2023 IEEE International Conference on Edge Computing and Communications (EDGE) | 979-8-3503-0483-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/EDGE60047.2023.00031

Probabilistic Error Reasoning on IoT Edge Devices

Charles Qing Cao
Department of Electrical Engineering and Computer Science
University of Tennessee
Email: cao@utk.edu

Yunhe Feng
Department of Computer Science and Engineering
University of North Texas
Email: Yunhe.Feng@unt.edu

***Abstract*—Existing IoT applications are increasingly using sensors to collect real-world measurements to make decisions. Such measurements are inherently limited by the accuracy of ADC devices, hence, introduce noise and errors. However, application developers often choose scalar data to represent sensor readings without regard to the errors associated with such data. This gives the illusion that the measurements are error-free, leading to error accumulation and false positive results. In this paper, we present a new type of programming abstraction for modeling errors and performing inference tasks in measurements of the physical world on resource-constrained IoT devices, which we call approximation variables (approxes). Using approxes does not require any changes to the programming language itself. Instead, it is designed as a suite of library functions that can be integrated directly into existing programming practices. We demonstrate how to use it in C programs. This framework makes decisions about the distributions of parameter values and inherently supports sampling and hypothesis testing to evaluate the accuracy of computational results. We compare its use to traditional programming practices and show how the library can be used to reveal uncertainty to the user, so that it can handle errors, reduce false positive results, and lead to better decision-making. These benefits make approxes a compelling and promising solution for programming with noisy sensor measurements for modern IoT applications.**

I. Introduction

Sensors in modern IoT devices hide a multitude of implementation details in their easy-to-understand interfaces. For example, when an ADC sensor [1] performs a sampling operation, it almost always involves some error, i.e., the difference between the estimated reading and the actual ground truth [2], [3]. Without careful management of the possible consequences of such errors, the interpretation of results can become more complex, and, in extreme cases, lead to compounding errors. Measurements from GPS sensors, for example, are typically maximum likelihood estimates, which are often accompanied by a pair of error estimations in both the horizontal and vertical directions [4]–[6]. Unfortunately, due to the inherent complexity of dealing with error distributions, IoT programs often only use the maximum likelihood estimates, usually scalar values, as inputs to the program's computational process. On the other hand, in statistics, random variables and probability distribution functions have been widely studied and used to model the uncertainty caused by probabilistic errors [7]. For example, a random variable may be used to represent an unbiased coin flip outcome, whose value has a 50% probability of being heads or tails. When multiple random variables are involved in a calculation, statistical methods can be used to derive the joint distribution density of the resulting random variables. Given the need to manage measurement errors, it is clear that modern IoT programs should offer support for random variables in the programming language level or in the application development level. This goal is not supported by current programming practices, as existing programming languages typically use scalar types (floats, integers, and booleans) to represent the best estimate values of the measurement results, leaving the programmer with the problem of reasoning about uncertainty due to errors. Hence, programmers often use heuristics to model uncertainties, such as reporting only their maximum possible error ranges [8], [9]. Since this task of finding exact distributions is generally too complex [10], it is not usually implemented on resource-constrained IoT platforms. As a motivating example, we examined several open source reference implementations of GPS chips. These programs, written in C, typically read GPS location data and accuracy parameters. However, in application notes, these latitude and longitude outputs as the best estimates as there is no documentation on how to use the accuracy readings. For example, the C code for the SAM-M8Q GPS receiver is shown in Figure 1, which provides best estimates as well as error estimates. Note that the hAcc and vAcc values, which represent horizontal and vertical accuracy estimates, are returned only when the verbose mode is turned on, and the documentation only provides one line of text on the use: a horizontal accuracy estimate is recorded to give an indication of fix quality [11]. Programmers are not able to directly reason about the distribution of GPS errors.

```
422         /* parse messages until  the LOG array is full, or the FIFO is empty */
423         while (i < LOG_SIZE && offset < GPS_FIFOSIZE) {
424                 /* find the first UBX message and update the position */
425                 offset += alignUBXmessage(&ubx_msg_p, &GPS_FIFO[offset], GPS_FIFOSIZE - offset);
426
427                 /* extract the data from the message according to the message type */
428                 if (ubx_msg_p != NULL) {
429                         switch (ubx_msg_p->hdr.msgId) {
430                                 case UBXMsgIdNAV_PVT :
431                                         log[i].position.fixOk   = (bool)ubx_msg_p->payload.NAV_PVT.flags.gnssFixOk;
432                                         log[i].position.lat             = ubx_msg_p->payload.NAV_PVT.lat;
433                                         log[i].position.lon             = ubx_msg_p->payload.NAV_PVT.lon;
434                     #if (GPS_VERBOSE_LOG == 1)
435                     log[i].position.fixType = ubx_msg_p->payload.NAV_PVT.fixType;
436                     log[i].position.hAcc    = ubx_msg_p->payload.NAV_PVT.hAcc;
437                     log[i].position.vAcc    = ubx_msg_p->payload.NAV_PVT.vAcc;
438                     #endif
```

Fig. 1: Example of GPS sample code. Note this sample provides the horizontal and vertical error estimates as additional information to the best estimate of location.

Consequently, while these simplified approaches are intu-

2767-9918/23/$31.00 ©2023 IEEE
DOI 10.1109/EDGE60047.2023.00031

t

Fig. 2: Inconsistencies of two GPS trackers in their reports. Both trackers are of the same model from the same company.

itive, they create a dilemma in that there is no adequate estimation of the maximum errors in the final results, much less about the distribution of errors under general conditions. When used for conditional tests, such errors will result in false positive and false negative reports, which trigger potentially incorrect conditional actions. To complicate matters further, we have no estimates of the frequency of these errors, so we do not know how much the calculated results differ from the ground truth. An example of this dilemma can be found in Figure 2, which shows an experiment where we measure the daily behavior of a group of dairy cows in a smart farm project. To get accurate estimates, we placed two GPS trackers on each cow's collar. As seen from this graph, the two trackers give quite different results, shown in the green and yellow dots, for a short period of time. Should we trust one of them more than the other? Or should we take the average of the reports? Such ad-hoc solutions yield no insights on how reliable the final results will be.

In general, this error estimation problem is not limited to GPS data. Other examples include acoustic sensors for broken glass detection, which rely on comparing the received acoustic signal with a previously trained model, or real-time inferences, which require the devices to estimate probability of certain events based on posterior evidence as measured by sensor [12]. We observe that in most scenarios, measurements are probabilistic in nature. While heuristic approaches use predefined thresholds to filter the results and decide whether an event should be considered positive or negative, probabilistic inference models are more accurate, flexible, and responsive to different application scenarios. Therefore, to model the distribution of errors, a new programming framework must be developed.

Contributions: In this paper, we propose a programming framework for random variables, called approximation variables, or approxes, and make it available for programming IoT devices. We implemented this library in the C programming language, which has been widely used on IoT devices. This abstract programming structure supports arbitrary probability distributions on limited computational resources by combining sampling techniques with sequential statistical analysis for decision-making, and forward/backward inference for Bayesian graph models. Its syntax and semantics are simple enough to be used by non-experts. Therefore, this programming library significantly reduces the effort and difficulty of integrating random variables into IoT programs. We demonstrate how this framework can estimate the probability distribution of errors, and how it is compiled as executable code for runtime support. Moreover, it can support multiple types of operations on random variables and propagate probability distributions, so that it perform inference of probabilities based on evidence. To this end, we introduce a Bayesian network semantics for conditional expressions, and show that it can accurately model noisy physical phenomena that cannot be easily captured by conventional scalar variables.

We achieve the aforementioned goals by creating a very compact Bayesian network that represents a computation of the distributions, and then performing inferences over the conditional expressions. When inferences alone are not sufficient to provide accurate estimates on probability based on evidence, we use empirical samplings to provide approximate answers. The accuracy and overhead of samplings can be changed by tuning runtime parameters through program analysis or dynamically adjusting them at runtime. By using hypothesis testing for specific conditions, we have both guaranteed accuracy bounds and satisfactory performance on energy efficiency on embedded devices. With this framework, we can easily deploy learning tasks when random variables are needed. We present case studies as examples on the usefulness of the programming framework. First, we show how approxes can be used to reduce random noise in digital sensors to obtain better sensor aggregation. Second, in a smart farm deployment of IoT devices, we show how we use the approxes to improve the accuracy of location calculations from GPS trackers.

The remainder of the paper is organized as follows. We first describe the background in Section II. We then present our design in Section III. We describe the sequence sampling analysis in Section IV. The implementation is discussed in Section V, and the performance evaluation is given in Section VI. We then survey related works in Section VII. Finally the conclusions are provided in Section VIII.

II. Motivating Case Study

A. Case study

We first describe the IoT case study based on the smart farm deployment we introduced in Figure 2. In this study, we use two GPS trackers on the same cow to infer the true locations of cows. Observe that the errors would have been quite large if we had only used the best estimates. However, dealing with the possible distribution of GPS measurements

requires additional knowledge, as the distribution of errors is often irregular. For example, relevant studies have shown that such errors can be best modeled with Pareto or Rayleigh distribution, depending on the environment conditions [13]. However, such a methodology introduces problems, such as a tracker with a constant velocity may experience calculated speeds that are higher and lower, or, the tracked subject may run through walls, as illustrated for example in the yellow trace in Figure 2. We next demonstrate how to use approxes to deal with errors in such cases. We observe that GPS devices actually provide estimated error radius (a confidence interval for the position).

B. Compounding Errors

Once measurements are represented as approxes, we need to support operations on errors as uncertainty will propagate through the computational procedures. In the previous example, one problem that domain experts are interested in is to study whether two animals have stayed together for an extended period of time. Therefore, it is necessary to determine the trajectory of each animal and the distance between them in real time. We have found that using best estimates only, distances can vary considerably in short time intervals due to errors in samplings, which is not realistic since the actual position of the cows does not change sharply in a short period of time.

In general, these problems are challenging as errors will compound over calculations. Without proper estimates, false results will be introduced and propagated. More specifically, in a typical program we often have to check true/false conditions based on the value of variables, e.g., whether the distance between two cows is below a threshold. We use the following code sample to illustrate how conventional programs handle conditional decisions. In this example, we assume two arrays, a and b, are used to store the location trajectories of two cows.

```
//return true if intersection detected
bool intersection(a, b)
{
for (int i = 0; i<n; i++)
{ if (distance(a[i], b[i]) < Threshold)
return true;
}
return false;
}
```

However, by using single data samples instead of their probabilistic distributions over the ground truth, we will generate false positive and false negative comparison results at each conditional statement, such as the IF statements above. In this example, the calculation of the function *distance* will involve the use of multiple basic operators, such as multiplication, addition, and finding the square root. Using approxes, on the other hand, applications can instead ask probabilistic questions, such as whether two animals have stayed together for a probability that is higher than a given threshold. As can be seen, without an adequate abstraction of uncertain data, calculations of error propagation and determining whether a condition is true or false go well beyond the reach of many developers.

We next illustrate how the same piece of code should be modified by using approxes. If locations are approxes instead of scalars, the IF statement will only be meaningful under a statistical sense: what is the probability that, over the the distribution range of $a[i]$ and $b[i]$, their distance is smaller than the predefined $Threshold$? Hence, we should revise the above code into the following:

```
//Updated intersection testing
bool intersection(a, b)
{
for (int i = 0; i<n; i++)
{
if (Prob(distance(a[i],b[i])<Threshold) > 0.9)
return true;
}
return false;
}
```

In this updated version, a and b are no longer storing scalar values, but instead, the actual distributions of each location sampling. Hence, the program will explicitly reason about the probability. However, one main challenge is that the limited storage space of IoT devices do not allow us to directly represent the probabilistic distributions. Further, even a simple addition of two random variables will require calculating the convolution of their distributions, which is too energy-intensive for IoT devices. This is also the reason why although many probabilistic programming frameworks have been proposed in the context of machine learning, none of them can be applied to IoT devices [14], [15]. In contrast, we adopt a novel combination of inferences and samplings to represent the distributions of each random variable. Our unique approach is based on a combination of Bayesian inference for precise answers and sequential analysis for sampling-based answers. When Bayesian inferences alone are sufficient, we do not need to perform samplings. On the other hand, when more complex calculations are required, we determine whether we can already answer an IF assertion by iterative samplings of their values until a decision is made. During this process, we obtain increasingly accurate approximations of the variables. This decision significantly reduces the computational load on the embedded devices to handle complex combinations of probability variables.

C. Bayesian Inference

We next describe a second example that was first raised by Pearl et al [16] to demonstrate Bayesian inference. In this example, assume a house has an alarm system against burglary. Further, the alarm system can get occasionally set off by an earthquake. Two neighbors, Mary and John, who do not know each other, agree that if they hear the alarm, they will call you, but this is not guaranteed. The conditional probability of events are given in Figure 3. Now, suppose that John has

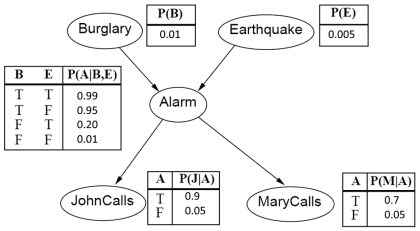


Fig. 3: Example of Bayesian reasoning over an burglary example.

called but Mary has not. How to reason about the probability of burglary?

This problem has conventionally been solved in the context of Bayesian inference and reasoning. That is, with posterior evidence that John has called, we can calculate the probability of specific events in a backward direction such as the likelihood for burglary. This can be solved by calculating the joint distribution of event combinations and apply the Bayesian rule on the graph model [17]. While this algorithm is not new, we find that being able to provide such type of support on IoT devices is helpful for a wide range of applications. Instead of requiring a server to run the inference tasks, an edge IoT device can instead perform such inference tasks individually, saving the cost of communication. In our design of approxes, we provide support for such Bayesian inference tasks by calculating the joint probability in both forward and backward direction on a Bayesian graph, and adjust the prior probability as new evidence from sensors becomes available. On the programmer side, they only need to specify the conditional graph as well as the initial conditions. The actual reasoning and inference is performed automatically on the generated graph model.

III. Design of Approxes

A. Overview and Examples

In this section, we present the design for the programming framework of approxes, including their associated operations to handle probability distributions of uncertain data. We use the C programming language as an example, but this framework can also be ported to other types of programming languages. The key data type for approxes is defined as a structure with associated runtime library functions. In contrast to existing probabilistic programming approaches, the approxes framework is primarily aimed at the growing number of IoT developers. Generally, an approx encapsulates a random variable of numerical type `T`. Here, `T` can be a double, an integer, or a boolean. Computations are defined over the approxes by constructing an implicit Bayesian network, which is a directed acyclic graph (DAG), where leaves represent the approxes and edges represent computational operations applied on the leaf nodes. The inner nodes are dependent variables, which represent the results of a sequence of operations computed on their children nodes. For example, the following code

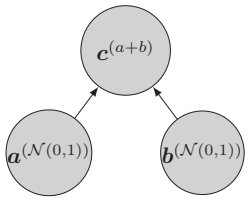


Fig. 4: Computational graph of $c = a + b$. Notice that c is not calculated until it is needed.

Categories	Functions
Leaf nodes	ApproxBool, ApproxInt, ApproxDouble, ApproxGeo
Dependents	Approx_1, Approx_2, etc.
Distributions	Flip, Gaussian, etc.
Conditions	Approx_Set_1, Approx_Set_2, etc.
Math	ApproxAdd, ApproxSub, ApproxMul, ApproxDiv, ApproxSqrt
Order	ApproxLessThan, ApproxGreaterThan, ApproxEqual
Logical	ApproxAnd, ApproxNot, ApproxOr
Reasoning	ApproxObserve, ApproxEstimate

TABLE I: List of approxes and operations

illustrates two approxes of the double type, each of them normally distributed between 0 and 1. A third approx, c, is defined as the addition of a and b.

```
//add two approxes together.
ApproxDouble *a = Gaussian (0,1);
ApproxDouble *b = Gaussian (1,1);
ApproxDouble *c = ApproxAdd (a, b);
```

The code implicitly creates a Bayesian network with two leaf nodes and a parent node for the computation result $c = a + b$. We evaluate this Bayesian network when a distribution of c is needed, which depends on the distribution of a and b. Figure 4 shows the graph construct. is working.

B. Grammar and semantics

We now describe the syntax and semantics of approx-based programs. Compared to using basic data types, programmers need to make very few changes when using approxes. For example, instead of defining an integer, the programmer needs to use the data type `ApproxInt`. This variable is a random variable that represents a probability distribution over a range of values. Table I shows a list of the supported data types and operators. Note that our random variable is defined in a type `T`, where `T` can be an integer, a floating-point number, a boolean, among others. For example, to express the coordinates of a geographic location consisting of x and y, we need a structure whose members x and y are defined as a floating-point based type of approxes. For each random variable, once defined, it can follow its own distribution, such as uniform, normal, exponential, Poisson, Rayleigh, among others. The supported distributions can be further extended as we provide the library as open-source software. In this way, each approx has an expected value and variance, which can be propagated through computational steps. For example, under the usual arithmetic operations, the variances can be scaled up or down.

One problem in writing programs with approxes is the correct choice of probability distribution. For uncertain data,

the probability distribution is problem specific. In many cases, developers already know these distributions, and sometimes they use this knowledge to define approxes. In other cases, developers can choose a broader range of nonparametric distributions. In general, developers have two ways to choose the distribution that fits their particular problem. They can choose a theoretical model, since many sources of uncertain data are available to developers with dedicated theoretical models. For example, according to the central limit theorem, the average error of a noisy data approximates a Gaussian distribution. Second, they can derive an empirical model. For example, for GPS measurements, experiments provide empirical models for the horizontal and vertical errors [13].

Due to space constraints, we cannot directly compute on the probability density functions on IoT devices for two reasons. First, the algebra of these random variables quickly becomes impractical due to multiple steps of calculations. Even the sum of two distributions requires the evaluation of convolution operations, a task usually too expensive for IoT devices. Second, many of the important distributions used for sensor-related purposes, especially the results of composite calculations, do not have a density function in closed form. Therefore, for storage reasons, they cannot be represented in this straightforward way. In fact, for the generic polynomial function $f(x)$, it is very difficult to obtain the density function even we know the density function of x. To overcome these problems, we choose to represent these distributions through a computation tree structure, and the root node can be sampled indirectly by calculating its value based on sampling those leaf nodes. That is, we sample for each approx that has a basic distribution supported by the programming library. Once its value is known, we use it to calculate the inner nodes upward to the root. Note that, given enough space and time, this approximation can be arbitrarily accurate by performing many samplings. For our problem, we usually have to achieve a trade-off between efficiency and accuracy.

C. Bayesian Reasoning

Once a tree is constructed, it can be used for both Bayesian reasoning and sampling purposes. We next describe how Bayesian reasoning can be expressed using the proposed syntax. Return to the burglary example earlier. We can write the following code for reasoning purposes.

```
ApproxBool *b = Flip(0.01); //burglary
ApproxBool *e = Flip(0.005); //earthquake
ApproxBool *alarm = Approx_2(b, e);

//set conditional probabilities
Approx_Set_2(alarm, false, false, 0.01);
Approx_Set_2(alarm, false, true, 0.2);
Approx_Set_2(alarm, true, false, 0.95);
Approx_Set_2(alarm, true, true, 0.99);

ApproxBool *JohnCalls, *MaryCalls;
JohnCalls = Approx_1(alarm);
MaryCalls = Approx_1(alarm);
```

Comparison of Conventional Program and Probabilistic Program

```
//get distance for two coordinates
//G.. stands for GeoLocation
double getDistance(G.. L1, G.. L2)
{
  return Sqrt((L1.x-L2.x)^2+(L1.y-L2.y)^2)
}

while (true)
{
//get coordinate readings from GPS
GeoLocation L1 = Cow1_getGPSReading();
GeoLocation L2 = Cow2_getGPSReading();
//if too near, raise alarm
double Distance = getDistance(L1, L2);
if (Distance < Threshold)
  RaiseAlarm();

}
```

```
//get distance for two coordinates
//A.. stands for ApproxGeoLocation
ApproxDouble *getApproxDistance(A.. *L1, A.. *L2)
{
  ApproxDouble *dx = ApproxMul(ApproxSub(L1.x-
L2.x),ApproxSub(L1.x-L2.x));
  ApproxDouble *dy = ApproxMul(ApproxSub(L1.y-
L2.y),ApproxSub(L1.y-L2.y));
  return ApproxSqrt(ApproxAdd(dx, dy));
}

while (true)
{
//get coordinate distributions from GPS
ApproxGeo *L1 = Cow1_getApproxGPSReading();
ApproxGeo *L2 = Cow2_getApproxGPSReading();
//L1 and L2 are approxes derived from GPS readings
//now get distance distribution
ApproxDouble *Distance = getApproxDistance(L1, L2);
//calculate probability compared to threshold
if (ApproxProbLessThan(Distance , Threshold) > 0.9)
  RaiseAlarm();

}
```

Fig. 5: Example of using approxes with GPS readings. Notice that when using approxes, a distribution is created and returned by the GPS sensor reading function instead of a concrete value.

```
//set conditional probabilities
Approx_Set_1(JohnCalls, true, 0.9);
Approx_Set_1(JohnCalls, false, 0.05);
Approx_Set_1(MaryCalls, true, 0.7);
Approx_Set_1(MaryCalls, true, 0.05);

//observed values to perform inference
ApproxObserve(JohnCalls, true)
ApproxObserve(MaryCalls, false)
double burglaryProb = ApproxEstimate(b);
```

This program is based on Figure 3, where we use the syntax of approxes to express the available evidence and condition the query on the unknown probability of burglary. Note that the programmer only needs to specify the observation outcomes. The actual inference procedures are performed automatically by the library function $ApproxEstimate$ based on available values.

D. Enhancing Bayesian Inference with Sampling

In real-world applications, it is less common for Bayesian inference to work end-to-end as many uncertain variables are not expressed with closed-form conditional distributions, as in the previous section, but are modeled by applying arithmetic, logical and comparison operators on them. The program then makes decisions differently depending on the uncertain results. We use a complete example to show how we handle uncertainty in conditional statement, such as the IF statement. Figure 5 represents a comparison of two different types of programs. Figure 6 shows a Bayesian network construction using leaf nodes for individual approxes. Here each leaf node is following a known distribution specified by the expert developer (e.g. Gaussian) and the top root node reflects the result of the calculation on the distributions of the distance between $L1$ and $L2$.

Generally, in a Bayesian network graph, the incoming edges of a node specify the other variables on which that node's variables depend. Observe that in the graph in Figure 6, the

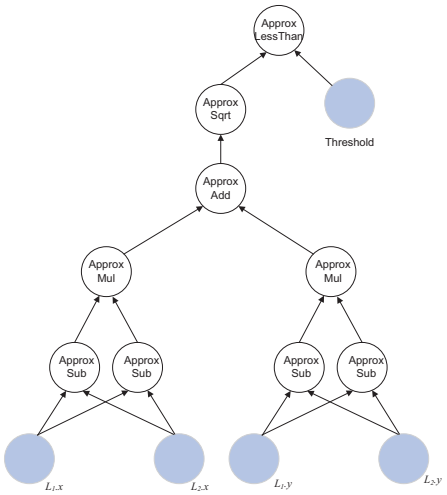


Fig. 6: Computational graph of the distance calculation.

two nodes $ApproxMul$ are based on the same $ApproxSub$ distributions. Then, the two $ApproxMul$ nodes must be multiplied together. Since Bayesian networks are constructed dynamically and incrementally during program execution, the resulting graph remains acyclic. Next, we need to test whether a conditional expression is true or not, such as the comparison between the root node $ApproxSqrt$ to $Threshold$ in Figure 6. Our core approach is to turn comparing a condition into a hypothesis testing problem. Specifically, we use approxes to encourage developers to ask appropriate questions about probabilistic data, e.g., "based on the available data and the assumed distribution, how confident are we that the distance exceeds a certain threshold?" Notice that in the program, we can use the confidence level as 0.9 in Figure 5. To illustrate the impact of distributions on the conditional check, Figure 7 shows two distributions of the distance, one Gaussian and one uniform, as well as the 90% percentile for the comparison $ApproxLessThan$ to be true (the shaded areas). Observe that for the Gaussian distribution the cutoff line is quite different from the uniform distribution. Furthermore, observe that even if the mean of the distribution is on one side of the threshold, the distribution may be so wide that the opposite conclusion is still likely to be true at lower thresholds. That is, a program using approxes can take both branches as true of the IF expression, if neither side of the conditional cannot reject the opposite hypothesis. On the other hand, the higher the threshold, the stronger the evidence needed, producing fewer false positives but more false negatives (which may be missed when the ground truth is true). Hence, we have to make decisions based on the unique characteristics for the distributions.

Ultimately, it is up to the developer to choose whether to encourage false positives or false negatives. Developers can reduce the limit of false positives by requiring stronger

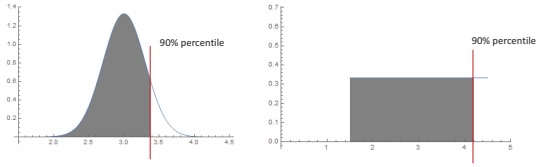


Fig. 7: Comparison of 90% percentile of different distributions. The first distribution is Gaussian(3,0.3) distribution, and the second is Uniform(1.5,4.5) distribution.

evidence, and vice versa. For each IF expression, the library function is backed up using statistical hypothesis testing to make inferences based on sample data. The algorithm works as follows. First, we create a hypothesis test when evaluating the conditional operator. In the above case, the null hypothesis H0: Distance $>$ Threshold and the alternate hypothesis H1: Distance $\leq$ Threshold. We then take a large number of samples through the computational graph and determine whether the conditional null hypothesis can be rejected based on the obtained sampling values, and whether the alternate hypothesis can be rejected. If the runtime sampling cannot reject either of them, then both of the hypothesis can be considered as possible. This behavior will have an impact on the execution of the program, where, in practice, an IF expression can pick 0, 1, or 2 branches for execution.

IV. Sequential Analysis based Decision-making

In this section, we describe the background of sequential analysis, which is the key theoretical foundation for us to decide how many samples to draw, and to reason about the probability of conditionals to be satisfied.

A. Problem Statement

We introduce the following problem: given a set of random variables each following their own distributions, and a computational DAG that performs operations on these variables, how many samples do we need to obtain at each leaf node, so that we can accept or reject the null hypothesis at a conditional statement for an inner node with statistical confidence? Note that the inner node may or may not the be top-level node in the DAG tree. The leaf nodes can be sampled as many times as needed.

B. Sequential Analysis

Our approach is based on sequential analysis [18], as it does not require excessive storage to keep random distributions in their complete forms. In this approach, we repeatedly take samples from the sources, that is, drawing from i.i.d. distributions until a decision is made. Sampling functions have no parameters and return a new random sample from the distribution on each call. For example, the pseudo-random number generator is a sampling function for a uniform distribution, and the Box-Mueller transform is a Gaussian distributed sampling function. For composite approxes that has multiple elements,

we need to draw a sample for each element to obtain a complete sample.

Once samples are obtained, we rely on sequential analysis to decide which hypothesis is correct. We adopt the commonly used method of sequential analysis called SPRT [19], [20], which starts with two hypothesis: $H_0 : p = p_0$ and $H_1 : p = p_1$, for an unknown distribution parameter p. The SPRT method then calculates the likelihood ratio as a function of the number of observations. It define that:

$$\Lambda_k := \prod_{i=1}^{k} \frac{p_1(X_i)}{p_0(X_i)}, k = 1, 2, \ldots$$

The next step is to calculate the cumulative products of the likelihood-ratio test, and the stopping rule is a simple thresholding scheme:

1) $a < \Lambda_k < b$: continue drawing samples;
2) $\Lambda_k \geq b$: Accept H_1;
3) $\Lambda_k \leq a$: Accept H_0;

Here, the values of a and b depend on the desired type I and type II errors, α and β. They may be chosen as follows:

1) α is the probability of rejecting H_0 when H_0 is true,
2) β is the probability of accepting H_0 when H_1 is true.

Generally, α and β must be decided beforehand in order to set the thresholds appropriately.

C. Analysis of Gaussian Distribution

We now apply the SPRT method to Gaussian distribution as an example, which is shown as follows. For other types of distributions, SPRT supports similar derivation steps.

Specifically, for a distribution $X \sim \mathcal{N}(\mu, \sigma^2)$, we draw a series of X_i, where $i = 0, 1, 2....$ We have two assumptions: H_0 be the hypothesis that the mean of X_i is μ_0, H_1 be the hypothesis that the mean of X_i is μ_1, different from μ_0. Therefore, the probability density function of X is:

$$p_i(X) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{X-\mu_i}{\sigma}\right)^2\right), i = 0, 1$$

Hence, we can calculate $\log \Lambda_k$ as:

$$\log \Lambda_k = \log \prod_{i=1}^{k} \frac{p_1(X_i)}{p_0(X_i)} = \sum_{i=1}^{k} \log \frac{p_1(X_i)}{p_0(X_i)}$$

Putting them together we can have $\log \Lambda_k = \sum_{i=1}^{k} t_i$, where $t_i = (\mu_1 - \mu_0)X_i + \frac{1}{2}(\mu_0^2 - \mu_1^2)$. Observe that X_i is a random variable and we have its expectation on H_0 and H_1 as μ_0 and μ_1, respectively. Therefore, we have $E(t_i|H_0) = -\frac{1}{2}(\mu_1 - \mu_0)^2$ and $E(t_i|H_1) = \frac{1}{2}(\mu_1 - \mu_0)^2$. Hence, given properly chosen A and B we see the result $\log \Lambda_k$ will either increase above a certain threshold or decrease below a certain threshold, hence proving the case that the results will be correct. Note this process will continue until a decision is made and the sampling procedure will stop. Later, in the evaluations, we show the empirical number of samples needed is affordable on embedded IoT devices.

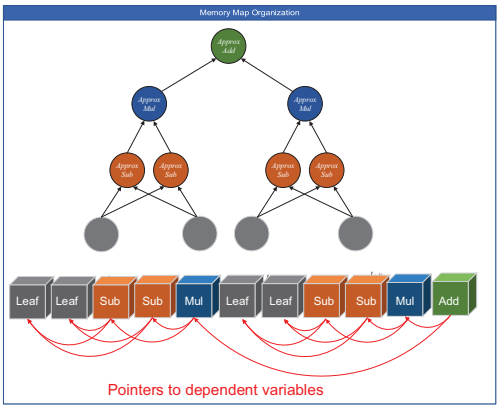


Fig. 8: Memory organization of the computational DAG on IoT devices.

V. Implementation

We discuss below how to implement approxes on IoT devices. We implemented the approxes on the Avr Atmega series microcontroller. The library implements both inference and sampling modules for hybrid programs. We store the tree structure into memory in a compact manner. Since it is difficult to maintain dynamic memory allocation in embedded systems, we use whole program optimization techniques to determine memory usage. Specifically, we identify the number of approxes created and used in a program before compiling the program with the compiler. Figure 8 shows the internal organization of the approxes in the array, where each element can be an approx or a derived approx. In the latter case, it contains references to other approxes by pointers.

VI. Performance Evaluation

In this section, we present the evaluation on approxes for uncertain sensor readings. We first demonstrate that when digital processing modules may yield false positives and negatives, approxes can effectively remove false positives and reduce the error rate to 0. Second, we demonstrate how we can improve accuracy of computations from a GPS location tracker using approxes. Finally, we evaluate the computational overhead and energy consumption of using approxes through repeated samplings.

A. Reducing noise and errors with approxes

In this evaluation, we demonstrate using approxes can effectively reduce Gaussian noise generated by sensor signal processing (DSP) algorithms. We consider an application scenario of alarm systems where multiple sensors are deployed to monitor certain events such as glass breaks. We carry out the experiments using twenty atmega2560 microcontrollers with acoustic sensor boards and pre-recorded acoustic signatures for glass-breaking events. The embedded signal processing module is responsible for converting the raw acoustic recordings into event reports (true or false reports). Without loss of

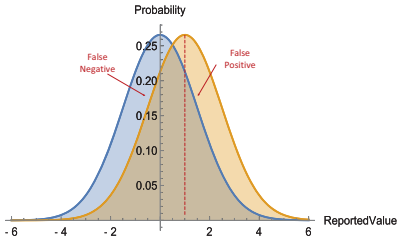


Fig. 9: Gaussian distribution of acoustic samples used in our evaluation. The blue curve represents the samples of raw signals (with noise added) without events, and the orange curve represents the samples of raw signals with events.

generality, we feed these acoustic sensors with inputs that are mixed with Gaussian noise with parameterized distributions. This allows us to take into consideration of arbitrary environment noise that may interfere with the accuracy of sensors. To improve accuracy, just like a typical house monitoring system, multiple acoustic sensors perform data aggregation to reduce false alarms.

Figure 9 demonstrates the underlying Gaussian noise we add to the sensors in their readings. We implement two types of signal processing modules to convert raw signals, where the signal input after mixing with noise as Gaussian. Note that this is not necessarily the case in more complex environments, but it demonstrates the most common scenarios and how our algorithms perform. Once a microcontroller samples from one of the curves, it is able to make a decision independently on whether a target event has been present. Due to the possible overlappings of the two Gaussian curves, both false positives and false negatives may be present.

We then compare the performance of two versions of event detection: the first version, BinarySensor, only reports 0 or 1 without the use of approxes. Once a sample is obtained, it compares this sample to the mean of the two distributions, and reports true or false by comparing the distances of the sample to the means. The second version, ApproxSensor, represents the probabilistic distributions of samples with approxes. It reads the samples as a random variable instead of binary outputs. Essentially, the version with ApproxSensor wraps each sensor with a distribution, so that each sensor may be sampled multiple times in a single deployment to make a decision.

Based on this implementation, we then measure the aggregate number of events by the total of 20 sensors. For caliberation needs, we change the number of events in the ground truth between 0 and 19. We then compare the reported number of events to the ground truth.

Figure 10 shows the results for comparing the use of approxes and the naive approach with binary sensors. Each experiment involving approxes implicitly performs a hypothesis test until a decision is made. Each experiment setting is run for 100 rounds to collect the number of samplings needed and their distribution is plotted.

Figure 10(a) shows the reported aggregation estimates with positive readings P drawn from the distribution $P \sim \mathcal{N}(0,1)$ and negative readings $Q \sim \mathcal{N}(0.5,1)$. Hence, there is a significant number of false alarms and we call the sensors as most noisy. Even with the number of actual events as 0, the sensors still reported on average 7.6 events due to false alarms. Similarly, in Figure 10(b) and Figure 10(c), we changed the distributions of Q to be farther away from P, which leads to better and more accurate reports. Specifically, the settings in Figure 10(b) draws from $P \sim \mathcal{N}(0,1)$ and negative readings $Q \sim \mathcal{N}(1,1)$, and Figure 10(c) draws from $P \sim \mathcal{N}(0,1)$ and negative readings $Q \sim \mathcal{N}(3,1)$, respectively.

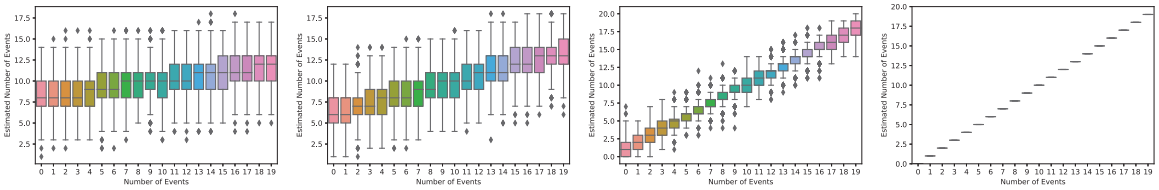
In Figure 10(d), we plot the results with approxes, with multiple samples drawn for each test. The error rates drop to 0, as multiple rounds of samplings from the distributions are made. We further plot the number of samplings for the settings in (a), (b), and (c), and plot the results in Figure 10(e) to (g). As shown, the more overlapping distributions require more samplings to obtain accurate answers for hypothesis test in the approx version of the implementation.

Finally, we plot the error rates of experiments (a) to (c) in (h). The number of incorrect decisions (y-axis) made by each noisy sensor at various noise settings are illustrated. As the noise becomes smaller from (a) to (c), the number of errors also decreases. This is consistent with the results from (a) to (c). Note that with the presence of false alarms, only when the number of actual events lies in the middle of total range, the error rate is the lowest.

B. Uncertain GPS Data

In this case study, we study the problem of location tracking using GPS sensors. Here, to better understand the ground truth, we use a flat parking lot to obtain GPS readings from two trackers. The trajectories of the deployment is shown in Figure 11(a), where multiple periodic intersections are illustrated. We first implement a naive version of a tracking program that calculates the raw distance between trackers using only the best estimates. Then, we implement an approx-based version that uses the measured EHPE (estimated horizontal position error) and EVPE (estimated vertical position error) to train a Rayleigh distribution model, and calculate the confidence intervals of the distances between two trackers. The measured EHPE distribution is shown in Figure 11(b), and the distribution of distances is shown in Figure 11(c).

Observe based on the experiment results, because of the uncertainty nature of the approx program, we have significantly improved the quality of GPS distance estimates. We can observe that the intersections cannot be properly captured by using single point GPS readings only (blue line), while the approx based approach successfully records more periodic intersections between the two trackers. On this basis, we can adjust the posterior distribution of the readings. As this program demonstrates, developers make only minimal changes to

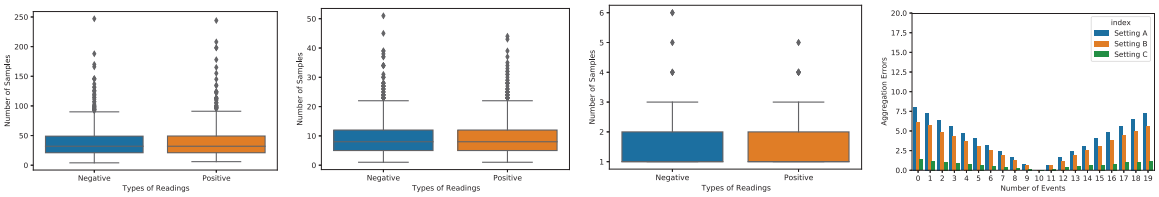


(a) Aggregation estimates with BinarySensor for most-noisy sensors (Setting A)

(b) Aggregation estimates with BinarySensor for medium-noisy sensors (Setting B)

(c) Aggregation estimates with BinarySensor for least-noisy sensors (Setting C)

(d) Aggregation estimates with ApproxSensor



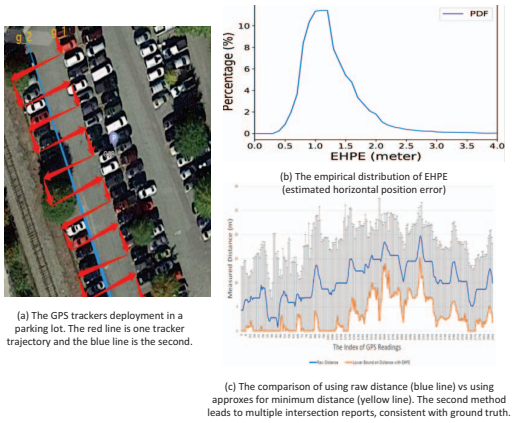
(e) Number of samplings required for aggregation estimates with ApproxSensor with setting A

(f) Number of samplings required for aggregation estimates with ApproxSensor with setting B

(g) Number of samplings required for aggregation estimates with ApproxSensor with setting C

(h) Average estimation errors with BinarySensor for settings A, B, and C

Fig. 10: Experiment comparison of BinarySensor vs. ApproxSensor for aggregation estimation.



(a) The GPS trackers deployment in a parking lot. The red line is one tracker trajectory and the blue line is the second.

(b) The empirical distribution of EHPE (estimated horizontal position error)

(c) The comparison of using raw distance (blue line) vs using approxes for minimum distance (yellow line). The second method leads to multiple intersection reports, consistent with ground truth.

Fig. 11: GPS tracking evaluation with and without approxes.

the original application and get benefits of increased accuracy. This is made possible by reasoning correctly about uncertainty and eliminating GPS errors better through modeling the distributions of errors in both horizontal and vertical directions.

VII. Related Work

This section introduces related work with similar probabilistic programming systems to our work. Several surveys from recent years investigated this in great details [15], [21], [22]. Here we mainly consider probabilistic programming approaches in this section.

Probabilistic programming: Several existing studies have attempted to introduce probabilistic programming in programming languages for statistical inference and support operations on probabilistic distributions. For example, the BUGS (Bayesian inference Using Gibbs Sampling) project [23] aims to develop flexible software for the Bayesian analysis of complex statistical models using Markov chain Monte Carlo (MCMC) methods. The Church programming language and its implementations [14], [24], [25] aimed to support probabilistic reasoning directly in the language level. For such problems, the program needs to automatically infer probability of various distributions and compute the posterior distribution.

Language designs: A variety of probabilistic programming systems attempt to achieve support for complex structures through a frontend/backend approach [26]–[28], such as the sample/observe paradigm, where probabilistic models are invoked for backend functions and implemented at the linguistic level through a checkpoint approach. Other systems, such as Gen [29], opt for a more integrated approach, using an extensible family of modeling languages and combinators to generate generative functions, which are essentially Julia objects conforming to the generative function interface [30].

Probabilistic programming on IoT devices. While the previous methods were mostly developed for machine learning where computational power is not a limitation, they meet challenges when implementing probabilistic programming on embedded systems [31]–[34]. Edge and IoT computing devices handle noisy data or make decisions in uncertain environments, and they require inexpensive and accurate probabilistic reasoning frameworks. However, existing algorithms are slow and often assume the need for accurate computation. Some recent work, such as Statheros [35], provides compilers for low-level, fixed-point approximation probabilistic programming. statheros compiles programs into fixed-point inference programs and is able to determine the best type of fixed-point to use. These research efforts are considerably different from ours in their methodology and goals.

VIII. Conclusions

This paper describes the probabilistic programming framework called the approxes, which offer flexible and extensible modeling and inference capabilities on IoT devices. We show that using approxes outperforms conventional programs on problems such as GPS tracking, estimating noise, among others. It is now feasible to add embedded domain-specific modeling languages based on approxes, each implementing its custom distributions interfaces, that capture problem structure from domains such as edge learning, inference, and decision making. We plan to make our contributions open-source and available for researchers and developers to use.

Acknowledgement

This work was supported, in part, by the U.S. NSF CPS/USDA NIFA. This data collected involved human subjects or animals in its research. Approval of all ethical and experimental procedures and protocols was granted by IACUC protocol under Protocol No. 2530-0620.

References

[1] Ron Mancini. Sensor to adc—analog interface design. *Analog Applications*, 2000.

[2] Afshin Haftbaradaran and Kenneth W Martin. A sample-time error compensation technique for time-interleaved adc systems. In *2007 IEEE Custom Integrated Circuits Conference*, pages 341–344. IEEE, 2007.

[3] Eulalia Balestrieri, Pasquale Daponte, and Sergio Rapuano. A state of the art on adc error compensation methods. *IEEE Transactions on Instrumentation and Measurement*, 54(4):1388–1394, 2005.

[4] James Rankin. An error model for sensor simulation gps and differential gps. In *Proceedings of 1994 IEEE Position, Location and Navigation Symposium-PLANS'94*, pages 260–266. IEEE, 1994.

[5] Simon DP Williams, Yehuda Bock, Peng Fang, Paul Jamason, Rosanne M Nikolaidis, Linette Prawirodirdjo, Meghan Miller, and Daniel J Johnson. Error analysis of continuous gps position time series. *Journal of Geophysical Research: Solid Earth*, 109(B3), 2004.

[6] Nabil M Drawil, Haitham M Amar, and Otman A Basir. Gps localization accuracy classification: A context-based approach. *IEEE Transactions on Intelligent Transportation Systems*, 14(1):262–273, 2012.

[7] Harald Cramér. *Random variables and probability distributions*. Number 36. Cambridge University Press, 2004.

[8] Jyrki Kullaa. Sensor validation using minimum mean square error estimation. *Mechanical Systems and Signal Processing*, 24(5):1444–1457, 2010.

[9] Sheng-Lan Ma, Shao-Fei Jiang, and Jun Li. Structural damage detection considering sensor performance degradation and measurement noise effect. *Measurement*, 131:431–442, 2019.

[10] Liang Heng, Grace Xingxin Gao, Todd Walter, and Per Enge. Statistical characterization of gps signal-in-space errors. In *Proceedings of the 2011 international technical meeting of the Institute of Navigation*, pages 312–319, 2011.

[11] SealHAT. Gps driver code example. https://github.com/SealHAT/SAM-M8Q/blob/master/gps.c.

[12] Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur. Bayesian event classification for intrusion detection. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 14–23. IEEE, 2003.

[13] KA Bin Ahmad, Mohamed Sahmoudi, and Christophe Macabiau. Characterization of gnss receiver position errors for user integrity monitoring in urban environments. In *Proceedings of the 2014 European Navigation Conference (ENC)-GNSS*, 2014.

[14] Noah D Goodman. The principles and practice of probabilistic programming. *ACM SIGPLAN Notices*, 48(1):399–402, 2013.

[15] Hrishav Bakul Barua and Kartick Chandra Mondal. Approximate computing: A survey of recent trends—bringing greenness to computing and communication. *Journal of The Institution of Engineers (India): Series B*, 100(6):619–626, 2019.

[16] JinHyung Kim and Judea Pearl. A computational model for causal and diagnostic reasoning in inference systems. In *International Joint Conference on Artificial Intelligence*, pages 0–0, 1983.

[17] Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[18] Abraham Wald. *Sequential analysis*. Courier Corporation, 2004.

[19] Bhaskar Kumar Ghosh and Pranab Kumar Sen. *Handbook of sequential analysis*. CRC Press, 1991.

[20] Tze Leung Lai. Likelihood ratio identities and their applications to sequential analysis. *Sequential Analysis*, 23(4):467–497, 2004.

[21] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[22] Qiang Xu, Todd Mytkowicz, and Nam Sung Kim. Approximate computing: A survey. *IEEE Design & Test*, 33(1):8–22, 2015.

[23] David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. Bugs 0.5: Bayesian inference using gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pages 1–59, 1996.

[24] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[25] WebPPL. Webppl website. http://webppl.org/.

[26] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A language for flexible probabilistic inference. In *International conference on artificial intelligence and statistics*, pages 1682–1690. PMLR, 2018.

[27] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032. PMLR, 2014.

[28] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

[29] Marco F Cusumano-Towner, Feras A Saad, Alexander K Lew, and Vikash K Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 221–236, 2019.

[30] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[31] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power mcus. In *2018 USENIX Annual Technical Conference*, pages 127–132, 2018.

[32] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: an analysis of iot devices on home networks. In *28th USENIX Security Symposium*, pages 1169–1185, 2019.

[33] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. *ACM SIGPLAN Notices*, 46(6):164–174, 2011.

[34] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the ASPLOS Conference*, pages 199–213. ACM, 2019.

[35] Jacob Laurel, Rem Yang, Atharva Sehgal, Shubham Ugare, and Sasa Misailovic. Statheros: Compiler for efficient low-precision probabilistic programming. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 787–792. IEEE, 2021.