Algorithmic Profiling for Randomized Approximate Programs

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 less memory consumption. Although they 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 automatically check whether the outputs of these programs satisfy their accuracy specifications.**

We present AxProf, an algorithmic profiling framework for analyzing approximate programs. AxProf allows the developer to specify the accuracy specification as a formula in a mathematical notation, using probability or expected value predicates. AxProf automatically synthesizes 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 for generated inputs with various relevant properties.

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 bugs in the implementations of the algorithms and created fixes, guided by AxProf. The developers already accepted two of our pull requests.

I. Introduction

Many 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], compilers and hardware [18], [19], [20], [21], [22], [23].

Despite many rigorous specifications, a software developer who needs to implement, test, and tune these randomized algorithms and systems has virtually no tool support for this effort. Existing techniques for program understanding and testing do not consider accuracy of the computation. Standard profilers are only track and build models of run time and memory consumption for individual inputs [24], [25], [26], [27] or build performance models for multiple input sizes in the case of *algorithmic profiling* [28]. Recent approximate computing techniques proposed using statistical testing to estimate accuracy [29], [19], but provide only a limited set of abstractions and statistical tests. Researchers also gave guidelines for how to rigorously apply statistical testing in software engineering, e.g., [30], but a developer needs to do all steps manually and may end up with inflexible and too conservative choice of test parameters.

There are numerous tasks that a 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 test 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 checking. Moreover, manually written checking code can have various subtle errors that prevent the discovery of errors in the algorithm implementation. A more promising alternative is profiling and testing frameworks that automate these tasks.

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.

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

- ***Probability Query:*** It specifies that the probability that the output returned by the approximate program satisfies a condition is equal to a given value or greater than or lesser than a certain threshold.
- ***Expectation Query:*** It specifies that the expected value of the output is below, above, or equal to a certain value.

These two kinds of abstractions can capture the key properties of many representative randomized and approximate computations. For instance, they are expressive enough to capture most of the accuracy specifications of randomized algorithms in Motwani and Raghawan [31]. They can also model common accuracy specifications from other domains, such as numerical linear algebra (where they extend to matrices), and approximate computing (where they represent the noise of the unreliable hardware operations).

AxProf's specification language allows the specifications to be written in a form close to their mathematical counterparts in an unambiguous manner. AxProf allows a developer to explicitly write if a probabilistic specification is over inputs, items within an input, or runs. Such different cases may be handled more precisely with different statistical tests. For each specification, AxProf automatically (1) selects a proper statistical test, (2) generates checking code that extracts the output, aggregates the values, and applies the selected test, and (3) determines the number of runs to achieve a desired level of statistical confidence. In particular, the effectiveness of AxProf's analysis can be controlled by two knobs – statistical confidence or execution time of profiling.

Testing these programs often requires specific inputs, even though the specifications apply for all inputs. To automatically produce informative inputs, we provide several input generators for scalars, vectors, and matrices, that allow for various input properties to be modified: difference in the frequency of values (e.g., uniform vs skewed), order of data (different permutations), or various forms of correlations (e.g., the next value is a linear function of the previous). We present a new dynamic analysis that infers which of these properties affect the algorithm's accuracy the most (Section V-A).

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

AxProf helped us identify and fix previously-unknown problems with 5 different implementations of these algorithms (Section VII). Our analysis shows that these problems could not have been identified with standard profilers that track only memory or runtime. The problems are related to wrong implementation of the algorithms or their key components like hash-functions. We prepared pull-requests for each of these problems. The implementation developers already accepted two pull-requests. AxProf also identified that some implementations 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 as the input size grows.

These results show that AxProf's focus on accuracy analysis opens up a new dimension in algorithmic profiling. Previous approaches for algorithmic profiling [28], [25] focused only on deriving models of performance. As such, they miss to characterize the 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[1], a system that automates many algorithmic profiling tasks and pinpoints potential algorithm specification violations.
- **Accuracy Analysis:** We present a synthesis approach that automatically generates statistical testing procedures from the high-level probabilistic specifications, which resemble the ones used in the mathematical papers that describe approximate computations. These analyses can pinpoint if the algorithm implementations significantly deviate from their mathematical specifications.
- **Evaluation:** We present the evaluation of AxProf on a set of 15 approximate benchmarks from three application domains (data analytics, numerical linear algebra, approximate computing). Our results show the effectiveness of AxProf. Our case studies show (1) how AxProf can be effectively used to find errors in randomized approximate programs and check for the correctness of the fixes and (2) how AxProf can identify polyalgorithms and optimizations of resource usage.

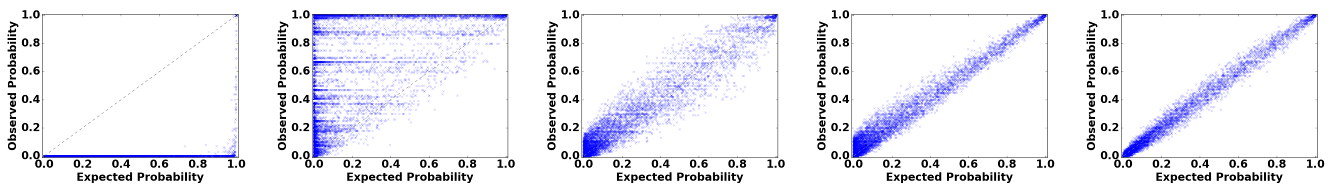
II. Example

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 using l different maps, each of which has different hash functions from the same family. LSH can also concatenate signatures from k different hash functions when mapping points to increase the probability that dissimilar points will be mapped to different bins. LSH returns a set of pairs of indices. The

[1]AxProf will be released as open-source software. Reference omitted because of double-blind evaluation.



(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 grey line.

first is the index of the query point and the second is the index of the detected neighbor.

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

In AXPROF, we specify this property as follows:

```
Input : list of (list of real)
Output : list of (list of real)
forall d_idx in indices(Input), q_idx in indices(Input) :
  Probability_{runs} [ [q_idx, d_idx] in Output ] ==
  1-(1-HashEqProb(Input[d_idx],Input[q_idx])^Cfg[K])^Cfg[L]
```

`HashEqProb` is a function that calculates $p_{d,q}$ and the parameters `K` and `L` are obtained from the configuration `Cfg`. The first two lines tell AXPROF the data type of the input and output. AXPROF uses this specification to automatically generate code to check that the property holds.

Time and Memory Specification. As each point must be hashed lk times, the insertion time of a single point is $O(lk)$. The query time is $O(lk+n)$. In AXPROF, we specify these simply as `L*K` and `L*K+DataSize` respectively. LSH only allocates space for non-empty map bins. Therefore, its memory usage is $O(nl)$. In AXPROF, we specify this as `DataSize*L`.

A. Testing the Implementation

We tested *TarsosLSH* [32], 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 generate 2 dimensional points with each coordinate in the range $[-10, 10]$. AXPROF experiments with various additional parameters such as the mean distance between points and the order of points.

B. Identifying and Fixing Bugs

The implementation included a test class 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.

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 many inaccuracies in the implementation 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 led to the result in Figure 1c. While the result in Figure 1c seems to somewhat conform with the diagonal line as expected, AXPROF's statistical tests indicated 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 algorithm conformed with the accuracy specification.

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

III. AXPROF OVERVIEW

Figure 2 presents the overview of the system. In this section, we present several key aspects of the AXPROF's analysis. In Section IV, we present statistical techniques for constructing models and checking whether they deviate from the ones provided in the specification, first for accuracy and later for time and memory consumption. We also detail how we

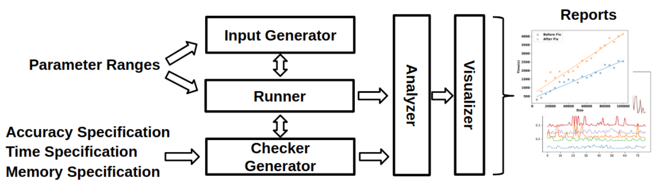


Fig. 2: Architecture of AxProf

generate code for performing these checks. Then we describe input generators in Section V.

Inputs. AxProf takes the following inputs:

- ***Implementation and Parameters:*** AxProf takes an implementation of the algorithm to test and a set of algorithm configuration parameters 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.
- ***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:

The *Checker Generator* module takes an accuracy, time, and memory usage specification from the user and automatically generates code to accumulate time, memory usage, and output data and check that the data conforms with the specifications.

The *Input Generator* module generates inputs to test the program. It can experiment with various input parameters to determine which ones affect the output of the algorithm.

The *Runner* module executes the program multiple times for various generated inputs. It returns the generated output and resource consumption statistics to AxProf.

The *Analyzer* module is the core of AxProf. It constructs models of accuracy, time, and memory consumption. Then it performs statistical tests that check whether the empirical models conform to the provided specifications, and if they do not, issues a warning to the user. AxProf's analyses provide statistical guarantees that a warning may signify a real discrepancy with high probability. The developer can set the parameters that influence the precision of the analyses.

The *Visualizer* module plots time, memory, and accuracy statistics for visual analysis by the user.

IV. Checker Synthesis

A. Specification Language

The user expresses specifications in a high-level language that resembles the mathematical specifications. Figure 3 presents its grammar. Specifications consist of the input and output types, accuracy expression, time expression, and memory consumption expression. The time and memory expressions are typical arithmetic expressions. The types of inputs and outputs can either be scalar values (integers or reals), or collections (a list, a matrix, or a map). Knowing the types helps our generators 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 and (2) expectation comparison. Both predicates explicitly define the probability space via *qualifiers*. The qualifier can be a set of items or a set of executions. Both commonly arise in the testing of random approximate computations. If the qualification is over items in the input, each item has equal weight, as used in average-case analysis of algorithms [33]. These predicates are translated to corresponding statistical tests (Section IV-C). The accuracy specification also allows quantification over the input items. We reinterpret these quantifiers as the requirement that the tests of the predicates inside the quantifiers should be correct for all or most items. We describe the code generation process in detail in Section IV-D. We use input generators (Section V) to generate representative inputs that exercise the algorithms for many representative values.

The specification can contain the special variables `Input` (the input data), `Output` (the algorithm output), and `Cfg` (the configuration being tested). 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 arithmetic 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, again using array access notation. Multiple expressions of the same type can be composed into a list.

B. Background on Statistical Testing

A statistical hypothesis can be tested by observing samples of one or more random variables. The user first forms a *null hypothesis* with the intention of rejecting it. 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 data being tested) or nonparametric (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 [34] is an exact nonparametric test used to compare the observed probability of an event against an expected probability. 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 both. The χ^2 test is also used for this purpose, however it is a parametric test that can give inaccurate results when the expected probability is very close to 0 or 1. It is also possible to simply estimate the probability of the event for any error bound ϵ and confidence interval $1 - \delta$. However the number of samples required for this method as calculated by the Chernoff bound [35] is a large overestimate, making it impractical.

The *one*-sample *t*-test [36] is a parametric test used to compare the mean of *one* set of samples against an expected mean. It too has one and two tailed variants. It returns a t-statistic and a p-value, one or both of which are used to determine if the sample mean satisfies the expectation. Although the t-test assumes that the sample data is normally distributed, when the sample size is at least 20, this requirement can be ignored.

Sometimes, it many be necessary to combine the results of multiple statistical tests for the same null hypothesis. Fisher's method [37] is a commonly used heuristic for this purpose. It takes a list of p-values of the individual tests and combines them into a single p-value that reflects the overall results. Fisher's method assumes that the results of the individual tests are independent, which is not always true. If the tests are dependent, Fisher's method may return a p-value lower than the correct p-value.

C. Choosing a Statistical Test

For randomized algorithms, our analyses provide statistical guarantees stemming from the well-known statistical tests described in the previous section.

Probability Predicate. We calculate the fraction of items (when the qualifier is a set of items) or executions (when the qualifier is the set of executions) that satisfy the inner boolean expression. Since we wish to compare this fraction against an expected probability, we use the binomial test for this comparison. Depending on the comparison operator used, we choose one of the three variants of the binomial test.

Expectation Predicate. We maintain a list of all samples of the inner expression over the set of items or executions (as required by the qualifier). We must compare the sample mean against the hypothesized mean. Therefore we use the one-sample *t*-test and again choose the variant based on the comparison operator.

Dealing with Quantifiers. Some specifications require that each item in a set satisfies an accuracy predicate. We must combine the p-values obtained from the individual tests for each item into a single p-value. We use the Fisher's method for this purpose. The same procedure is followed when there are multiple nested quantifiers.

D. Generating Accuracy Checker Code

Type of Checker. Based on the type of accuracy predicate, the framework must perform a check on every execution of the tested algorithm (*per-run checker*), or a single check that takes into account the results of multiple executions on the same input (*per-input checker*). If the predicate is a probability or expectation over a set of executions, then we must use a per-input checker. Otherwise, we use a per-run checker.

c	∈	*Consts*
x	∈	*Vars* ∪ {`Input, Output`}
f	∈	*Functions*
aop	∈	$\{+, -, *, /, \hat{}\}$
bop	∈	{&&, ||, ...}
rop	∈	{==, >, ...}
Spec	::=	*TSpec ASpec*; *DExpr*; *DExpr*
TSpec	::=	`Input :` *Type* `Output :` *Type*
Type	::=	`int` | `real` | `list of` *Type* | `matrix of` *Type* | `map of` *Type* `to` *Type*
ASpec	::=	`Probability_`{*Qualif*} [*BExpr*] *rop DExpr* | `Expectation_`{*Qualif*} [*DExpr*] *rop DExpr* | `forall` *Range* $^+$: *ASpec* | `let x =` *DExpr* `in` *ASpec*
Qualif	::=	`Runs` | *Range* $^+$
Range	::=	`x in` *DExpr* | `x in unique(`*DExpr*`)` | `x in indices(`*DExpr*`)`
BExpr	::=	`true` | `false` | *BExpr bop BExpr* | `!`*BExpr* | *DExpr* `in` *DExpr* | *DExpr rop DExpr*
DExpr	::=	*c* | *x* | *x*[*DExpr*] | |*DExpr*| | *DExpr aop DExpr* | *f*(*DExpr*$^+$) | `Cfg`[*x*] | [*DExpr* $^+$]

Fig. 3: AxProf Specification Language

Data Aggregation. Along with the checker function, we also create aggregator functions. These aggregate time and space usage data obtained from the runner function using a simple average reduction. For per-input checkers, we also aggregate output data. The reduction function for output data depends on the type of the Output data. For example, if the output is a real number, the aggregate is a list of reals.

Operator Overloading. Operations such as addition and multiplication are well defined for many types of data, with different meanings depending on the type of data. For integers, the + symbol represents simple addition. For lists, the same symbol represents list concatenation. We do a simple type inference pass to determine the types of all data expressions in the specification. Then we automatically generate code for the correct operation based on the data type and the operator.

E. Determining the Required Number of Runs

For per-run checkers, the implementation must satisfy the accuracy specification for each run. We perform multiple runs and trigger a warning if even one run fails. We calculate the number of test runs that need to successfully pass the specification using Sequential Probability Ratio Test (SPRT) [38]. The goal of SPRT is to select between two hypotheses with a minimum number of samples. We assume the hypothesis that the program succeeds with a probability of 1. We then use SPRT to calculate the minimum number of runs required (n) based on a minimum acceptable probability of success (H), a maximum probability of failure (L), and a significance level α and power $1-\beta$ as $n \geq \frac{log(\beta)-log(\alpha)}{log(H)-log(L)}$.

For per-input checkers, the number of runs required depends on the test being used, desired significance level α, the statistical power $1-\beta$, and other test-specific parameters.

When using the binomial test, the number of runs required also depends on $\delta = |p_0 - p_a|$, the minimum difference in the probabilities corresponding to the null hypothesis (p_0) and an alternative hypothesis (p_a). The minimum number of runs required 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 z_{sig} is $z_{1-\alpha/2}$ for a two-tailed test and $z_{1-\alpha}$ for a one-tailed test. z_q is the critical value of the normal distribution corresponding to the probability q.

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

F. Profiling with Time Constraints

We also allow developers to specify a time limit for profiling. We perform a set of bootstrapping runs, one per configuration. We add the runtimes for each configuration to calculate the total time it takes to perform a single run for all configurations. Using this value we calculate the maximum amount of runs that can be performed in the allocated time limit. If it is less than the amount required to achieve the default significance level α, we run the program for as many runs possible, and calculate a significance value α we can report to the user with the equations described in section IV-E.

By increasing the significance level we are increasing the probability of mistakenly reporting a correct implementation as incorrect. Therefore, shorter time constraints will increase the false positive rate. We continue keeping track of runtime for each configuration and repeat the timing calculations periodically to better estimate the number of runs that can be performed in the remaining time.

G. Analyzing Resource Utilization

To analyze the time and memory consumption of the computation AxProf employs multiple polynomial fitting to build the most likely models and checks the quality of the fit. As the first step, AxProf generalizes the specification expression provided by the developer to capture the hidden constants in asymptotic notations. For example, if the specification of time/memory is the expression is $x+y*z$, then AxProf will generate the general expression: $p_0x + p_1 + (p_2y + p_3)(p_4z + p_5)$. For this expression, AxProf searches for the values of the free variables $p_{0\ldots5}$ that best fit the data using statistical curve fitting [39].

The curve fitting procedure computes the R^2 metric [40], 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 an R^2 value above a certain threshold.

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-Float Generator. This generator can be used to generate a list of integers or floats. 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.

A. 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.

We adapt a technique from [41] to select important input features that need to be explored. We use *Maximal information coefficient* (MIC) [42] 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 of given type we perform sample runs of the program, gather the accuracy of the runs. We then use that data to calculate a MIC value. If the MIC value is beyond a threshold (0.5), we accept that input feature as one that affects output, and add it to the configurations to test.

VI. Methodology

A. Algorithm Specifications

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).

Locality Sensitive Hashing. This algorithm is described in detail 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

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

Name	Parameters	(Informal) Accuracy	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: $O(log(1/p))$	$O(nlog(1/p))$
Countmin Sketch	ϵ : error factor δ : error probability	$P[Error > n * \epsilon] < 1 - \delta$	$O(ln(1/\delta))$	$O((1/\epsilon^2)log(1/\delta))$
HyperLogLog	k : Number of hash values	$P[error \leq 1.04/sqrt(k)] >= 0.65$	$O(1)$ 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	$\mathbb{E}[PSNR(d, d')] \geq -10.log_{10}(1 - r)$	$O(r^2mn)$	$O(r^2mn)$

TABLE II: Accuracy Specifications Provided to the Checker Generator

Algorithm	Accuracy Specification for AxProf

LSH	`forall di in indices(Input), qi in indices(Input) :`
	`  Probability_{runs} [[qi, di] in Output] == 1-(1-HashEqProb(Input[di],Input[qi])^Cfg[K])^Cfg[L]`

Bloom Filter	`Probability_{i in excluded(Cfg[load],Input)} [ i in Output ] <= Cfg[p]`
Countmin Sketch	`Probability_{i in uniques(Input)} [ (count(i,Input)-Output[i]) > Cfg[epsilon]*|Input| ] < 1-Cfg[delta]`
HyperLogLog	`Probability_{Runs} [ abs(Output-|Input|)/|Input| <= 1.04/sqrt(Cfg[k]) ] >= 0.65`
Reservoir Sampling	`forall i in Input : Probability_{Runs} [ i in Output ] == min(Cfg[ressize]/|Input|,1)`

Matrix Multiply	`let eta = 1+sqrt(8*log(1/Cfg[delta])) in`
	`  let C = eta/Cfg[c]*frobenius(Input[0])*frobenius(Input[1]) in`
	`    Probability_{Runs} [ frobenius(Input[0]*Input[1]-Output) < C ] > 1-Cfg[delta]`

Chisel/blackscholes	`Probability_{Runs} [ Output == oracle(Cfg,Input) ] > Cfg[r]`
Chisel/sor	`Probability_{Runs} [ Output == oracle(Cfg,Input) ] > Cfg[r]`
Chisel/scale	`Expectation_{Runs} [ PSNR(oracle(Cfg,Input),Output) ] >= -10*log10(1-Cfg[r])`

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.

Countmin Sketch. This algorithm [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 frequency of a item is then calculated as the minimum value of all the counters in corresponding bins. The accuracy can be improved by increasing the number of hash functions and bins.

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 hash functions (higher k).

Reservoir Sampling. Reservoir Sampling[7] creates a uniform sample of a data stream. For a reservoir of size s, the first s elements are directly inserted into the reservoir. After that, 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 a 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 a 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* kernels, Chisel provides bounds on the probability that the output differs from the exact value. For *scale*, the authors have provided a calculation of the expected value for the PSNR between the exact and approximate results.

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)	10000 - 100000 step 10000
	(accuracy)	100
Bloom Filter	Max. false positive prob.	0.1, 0.01, 0.001
	Capacity of filter	20000 - 100000 step 20000
	Load (fraction of capacity)	0.2 - 0.8 step 0.2
Countmin	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	Size of reservoir	10000 - 100000 step 10000
	Input: size	10000 - 100000 step 10000
Matrix Multiply	Size of matrices	50 x 50 - 200 x 200
	Sampling rate	0.2, 0.4, 0.8
	Sparsity	0.2, 0.4, 0.8
blackscholes	load error rate	0.000024, 0.000048, 0.00096
sor	Size of matrices	50 x 50 - 200 x 200
	Iterations	5, 10, 20
	omega	0.1, 0.5, 0.75
scale	Input Image	5 Images
	scale factor	2, 4, 8

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 search results in GitHub for the name of the algorithm. Finally, we chose three implementations of kernels from the evaluation of Chisel [20].

C. Parameters and Their Ranges

The parameter value choices offer a trade-off between profiling time and the confidence in the correctness of the algorithm. We chose algorithmic parameters across their valid range. For each parameter that appears in the time or memory specifications, we used at least 3 values across the range to enable curve fitting. Column 4 of Table III presents the ranges we profiled.

D. Statistical Tests and Environment

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, we determined that 320 runs were sufficient. For per-input checkers that used the binomial test, we chose a probability deviation $\delta = 0.1$. For those that used the t-test, we chose an effect size $d = 0.2$. For both types of per-input checkers, we determined that slightly less than 200 runs were sufficient, so we decided to do 200 runs. For identifying resource model discrepancies, we used an R^2 threshold of 0.9.

Our testing setup used a Intel Xeon E5-1650 v4 CPU, 32 GB RAM, Ubuntu 16.04. For time profiling we used a real-time timer around the relevant functions in the implementation. 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 [54].

VII. Evaluation

This section discusses the three main research questions:

- **RQ1:** How effective is AxProf in profiling accuracy of applications?
- **RQ2:** Can AxProf identify input features that affect accuracy of programs?
- **RQ3:** Can algorithmic profiling assist in detecting accuracy bugs?

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. Column 4 and Column 5 present the analysis for time and memory. In each column, a ✓ represents that AxProf did not find any issues during profiling. WARN(X/Y) represents situations where AxProf issued a warning in X out of Y tests. For accuracy tests each configuration was tested independently. For time and memory specifications, we performed model fitting as described in Subsection IV-G.

Out of the 15 implementations that we profiled, 5 implementations (HyperLogLog in *ekzhu*, 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 10 implementations. We manually analyzed the implementations that caused warnings. We found two causes for the 9 *real* warnings,

- Errors in implementations (5): 4 implementations used hash functions with errors that caused higher than expected accuracy loss. One implementation had a misconfigured random number generator that effected sampling.
- Performance optimizations that caused unexpected runtimes or memory usage

These implementations are described in detail in Section VII-B and VII-D. We provided bug fixes for all errors and 2 fixes have already been accepted by the developers.

We observed a false warning in the memory specifications check for a implementation of Reservoir Sampling (*sample*) due to noise in memory usage measurements.

The results show that a developer can profile applications with varying specifications with little effort using AxProf. Once a developer translates a algorithm specification to our language, we automatically generate the test harness required to run the program and check its conformance to a specification. We only require the user to provide the functionality to perform the simple tasks of running the implementation and formatting the output. Once a checker is generated for an algorithm it can be used for other implementations of the same algorithm, with little to no modification, further reducing the effort required from a developer.

In all cases tests written by the developers failed to catch the bugs identified through AxProf. We observed three main reasons for such situations: 1) Unit tests that test parts of

TABLE IV: Summary of the Results of Profiling

Algorithm	Implementation	Accuracy	Time	Memory
LSH	*TarsosLSH* [32]	WARN (12/12)	✓	✓
LSH	*java-LSH* [43]	WARN (4/4)	✓	N.A.
Bloom Filter	*libbf* [44]	✓	Insertion:✓ Query:WARN (1/1)	✓
Bloom Filter	*BloomFilter* [45]	✓	Insertion:✓ Query:WARN (1/1)	✓
Countmin	*alabid* [46]	WARN (90/90)	✓	✓
Countmin	*awnystrom* [47]	WARN (81/90)	✓	✓
HyperLogLog	*yahoo* [48]	✓	WARN (4/4)	WARN (4/10)
HyperLogLog	*ekzhu* [49]	✓	✓	✓
Reservoir Sampling	*yahoo* [48]	✓	✓	WARN (1/1)
Reservoir Sampling	*sample* [50]	✓	✓	WARN (1/1)
Matrix Multiplication	*RandMatrix* [51]	WARN (30/243)	✓	✓
Matrix Multiplication	*mscs* [52]	✓	✓	✓
blackscholes	Chisel [53]	✓	✓	✓
SOR	Chisel [53]	✓	✓	✓
scale	Chisel [53]	✓	✓	✓

the algorithm, but not the whole implementation (*java-LSH*) 2) Tests that use a fixed set of inputs that do not trigger implementation bugs (*alabid*, *awnystrom*, *RandMatrix*) 3) Bugs in the test framework itself (*TarsosLSH*).

B. Errors in Implementations

LSH: *TarsosLSH*. We discussed the errors found in this hash function, and the fixes in detail in Section II

LSH: *java-LSH*. This is a MinHash-LSH implementation in Java for Jaccard similarity metric [55]. 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.

Countmin Sketch: *awnystrom*, *alabid* For each data item countmin sketch calculates multiple hash functions chosen randomly from a family of hash functions. The family of hash functions used in the implementation is *pairwise independent*. i.e. if h is a function chosen uniformly at random from the hash family, the random variables $h(x)$ 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 bugged implementations of hash functions, that does not satisfy the pairwise independence property was being used. Figure 5 presents the errors we observed in *awnystrom* due to this. The X axis show the various input sizes, and the Y axis show the observed 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 a optimum sampling method that was not implemented correctly (wrong initialization of `std::discrete_distribution` in c++) in the implementation leading to wrong results.

TABLE V: Input feature impact on accuracy

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

C. Effectiveness of Input Feature Selection

We looked at four input features from our input generators and their effect on accuracy of the program. The results of the analysis are presented in Table V. We only look at correct implementations of the programs. We performed manual analysis to confirm the results of the *MIC* based approach described in Section V-A. Each column is of the format (Does automatic analysis identify the input feature as having impact on accuracy / does manual analysis show that the input feature affects accuracy). For reservoir sampling, we were unable to derive a accuracy measurement due to the nature of the specification. For the Chisel benchmarks, the accuracy depends only on errors in 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.

D. Performance Optimizations

We analyzed all of the accuracy bugs we identified and confirmed that they can not be detected through regular profiling techniques that focus only on runtime and/or resource consumption. This shows the importance of including accuracy in algorithmic profiling settings.

Reservoir Sampling: The memory usage was unexpectedly low due to the implementation incrementally allocating mem-

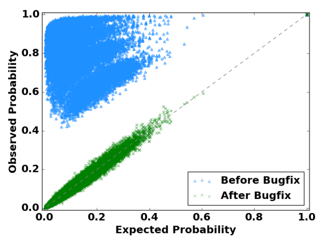


Fig. 4: *java-LSH* Accuracy

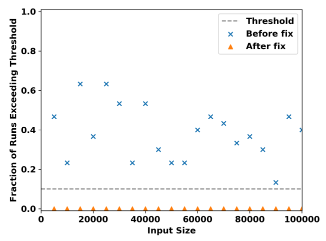


Fig. 5: *awnystrom* Accuracy

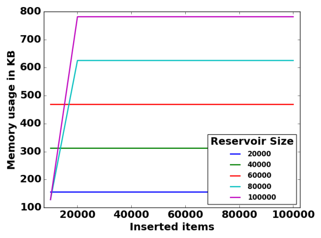


Fig. 6: RS:*yahoo* Memory

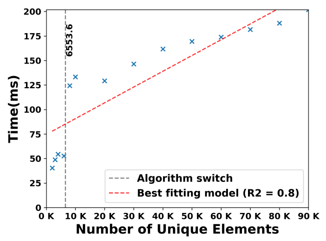


Fig. 7: HLL:*yahoo* runtime

ory. 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 implementation using a hybrid technique [56]. 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 implementations report an item as not found when the first 0 is found resulting in less runtime. This factor is not taken into account in the specification.

VIII. Threats to Validity

Internal. Our AxProf implementation may contain bugs, some issues may have been miscategorized during our evaluation (e.g., due to statistical test imprecision), and we may have made wrong conclusions about some of the potentially fault-revealing programs. To mitigate the risk of bugs in the implementation, multiple co-authors of this paper conducted a code-review. Similarly, the existing issues in the case studies were inspected by at least two co-authors, to ensure that they were indeed bugs.

External. AxProf may not generalize to all randomized algorithms. There are three aspects of our design of AxProf, as well as our experimental design which help to mitigate this risk. First, we ensured that all benchmarks have high popularity on GitHub and have been used by a variety of users. Second, the front end of AxProf is general and can accept programs from any language. Third, for performance measurements we use clock time, not simple counters.

Construct. AxProf is designed to catch bugs that significantly deviate from the expected specifications. It may not catch all bugs due to 1.) statistical tests have a pre-specified probability of failure. 2.) the errors that only slightly change the results may require many executions to capture.

IX. Related Work

Standard profilers provide the developer with the information on how a system spends its resources as it executes the programs [57], [58], [24], [59], [60], [61], [25]. They only provide information about a particular execution on an individual input and do not analyze accuracy of the computation.

Algorithmic Profiling. Recently, researchers explored techniques for advanced algorithm-aware profiling and estimation. Several approaches [26], [27] have been proposed to model performance of programs as a function of workload. Algorithmic profiling [28] is a framework that focuses on automating such profiling tasks by automatically detecting algorithms and their inputs. COZ [62] 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., [63], [64]. In contrast to these profilers that focus only on time or memory, our work focuses on multiple aspects of approximate algorithms. These algorithms have richer specifications of accuracy that are outside of the scope of the existing algorithmic profilers.

Statistical debugging and profiling tools [65], [66], [67], [68] use statistical models of programs to predict and isolate bugs. This area of work is conceptually orthogonal to ours.

Cost Models and Analyses. Researchers have looked at dynamic [69], [70], [71], [72], static [73], [74], [75], and adaptive [76] techniques for constructing cost models and analyzing resource consumption in programs. These techniques can be used to augment timing/resource models in AxProf.

Analysis of Accuracy. Researchers have also looked at various dynamic analyses [77], [78], [79] that apply program transformations that change semantics and analyzed their effect on accuracy. In contrast, AxProf uses theoretical specifications of algorithms and approximations and aims to find discrepancies in implementations. MayHap [19] proposes an analysis framework that uses statistical testing (based on Chernoff bound) to check probabilistic assertions. For checking the probabilistic specifications of programs, developers would need to perform all manual steps (that AxProf now generates) except selection of the number of runs of the statistical test. Uncertain<T> [80] uses statistical tests during program execution to determine the level of sampling for data from sensors.

X. Conclusion

We presented AxProf, an algorithmic profiling framework for analyzing execution time, memory consumption and accuracy of randomized approximate programs. We showed that AxProf helps developers to easily profile implementations of such algorithms and check if they conform to the algorithm specifications. Using AxProf, we discovered 5 bugs in implementations of such algorithms and created fixes. The developers already accepted two of our pull requests.

References

[1] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98. New York, NY, USA: ACM, 1998, pp. 604–613. [Online]. Available: http://doi.acm.org/10.1145/276698.276876

[2] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *COMMUNICATIONS OF THE ACM*, vol. 51, no. 1, p. 117, 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*. Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[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*. Springer, 2002, pp. 348–360.

[5] J. Misra and D. Gries, "Finding repeated elements," *Science of computer programming*, vol. 2, no. 2, pp. 143–152, 1982.

[6] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[7] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, Mar. 1985. [Online]. Available: http://doi.acm.org/10.1145/3147.3165

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[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, no. 1, pp. 132–157, 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, no. 2, pp. 217–288, 2011.

[11] J. A. Tropp *et al.*, "An introduction to matrix concentration inequalities," *Foundations and Trends® in Machine Learning*, vol. 8, no. 1-2, pp. 1–230, 2015.

[12] P. Drineas and M. W. Mahoney, "Randnla: randomized numerical linear algebra," *Communications of the ACM*, vol. 59, no. 6, pp. 80–90, 2016.

[13] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*. IEEE, 1996, pp. 536–545.

[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 *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 29–42.

[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 *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 481–492.

[16] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo, "Abs: a system for scalable approximate queries with accuracy guarantees," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1067–1070.

[17] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approxhadoop: Bringing approximations to mapreduce frameworks," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 383–397.

[18] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 33–52.

[19] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 112–122, 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*, vol. 49, no. 10. ACM, 2014, pp. 309–328.

[21] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, 2014.

[22] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, "Quality configurable reduce-and-rank for energy efficient approximate computing," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 2015, pp. 665–670.

[23] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 470–487, 2015.

[24] S. Graham, P. Kessler, and M. Mckusick, "Gprof: A call graph execution profiler," in *SCC '82*.

[25] J. Huang, B. Mozafari, and T. F. Wenisch, "Statistical analysis of latency through semantic profiling," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 64–79.

[26] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 395–404.

[27] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," *SIGPLAN Not.*, vol. 47, no. 6, pp. 89–98, Jun. 2012. [Online]. Available: http://doi.acm.org/10.1145/2345156.2254076

[28] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 67–76, 2012.

[29] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, p. 88, 2013.

[30] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.

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

[32] "TarsosLSH - Locality Sensitive Hashing (LSH) in Java `github.com/JorenSix/TarsosLSH`."

[33] A. Bogdanov, L. Trevisan *et al.*, "Average-case complexity," *Foundations and Trends® in Theoretical Computer Science*, vol. 2, no. 1, pp. 1–106, 2006.

[34] M. M. WagnerMenghin, *Binomial Test*. American Cancer Society, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat06340

[35] T. Hagerup and C. Rüb, "A guided tour of chernoff bounds," *Information processing letters*, vol. 33, no. 6, pp. 305–308, 1990.

[36] H. Cramér, "Mathematical methods of statistics (princeton, 1946)," *Google Scholar*, p. 212, 1950.

[37] R. A. Fisher, *Statistical methods for research workers*.

[38] A. Wald, "Sequential tests of statistical hypotheses," *The annals of mathematical statistics*, vol. 16, no. 2, pp. 117–186, 1945.

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

[40] A. C. Cameron and F. A. Windmeijer, "An r-squared measure of goodness of fit for some common nonlinear regression models," *Journal of Econometrics*, vol. 77, no. 2, pp. 329–342, 1997.

[41] S. Mitra, G. Bronevetsky, S. Javagal, and S. Bagchi, "Dealing with the unknown: Resilience to prediction errors," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 331–342.

[42] D. N. Reshef, Y. A. Reshef, H. K. Finucane, S. R. Grossman, G. McVean, P. J. Turnbaugh, E. S. Lander, M. Mitzenmacher, and P. C. Sabeti, "Detecting novel associations in large data sets," *science*, vol. 334, no. 6062, pp. 1518–1524, 2011.

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

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

[45] "java-bloomfilter: A stand-alone Bloom filter implementation written in Java `github.com/MagnusS/Java-BloomFilter`."

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

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

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

[49] "datasketch: Big Data Looks Small `github.com/ekzhu/datasketch/`."

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

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

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

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

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

[55] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, no. 5323, p. 34, 1971.

[56] E. Cohen, "All-distances sketches, revisited: Hip estimators for massive graphs analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2320–2334, 2015.

[57] "VTune Performance Analyser, Intel Corp."

[58] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, 1997.

[59] G. Pennington and R. Watson, "jProf – a JVMPI based profiler," 2000.

[60] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 74–89, 2003.

[61] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems."

[62] C. Curtsinger and E. D. Berger, "Coz: finding code that counts with causal profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 184–197.

[63] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 151–160.

[64] E. Raman and D. I. August, "Recursive data structure profiling," in *Proceedings of the 2005 workshop on Memory system performance*. ACM, 2005, pp. 5–14.

[65] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 34–44.

[66] L. Song and S. Lu, "Statistical debugging for real-world performance problems," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 561–578.

[67] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 104–119.

[68] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 1105–1112.

[69] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 301–311.

[70] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 145–155.

[71] P. Huang, X. Ma, D. Shen, and Y. Zhou, "Performance regression testing target prioritization via performance risk analysis," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 60–71.

[72] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating performance bottleneck detection using search-based application profiling," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 270–281.

[73] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *ACM Sigplan Notices*, vol. 44, no. 1. ACM, 2009, pp. 127–139.

[74] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 357–370.

[75] O. L. Olivo, "Automatic static analysis of software performance," Ph.D. dissertation, 2016.

[76] A. Das and J. Hoffmann, "Ml for ml: Learning cost semantics by experiment," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 190–207.

[77] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 25–34.

[78] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 27.

[79] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman, "Monitoring and debugging the quality of results in approximate programs," in *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 399–411.

[80] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<t>: A first-order type for uncertain data," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 51–66, 2014.