ✓)
LSH	(x, x)	(x, x)	(✓, ✓)	-

profiling techniques that focus only on runtime and/or resource consumption. As can be seen from Table IV, in all situations accuracy bugs could not be detected through unexpected runtime or memory consumption that could have been identified through usual algorithmic profiling methods. The only case where accuracy warnings coincide with other warnings was due to noise in performance measurements for the memory specification check for *awnystrom*. These results show the importance of including accuracy in algorithmic profiling.

E. Effectiveness of Input Feature Selection

We studied four input features from AXPROF's input generators and their effect on accuracy of the program. Table V shows the results of the analysis. We only looked at the correct implementations of the programs. We analyzed the benchmarks manually to confirm the results of the *MIC*-based approach (Section V). Each column with the format "(automatic/manual)" has a check mark if AXPROF's automatic and our manual analyses, respectively, show that the input feature affects accuracy. For Reservoir Sampling, we were unable to derive an accuracy measurement due to the nature of the specification. For the Chisel benchmarks, the accuracy depends only on the underlying hardware, therefore input features we changed did not have any impact.

We observed one situation where the manual analysis differs from the results of the *MIC* based approach. In HyperLogLog, the *distance between individual values* is falsely identified as affecting accuracy even though it should not. We attribute this to randomized properties of the hash functions.

VIII. RELATED WORK

Algorithmic Profiling. Recently, researchers explored techniques for advanced algorithm-aware profiling and estimation. Several approaches have been proposed to model performance of programs as a function of workload [26], [27], [28]. Algorithmic profiling [28] is a framework that focuses on automating such profiling tasks by detecting algorithms and their inputs. COZ [55] is a causal profiler that estimates the effect of potential optimization of subcomputations on the performance of the whole program. Researchers proposed similar techniques for analyzing memory and recursive data structures, e.g., [56], [57]. AXPROF's accuracy analysis is complementary to these existing approaches.

Statistical debugging and profiling tools, e.g., [58], [59], [60] use statistical models of programs to predict and isolate bugs. This line of research is conceptually orthogonal to ours.

Analysis of Accuracy. Researchers have also looked at various dynamic approaches [61], [62], [63], [64], [65] to empirically analyze the impact on accuracy from transformations that change program semantics. In contrast, AXPROF uses theoretical specifications of approximate algorithms and checks for discrepancies in their implementations. MayHap [19] converts program code to a Bayesian network and uses Chernoff bounds to check probabilistic assertions over a set of executions. In contrast, AXPROF operates on a program as a black-box system, supports a richer set of predicates including inputs and items, and automatically selects the appropriate test.

Statistical Model Checking. Statistical model checking [37], [66], [67] is a general method to verify properties of black-box stochastic systems using statistical hypothesis testing. For instance, the framework of Sen et al. [66] expresses properties in Continuous Stochastic Logic (CSL) and samples the outputs from the tested system. CSL's probability predicate, like our "probability over runs" predicate, estimates the probability that the system satisfies a specified logical property. However, expressing the predicates over inputs and items would be significantly more complicated in CSL and it does not support expectation predicates or complex data structures, like lists or matrices. In addition, AXPROF automatically generates code for collecting and aggregating data, thus giving a developer an intuitive tool to simultaneously explore various aspects of program's accuracy and resource consumption.

IX. CONCLUSION

We presented AXPROF, an algorithmic profiling framework for analyzing execution time, memory consumption, and accuracy of randomized approximate programs. Our evaluation demonstrated that AXPROF helps developers to check that implementations of such algorithms conform to the algorithm specifications and can help in fixing accuracy related bugs. With its ability to analyze accuracy specifications, AXPROF opens a new dimension in algorithmic profiling.

ACKNOWLEDGEMENTS

The research presented in this paper was funded in part by NSF Grants CCF-1629431 and CCF-1703637.

REFERENCES

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *STOC*, 1998.

[2] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Communications of the ACM*, vol. 51, 2008.

[3] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA: Analysis of Algorithms*, 2007.

[4] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *European Symposium on Algorithms*, 2002.

[5] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, 1982.

[6] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the Count-Min sketch and its applications," *Journal of Algorithms*, vol. 55, 2005.

[7] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, 1985.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, 1970.

[9] P. Drineas, R. Kannan, and M. W. Mahoney, "Fast monte carlo algorithms for matrices i: Approximating matrix multiplication," *SIAM Journal on Computing*, vol. 36, 2006.

[10] N. Halko, P.-G. Martinsson, and J. A. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, 2011.

[11] J. Tropp, "An introduction to matrix concentration inequalities," *Foundations and Trends in Machine Learning*, vol. 8, 2015.

[12] P. Drineas and M. W. Mahoney, "RandNLA: randomized numerical linear algebra," *Communications of the ACM*, vol. 59, no. 6.

[13] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *ICDE*, 1996.

[14] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *EuroSys*, 2013.

[15] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica, "Knowing when you're wrong: building fast and reliable approximate query processing systems," in *SIGMOD*, 2014.

[16] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo, "Abs: a system for scalable approximate queries with accuracy guarantees," in *SIGMOD'14*.

[17] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ASPLOS*, 2015.

[18] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[19] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," *PLDI*, 2014.

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

[21] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[22] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *DATE*, 2015.

[23] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *OOPSLA*, 2015.

[24] S. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," in *SCC*, 1982.

[25] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *EuroSys*, 2017.

[26] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *ESEC/FSE*, 2007.

[27] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," *PLDI*, 2012.

[28] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," *PLDI*, 2012.

[29] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE*, 2011.

[30] R. Motwani and P. Raghavan, *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[31] "TarsosLSH `github.com/JorenSix/TarsosLSH`."

[32] A. Bogdanov, L. Trevisan *et al.*, "Average-case complexity," *Foundations and Trends in Theoretical Computer Science*, vol. 2, 2006.

[33] M. M. WagnerMenghin, *Binomial Test*, 2014.

[34] H. Cramér, *Mathematical methods of statistics*. Princeton university press, 2016.

[35] A. Wald, "Sequential tests of statistical hypotheses," *The annals of mathematical statistics*, vol. 16, 1945.

[36] R. A. Fisher, *Statistical methods for research workers*. Oliver and Boyd, 1925.

[37] H. L. Younes, M. Kwiatkowska, G. Norman, and D. Parker, "Numerical vs. statistical probabilistic model checking," *International Journal on Software Tools for Technology Transfer*, vol. 8, 2006.

[38] C. John, W. Cleveland, B. Kleiner, and P. Tukey, "Graphical methods for data analysis," *Wadsworth, Ohio*, pp. 128–129, 1983.

[39] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi, "Dealing with the unknown: Resilience to prediction errors," in *PACT*, 2015.

[40] D. Reshef, Y. Reshef, H. Finucane, S. Grossman, G. McVean, P. Turnbaugh, E. Lander, M. Mitzenmacher, and P. Sabeti, "Detecting novel associations in large data sets," *Science*, vol. 334, 2011.

[41] "java-LSH `github.com/tdebatty/java-LSH`."

[42] "libbf: Bloom filters for C++11 `mavam.github.io/libbf`."

[43] "java-bloomfilter: `github.com/MagnusS/Java-BloomFilter`."

[44] "Count-Min Sketch `github.com/alabid/countminsketch`."

[45] "CountMinSketch `github.com/AWNystrom/CountMinSketch`."

[46] "Sketches Library from Yahoo `datasketches.github.io`."

[47] "Datasketch `github.com/ekzhu/datasketch/`."

[48] "sample `github.com/alexpreynolds/sample`."

[49] "Randomized Matrix Product `github.com/NumericalMax/Randomized-Matrix-Product`."

[50] "mcsc `https://github.com/gnu-user/mcsc-6030-project`."

[51] S. Misailović, "Accuracy-aware optimization of approximate programs," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[52] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[53] E. Cohen, "All-distances sketches, revisited: Hip estimators for massive graphs analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, 2015.

[54] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, 1971.

[55] C. Curtsinger and E. D. Berger, "Coz: finding code that counts with causal profiling," in *SOSP*, 2015.

[56] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *ICSE*, 2008.

[57] E. Raman and D. I. August, "Recursive data structure profiling," in *Workshop on Memory system performance*, 2005.

[58] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.

[59] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *OOPSLA*, 2014.

[60] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *ICML'06*.

[61] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[62] M. Carbin and M. Rinard, "Automatically identifying critical input regions and code in applications," in *ISSTA*, 2010.

[63] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, 2013.

[64] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC*, 2013.

[65] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *ASPLOS*, 2015.

[66] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," in *CAV*, 2004.

[67] ——, "On statistical model checking of stochastic systems," in *CAV*, 2005.