Statheros: Compiler for Efficient Low-Precision Probabilistic Programming

Jacob Laurel, Rem Yang, Atharva Sehgal, Shubham Ugare, Sasa Misailovic
Department of Computer Science, University of Illinois at Urbana-Champaign

***Abstract*—As Edge and IoT computing devices process noisy data or make decisions in uncertain environments, they require frameworks for inexpensive, yet accurate probabilistic inference. Probabilistic programming has emerged as a powerful way for developers to write high-level programs, while abstracting away the implementation details of inference. However, the existing algorithms are slow and often assumed to require precise calculations. We present Statheros, the first compiler for low-level, fixed-point approximation of probabilistic programming. Statheros compiles programs to fixed-point inference procedures and is able to determine the optimal fixed-point type to use. We evaluate Statheros on 13 benchmarks and three embedded platforms. The results show that Statheros-generated code is 11.5x (Arduino), 3.8x (PocketBeagle), and 2.2x (Raspberry Pi) faster than single-precision floating-point computation, with minimal accuracy loss.**

I. Introduction

Probabilistic Programming (PP) [6] has emerged as a powerful paradigm for reasoning about uncertainty, as it offers an intuitive way to represent complex probability models as simple high-level programs. Application developers can abstract away the low-level details of how the inference, or learning, from the data is implemented. Probabilistic languages and compilers perform the heavy lifting to generate efficient procedures for Bayesian inference from high-level programs. Existing PP Languages (PPLs) are designed for powerful desktop or cloud-scale environments. Even there, probabilistic inference is computationally hard, and developers need to resort to approximate algorithms such as Markov chain Monte Carlo (MCMC).

Running probabilistic programs on resource-constrained systems (e.g., Edge or IoT) brings substantial challenges. These systems may have CPUs with limited processing power, or even lack support for floating-point calculations. This problem is further complicated by intensive numerical computations that, if naively implemented, can cause silent numerical overflows or underflows that hamper accuracy. Several studies designed novel customized hardware [9], [11], [21], presented non-scalable exact inference for restricted model classes [17], or used approximation without language-level integration [14]. Approximating programs using portable, low-precision numerical data types that still enable high-level probabilistic programs to run efficiently is an intriguing problem.

Statheros. We propose Statheros, an analysis and compilation system for fixed-point probabilistic programming. It exposes the precision knob for program variables, and offers flexible fixed-point numerical representations, instead of the default floating-point representations. A fixed-point representation stores numerical values in a specified number of *integer* and *fractional* bits (before and after the radix point, respectively), and fixed-point calculations use only integer operations.

Statheros presents the automatic selection of the optimal fixed-point representation (i.e., the number of integer and fractional bits for the program variables) that leads to a high level of accuracy. We use domain information to tailor this analysis to produce accurate results. We start from the observation that not all overflows are created equal, and only some may impact the accuracy of the inference. Our analysis then operates on the two semantic objects of the program: 1) the distribution functions and 2) the likelihood functions computed during inference. For each of these, Statheros performs a separate version of interval analysis to determine the optimal number of integer and fractional bits. Our analysis enables Statheros to use a minimal number of dynamic overflow checks during program execution, significantly reducing runtime overhead. Thus, the generated fixed-point code is not only fast, but also produces results that approximate the correct values well.

We present several optimizations on the level of the model that, informed by our analysis, improve the performance of the generated code. These include overflow-check reduction, PPL-level constant propagation, and memoization. Statheros's compiler generates the low-level C/C++ code for inference. It implements the Metropolis-Hastings MCMC algorithm, probability distributions, and standard math library functions (e.g., exp, log) with all computations in fixed-point representation. Its standard building blocks (fixed-point types, library, and translation to C/C++) enable Statheros to produce portable code that can run on many resource-constrained platforms.

Evaluation. We evaluate Statheros on 13 probabilistic programs from the literature. We evaluate three popular ARM CPU platforms – Arduino Due (with no hardware floating-point support), PocketBeagle, and Raspberry Pi. The evaluation shows that Statheros-generated code is 11.5x faster (geomean) on Arduino compared to soft-floats generated by a baseline compiler. Statheros-generated code is 3.8x faster on PocketBeagle and 2.2x faster on Raspberry Pi than floating-point versions running natively on these platforms. The accuracy losses of our optimized programs are negligible.

In summary, our main contributions are: (1) ***Fixed-Point PPL Compiler:*** Statheros is the first compiler for a PPL with *fixed-point* as its data type that generates fast, accurate, and portable code. (2) ***Program Analysis:*** We present a novel analysis for inferring the size of optimal fixed-point configuration, specialized to handle probability distributions and likelihoods. (3) ***Optimizations:*** We present how the analysis can guide several optimizations that improve program performance. (4) ***Evaluation:*** Our evaluation on 13 benchmarks and three platforms showcases Statheros's ability to generate efficient, accurate, and portable code.

978-1-6654-3274-0/21/$31.00 ©2021 IEEE

II. Overview of Statheros

A. Language

As our starting high-level PPL, we work with a simple imperative language extending from AutoPPL [20] (which is embedded in C++). Therefore, the syntax is C-like.

Syntax. Figure 1 presents the core syntax. The language exposes random primitives for *sampling*. For example, `r|=bernoulli(0.5)` expresses a coin flip with a probability of heads of 0.5. In PPLs, one may also write observe statements, `observe(B)`, for *conditioning* on a boolean event `B`.

```
Model    ::= Param+ ; Data+ ; DistStmt+
Param    ::= Param<Type>(...)
Data     ::= Data<Type>(...)
DistStmt ::= Var |= DistExpr | Var |= Expr
           | Var |= BExpr ? Expr : Expr
           | observe(BExpr)
           | for (i=c1; i<c2; i++){ Var[i] |= DistExpr }
DistExpr ::= bernoulli(Expr)
           | uniform(Expr, Expr)
           | normal(Expr, Expr)
BExpr    ::= BExpr ⊕ BExpr | Expr ⊙ Expr | true | false
Expr     ::= Expr ⊗ Expr | Var | c
Type     ::= int | real | fixed<c,c> | vector<Type>
    ⊕∈{&&,||,...}   ⊙∈{<,<=,==,...}   ⊗∈{+,−,*,...}
```

Fig. 1. The Syntax of the Statheros Base Language

Semantics. The semantic representation of a PP without infinite loops is a joint probability distribution over its variables, which can be encoded as a Bayesian Network (BN). Each node x_i corresponds to a distribution assignment, and dependencies between nodes (e.g., def-use) represent conditional probability relationships. To describe a joint probability distribution, one takes the conditional probability of each variable x_i conditional on its parent nodes in the dependence graph $\pi(x_i)$. Thus, the joint distribution becomes $Pr(\mathbf{x}) = \Pi_{i=1}^{n} Pr(x_i|\pi(x_i))$. One may then use this BN in combination with the observed data $d_{1...m}$ to perform inference and obtain a *posterior* distribution over model parameters, denoted $Pr(\mathbf{x}|d_{1...m})$.

MCMC-based Inference. Practitioners settle for approximate MCMC methods, which allow one to generate samples that approach the true desired distribution. Metropolis-Hastings is the most common MCMC inference algorithm. First, a proposal distribution $q(\mathbf{x})$ (which a PPL automatically selects) is used to propose a new sample $\mathbf{x_p}$ based on a current sample $\mathbf{x_c}$. These are then used to score the likelihood of the observed data d_i. If the proposed sample leads to a higher likelihood of the data, it is more likely to be accepted. Therefore, the most important step is the computation of the acceptance ratio, acc. To avoid underflow, one typically computes its logarithm:

$$\begin{aligned}\log(acc) = &\sum_{i=1}^{m}\big(\log(Pr(d_i|\mathbf{x_p})) + \log(Pr(\mathbf{x_p})) + \log(q(\mathbf{x_c}|\mathbf{x_p}))\big) \\ &- \sum_{i=1}^{m}\big(\log(Pr(d_i|\mathbf{x_c})) + \log(Pr(\mathbf{x_c})) + \log(q(\mathbf{x_p}|\mathbf{x_c}))\big)\end{aligned} \tag{1}$$

A PPL abstracts away all these details from the developer. Although Statheros builds the structure of the Bayes Net at compile time, the values and sizes of the data vectors used by MCMC need not be known at compile time.

Fixed-Point Arithmetic. Fixed-point arithmetic is an alternative to the IEEE 754 floating-point standard [1], particularly for embedded systems [8], which allows one to represent a real number as an *integer*. Formally, given a word size $w \in \{8,16,32,64,...\}$, a fixed-point type is a tuple $\langle S,N,F\rangle$ where S is the sign bit, $N \in \mathbb{N}$ is the number of integer bits and $F = w-1-N$ is the number of fractional bits. This means that all numbers that can be represented are contained in the interval $[-2^N, 2^N - 2^{-F}]$. All standard arithmetic operations can be implemented by using integer operations and bit shifts. Statheros by default uses a round-to-nearest mode to round fixed-point results to their closest representation.

B. Analysis and Compilation Algorithm

Algorithm 1 presents Statheros's compilation procedure. We operate on the program's data-flow graph G that models the dependencies between all variables. We use a combination of static program analyses (`GetDistributionIntervals`, `GetLikelihoodIntervals`, and `SelectSize`) to determine the worst-case model and likelihood interval bounds, stored in I_M and I_{LL}, respectively. These are then used to compute N_M and N_{LL}, the number of integer bits for the model and likelihood's respective fixed-point types, as well as their respective number of fractional bits F_M and F_{LL}. We then use aggressive compiler transformations (`AddDynamicChecks` and `ConstPropAndMemoization`) that allow us to improve speed by leveraging the properties of fixed-point arithmetic.

Algorithm 1 Compilation Procedure

```
Inputs: Probabilistic program P, Word size w
G ← Translate(P)
I_M, I_LL ← ∅                                   ▷ Initialize Data Structures
I_M ← GetDistributionIntervals(G)
I_LL ← GetLikelihoodIntervals(G, I_M)
N_M, F_M, N_LL, F_LL ← SelectSize(I_M, I_LL, w)
P_1 ← AddDynamicChecks(P, N_M, F_M, N_LL, F_LL)
P_2 ← ConstPropAndMemoization(P_1, G, N_M, F_M, N_LL, F_LL)
return P_2, ⟨N_M, F_M⟩, ⟨N_LL, F_LL⟩
```

Translation. The syntax presented in Fig. 1 is high-level so we first simplify the control flow and create the data-flow graph G. Each variable corresponds to a node in this graph. We desugar the conditional choice into a mixture distribution primitive. We desugar all `observe` statements to `bernoulli` indicator variables, whose argument `p` is the result of the Boolean evaluation of the observed condition (e.g., `bernoulli(x>y)`). Upon obtaining G, we propagate our analysis through it. After the optimizations, Statheros generates the low-level C++ code that can be consumed by the platform's backend compiler.

III. Domain-Specific Fixed-point Selection

Statheros employs fixed-point *range analysis* on both ***model distributions*** and ***log-likelihoods***. Our technical insight is that existing Bayesian methods [4] for propagating intervals can (a) be applied to the new problem of determining the best fixed-point configuration and (b) be fully automated as a program analysis. By deducing the interval that all variables lie in, we can choose a configuration that will avoid overflows.

A. *Propagating Intervals through Model Distributions*

The first step of range analysis of any MCMC computation is bounding the range of the samples themselves. As with existing PPLs, we require the posterior to have the same support as the prior, hence we only need to bound the range of the program's prior distributions. However, this still seems intractable: even obtaining interval bounds on simple Gaussians seems hopeless, since their support is $(-\infty,+\infty)$. However, random number generator (RNG) implementations can be truncated after a certain point (e.g., $\pm 6\sigma$) [18]. Our analysis exploits the fact that these intervals are *finite* in practice.

Statheros performs interval propagation on the data-flow graph G built during Translation. We propagate data-flow facts (each variable's interval) through successive variables and statements and store them in I_M. To determine each variable's range, we perform interval analysis on its assigned expression using `GetDistributionIntervals`. In I_M, each variable is mapped to an interval $[a,b]$, where $a,b \in \mathbb{R} \cup \{\pm\infty\}$ and $a \leq b$. The interval analysis of deterministic arithmetic expressions (e.g., $x + y$) is standard [7], but the interval analysis of distributions is not. We now detail this analysis.

- **Bernoulli**: For `r|=bernoulli(p)`, we know $r \in \{0,1\}$, hence $I_M[r] = [0,1]$.
- **Uniform:** Assume Statheros has already deduced $x \in [a,b]$ and $y \in [c,d]$. For `r|=uniform(x,y)`, we infer that $I_M[r] = [\min(a,c), \max(b,d)]$. If the analysis determines that the expression is ill-formed, meaning $\min(a,c) > \max(b,d)$, Statheros raises an error.
- **Gaussian:** Assume Statheros has already deduced $\mu \in [a,b]$ and $\sigma \in [c,d]$. For `r|=normal(μ,σ)`, we infer that $I_M[r] = [a - K \cdot d, b + K \cdot d]$ where K represents the maximum number of standard deviations the RNG sampler permits. For common RNGs, $K = 6$, but a user can provide a different value (or RNG), which would be used by the analysis.

We support several other common distribution classes: mixture distributions can be bounded by taking the union of the interval bounding each component distribution. Heavy-tailed distributions like Cauchy are also supported by this analysis as their RNGs still truncate the probability mass to lie within a finite range (that is a function of the distribution's parameters).

B. *Propagating Intervals through Likelihoods*

To perform the MCMC acceptance step, Statheros must score the likelihood of proposed samples as in Eq. 1. Each variable has a log-likelihood function through which Statheros will statically propagate intervals (stored in I_{LL}) via the `GetLikelihoodIntervals` pass. Since these log-likelihoods are also parameterized by sampled model values, Statheros leverages the previously-computed distribution intervals, I_M. Likewise, for any log-likelihoods involving observed data variables d, we simply use $I_M[d] = [\min_i(d_i), \max_i(d_i)]$ as their interval, provided these values are known a priori. If they are not, we simply use the range analysis of the observed variable's distribution. Lastly, since a log-likelihood is $-\infty$ for regions outside of a distribution's support, we keep track of this *separately* and ignore it for the size determination step as it does not influence the necessary fixed-point size. We now detail representative cases (with intervals not simplified for presentation):

- **Bernoulli**: Assume Statheros inferred $p \in [a,b]$ and that $0 \leq a,b \leq 1$. For `r|=bernoulli(p)`:
$$I_{LL}[r] = \log([a,b]) \cup \log(1 - [a,b])$$
- **Uniform**: Assume Statheros inferred $x \in [a,b]$, $y \in [c,d]$ with $b \leq c$. For `r|=uniform(x,y)`:
$$I_{LL}[r] = -\log\big([c,d] - [a,b]\big)$$
- **Gaussian:** Assume Statheros inferred $\mu \in [a,b]$, $\sigma \in [c,d]$, $c > 0$ and $I_M[r] = [r_1,r_2]$. For `r|=normal(μ,σ)`:
$$I_{LL}[r] = \log\left(\frac{1}{\sqrt{2\pi}[c,d]}\right) - \frac{1}{2}\left(\frac{[r_1,r_2] - [a,b]}{[c,d]}\right)^2$$

Statheros analyzes other distributions similarly. After bounding the range of the model's distribution samples and their respective log-likelihoods, the bound on the log of the acceptance ratio $\log(acc)$ in Eq. 1 can then be determined by standard interval arithmetic (as we can now bound all individual terms). Sec. III-C shows how to relax this bound.

Observe Statements. A PPL, unlike conventional languages, also supports `observe` statements during the log-likelihood sum computations that condition upon Boolean predicates. If the particular samples satisfy the predicate, the log-likelihood sum is unchanged. However, if the predicate is violated, the proposed sample whose likelihood was being computed is instantly rejected. Since this process does not depend on the size used, these statements can be safely ignored by the analysis.

C. *Selecting Fixed-Point Representation*

The next pass Statheros performs is the `SelectSize` pass to select the actual fixed-point data type for both model distributions and likelihoods. Statheros uses a separate type for both since their respective ranges can be drastically different.

Initial Configuration. We can determine the number of integer bits for model distributions, N_M, by finding the maximum magnitude: $N_M = \lceil \log_2(\max_{var \in Vars}(|I_M[var]|)) \rceil$. However, if the largest interval in I_M degenerates to $(-\infty,+\infty)$, then we simply use the largest possible number of integer bits, while still retaining 12 fractional bits. Likewise, for I_{LL}, we can use the conservative bound on $\log(acc)$ obtained by the interval analysis of Eq. 1 that sums over all data points.

Minimal Configuration. Since the bound on $\log(acc)$ may be much larger than other terms in I_{LL} or I_M, it begs the question, does one *need* this guaranteed bound? The answer is surprisingly **no**. We can select fewer integer bits than the conservative static analysis would suggest and allow the likelihood sums in the acceptance ratio to overflow, provided the true ratio is still within the given size's range, since fixed-point arithmetic in two's complement form is modular. We only need as many integer bits as the magnitude of the largest *individual* likelihood: $N_{LL} = \lceil \log_2(\max_{var \in Vars}(|I_{LL}[var]|)) \rceil$. However, if an *individual* likelihood overflows, its negative

value wraps around and becomes positive, incorrectly registering as a *high likelihood*, hence why it must not overflow.

Maintaining Portability. Statheros supports multiple different configurations for the number of integer and fractional bits. Statheros requires at least 12 fractional bits in all cases, as we found that using less fractional precision negatively impacted the accuracy. Likewise, we noticed diminishing returns of using more than 24 fractional bits. Furthermore, though our analysis supports any amount of fractional bits, many third-party libraries only support select sizes (e.g., Q16 or Q24), which may require developers to reimplement multiple versions of standard fixed-point functions. For simplicity in portability, we round the number of required fractional bits, F_M and F_{LL}, to the largest multiple of 4: $F_M, F_{LL} \in \{12, 16, 20, 24\}$ such that we still have $w-1-F_M \geq N_M$ and $w-1-F_{LL} \geq N_{LL}$. All remaining bits in w are then added to N_M and N_{LL}, respectively.

IV. Domain-Specific Program Optimizations

Dynamic Checks. Statheros can exploit the fact that overflows in the log-likelihood sum are permissible, provided the end result still lies within the representable range of the chosen fixed-point size. This follows directly from properties of two's complement arithmetic, described below along with reasons why MCMC computations are a natural fit for this arithmetic. We then simply have to check for this case at runtime, allowing us to omit almost all checks for arithmetic overflow, thus reducing overhead.

• *Two's Complement:* When subtracting two terms in two's complement, as long as their difference is representable, it does not matter if intermediate terms overflowed due to wrap-around. Consequently, even though the two summation terms in Eq. 1 sum over all m data observations, the *difference* between the log-likelihood sums for $\mathbf{x_p}$ (proposed sample) and $\mathbf{x_c}$ (current sample) is almost always inside the representable range. The reason is, in Metropolis-Hastings, the likelihood terms for $\mathbf{x_p}$ and $\mathbf{x_c}$ have similar magnitudes due to between-sample correlation.

• *Single Check:* The problem of checking every arithmetic operation now reduces to checking only if the final result is in the representable range. Algorithm 1 adds this check with the `AddDynamicChecks` function. If the two summation terms of Eq. 1 have opposite signs, we check if the positive term minus the max value of the likelihood type ($2^{N_{LL}} - 2^{-F_{LL}}$) is still greater than the negative term. This check is dynamically performed in each iteration of MCMC and its overhead is negligible. If the check returns `true` at any iteration *after burn-in*, we allow the computation to proceed but provide a warning that the MCMC inference may not have correctly converged. This check is also useful when the range analysis determined that the distribution or likelihood range degenerated to $(-\infty,+\infty)$, to know if an error caused by overflow occurred (even after using the maximum allowable number of integer bits).

Operation Optimizations. We particularly focus on reducing the cost of integer division (which is known to be up to 5x slower than addition and multiplication). Division is an important part of the log-likelihood computation for many distributions (e.g., Gaussian), yet we observe that it is often called with (1) a constant argument (e.g., Gaussian variance is often one), or (2) successive iterations reuse the values computed previously. We perform two optimizations:

• *Constant propagation:* As mentioned, many distributions whose likelihood will be computed have constant values for parameters, which conventional compilers cannot identify in BNs. Our library optimization computes these values and functions of them (e.g., the inverse of the standard deviation in the case of a Gaussian) only once.

• *Subexpression memoization:* Even when distributions do not have fully constant parameters, the same parameters may be used for many successive likelihood calculations (for each data point). Thus, we can memoize the subexpressions computed in one iteration and later reuse those values, provided they remain the same in the subsequent iteration. We leverage this temporal approximation by checking if the parameter is the same as in the previous function call, and if so, returning the pre-computed value.

Library Optimizations. Our library implements efficient functions for (1) distribution sampling, which efficiently generate uniformly random fixed-point numbers from uniformly random integers (which is much faster than floating-point sampling); (2) fixed-point versions of standard functions such as log, exp, sqrt for different fixed-point configurations and levels of accuracy.

V. Methodology

Benchmarks. We use a diverse set of probabilistic programming benchmarks used previously in the literature. We choose benchmarks with continuous distributions and large parameters (to observe how requiring more integer bits affects quality) as well as BNs with only Bernoulli distributions (to see how the number of fractional bits affects quality). GaussianStan, GenderHeight, IQStan, LinReg, Plankton, and TrueSkill fall into the former category while BetaBinomial, BurglarAlarm, Grass, and TwoCoins fall into the latter. Altimeter and ElectricPower are both used in embedded systems and contain categorical distributions. SVE also represents an embedded robotics use case and contains both triangular and mixture distributions. The data used for inference came with each benchmark.

Implementation. To implement Statheros, we extended the AutoPPL library [20] and its MCMC algorithm.

Inference. For each benchmark, we run 20 independent experiments for each type and average the results. We take 10K MCMC samples (plus 5K as burn-in) and the posterior mean as the point estimate, est. To obtain ground truth (*gt*) values, we use PSI's exact inference [5] for Altimeter, BetaBinomial, BurglarAlarm, ElectricPower, GenderHeight, and Grass. For all others, we run WebPPL [15] for 10^6 samples and take the mean as ground truth. Experimental error is computed as $|\frac{gt-est}{gt}|$.

Experimental Setup. We validate Statheros's functionality by comparing the accuracy and runtime of our fixed-point versions of standard probabilistic programs with both 32-bit

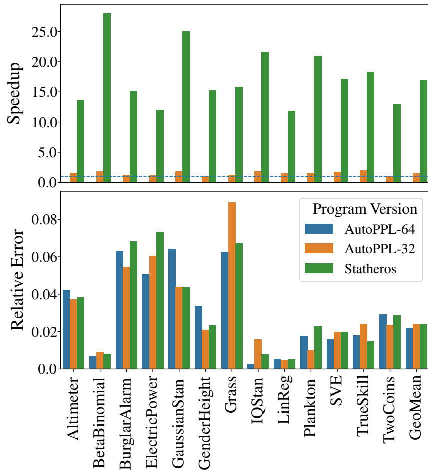

Fig. 2. **Arduino** Runtime and Accuracy

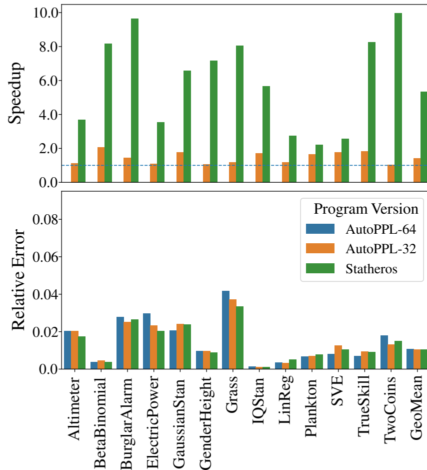

Fig. 3. **PocketBeagle** Runtime and Accuracy

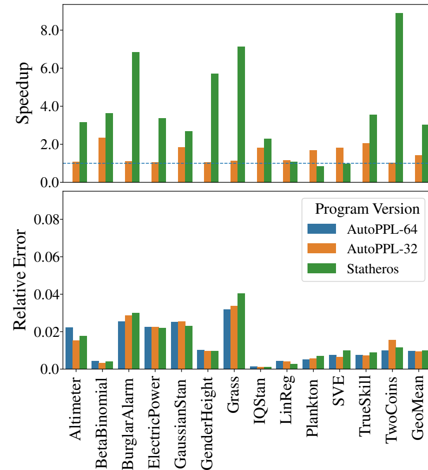

Fig. 4. **Raspberry Pi** Runtime and Accuracy

and 64-bit floating-point versions written in AutoPPL. To study how the embedded, resource-constrained setting affects these factors, we perform the evaluation on the following boards: an Arduino Due microcontroller (32-bit Atmel SAM3X8E M3, no FPU), a PocketBeagle SBC (32-bit Octavo OSD3358 A8, has FPU), and a Raspberry Pi 3 B+ SBC (64-bit Broadcom BCM2837 A53, has FPU). We compile to each platform using the gcc-arm compiler toolchain with `-O3` optimizations on C++17.

TABLE I
STATHEROS CONFIGS: $\langle I, F \rangle$, I INTEGER AND F FRACTIONAL BITS.

Benchmarks	**Param. Config**	**Likelihood Config**
Altimeter [16]	⟨7,24⟩	⟨7,24⟩
BetaBinomial [5]	⟨7,24⟩	⟨19,12⟩
BurglarAlarm [5]	⟨7,24⟩	⟨7,24⟩
ElectricPower [12]	⟨7,24⟩	⟨7,24⟩
GaussianStan [3]	⟨11,20⟩	⟨19,12⟩
GenderHeight [5]	⟨11,20⟩	⟨11,20⟩
Grass [5]	⟨7,24⟩	⟨7,24⟩
IQStan [3]	⟨11,20⟩	⟨19,12⟩
LinReg	⟨7,24⟩	⟨19,12⟩
Plankton [10]	⟨7,24⟩	⟨7,24⟩
SVE [6]	⟨7,24⟩	⟨19,12⟩
TrueSkill [6]	⟨11,20⟩	⟨7,24⟩
TwoCoins [5]	⟨7,24⟩	⟨7,24⟩

VI. EVALUATION

A. *Execution Time*

The upper plots of Figs. 2 - 4 present the speedup ratios relative to double-precision (horizontal line). The X-axis shows each benchmark; the Y-axis shows speedup. The bars represent single-precision computation (AutoPPL-32) and Statheros.

For the Arduino, the geomean speedup of Statheros compared to AutoPPL-32 and AutoPPL-64 was 11.54x and 16.91x, respectively. Statheros's Arduino raw runtimes ranged from 20ms (GenderHeight) to 427ms (LinReg). The difference stems from the lack of an FPU on the Arduino. The software-implemented floating-point (compiled using GCC) requires more instructions to complete arithmetic operations. Double-precision computation is slower, due to both more bit operations and transferring more data (each is 64-bit). In contrast, fixed-point computation uses only integer operations.

For the PocketBeagle, the geomean speedup of Statheros compared to AutoPPL-32 and AutoPPL-64 was 3.77x and 5.33x, respectively. Statheros's PocketBeagle raw runtimes ranged from 3.8ms (TwoCoins) to 98ms (LinReg). For the Raspberry Pi, the geomean speedup of Statheros compared to AutoPPL-32 and AutoPPL-64 was 2.15x and 3.04x, respectively. Statheros's Raspberry Pi raw runtimes ranged from 5.6ms (TwoCoins) to 119ms (LinReg).

Statheros was faster than float and double on all benchmarks on both the Arduino and PocketBeagle. For the Raspberry Pi, Statheros offered similar speedups on all benchmarks except Plankton, SVE, and LinReg – their speedup is smaller because the Gaussian likelihoods require expensive division operations.

B. *Accuracy*

The lower plots of Figs. 2 - 4 present the error of our inferred estimates relative to ground truth.

For the Arduino, the geomean error ratios over all benchmarks were 0.0239 (Statheros), 0.0238 (AutoPPL-32), and 0.0218 (AutoPPL-64). For the PocketBeagle and Raspberry Pi, the geomean error ratios over all benchmarks were 0.01 for Statheros, AutoPPL-32 and AutoPPL-64 (note that the benchmark errors have a small amount of variation due to randomness). The PocketBeagle and Raspberry Pi have such similar results since both use the same math library implementation that ships with the Linux kernel. In contrast, the Arduino uses a different math library specific to embedded systems.

On all benchmarks, the accuracy of Statheros with the inferred fixed-point sizes was comparable to both 32-bit and 64-bit floating-point AutoPPL, and in all benchmarks, the error was less than 10%. Models with smaller values (particularly with Bernoulli distributions) benefited from the precision of more fractional bits. Conversely, models with larger parameters could attain high accuracy with less precision.

To study how the inference execution *itself* changed (instead of its result), we measured the MCMC acceptance

ratios. In all cases except LinReg, the difference in acceptance ratios between AutoPPL-32, and Statheros was small, under 7%. For LinReg, the difference in acceptance ratio was slightly larger at 22%, with Statheros being slightly more efficient at accepting proposed samples.

Configurations. Table I presents the optimal configurations of $\langle N_M, F_M \rangle$ and $\langle N_{LL}, F_{LL} \rangle$ that Statheros inferred. Statheros was able to infer a finite bound in all cases except for BetaBinomial, GaussianStan, IQStan, and SVE. Additionally, it was critical to use this analysis, as we previously found that naively using standard sizes everywhere (e.g., $\langle 15{,}16 \rangle$) led to inaccurate results. As expected, there was a strong correlation between the model parameter values and the fixed-point parameter configuration. Many benchmarks only have 7 integer bits for model values, but significantly more for the likelihood. This is because for these models, the interval analysis infers that the log-likelihood sum can be a negative number that is large in magnitude (small probability events have log-likelihoods approaching $-\infty$). Likewise, a Gaussian's σ could be small ($\sigma \ll 1$), which leads to a large likelihood. This also justifies the choice of having separate fixed-point types for model values and likelihoods. For BetaBinomial, GaussianStan, IQStan, and SVE, the likelihood interval degenerated to $(-\infty,+\infty)$ in a manner that could *not* be ignored by Statheros's analysis. For these, we set the number of integer bits for the log-likelihood to be the maximum of the possible settings (19). Furthermore, in all cases, the dynamic check never reported an error, meaning that all benchmarks tolerated the approximation induced by their respective configurations.

C. *Impact of Optimizations*

Fully checking for overflows on every arithmetic operation added significant overhead to inference, taking 2.74x more time on average and 3.94x more time in total across all 13 benchmarks. By keeping only a single dynamic check to detect when the difference of model log-likelihoods lies outside the representable fixed-point range, Statheros adds essentially no overhead to detect erroneous results.

Memoization improved our execution time by 61.8% on average for the eight impacted benchmarks. LinReg experienced the biggest improvement, taking only 18.1% of the original execution time, because it operates on a large data set (which makes it expensive to calculate the log-likelihood).

VII. Related Work

From a language design standpoint, the majority of PP languages are sample-observe, but none support fixed-point computations. Probabilistic-C [13] is the closest in spirit to ours, though they support only floating-point data types and the code is not publicly available. Stan [3] is another popular language that compiles to low-level code, but its inference algorithm requires high (double) precision [2].

There is significant work in the approximate computing literature showing the benefits of lowering precision [19]. Though much of this work targets non-probabilistic techniques, there have been recent developments in lowering precision for Bayesian Inference [9], [17]. Most similar to ours is ProbLP [17], which compiles discrete Bayesian Networks to arithmetic circuits with user-specified precision. However, our work allows for both discrete and continuous models more expressive than arithmetic circuits, and our MCMC inference is more general than ProbLP's exact inference.

There is prior work in using low-precision for probabilistic inference at the *hardware* level [9], [11], [21]. However, these accelerators achieve gains by specializing hardware layouts for a specific class of problems, such as logistic regression [11] or image restoration [9]. Therefore, these architectures cannot be applied toward arbitrary Bayesian models and are thus not as expressive as a full-scale PPL. Our goal instead is to support *existing* general purpose hardware, but we anticipate that our results can also be useful for custom hardware.

VIII. Conclusion

Statheros is a language and compiler for allowing configurable low-precision probabilistic programming. We showed that probabilistic programs can operate with low-precision fixed-point computations. Our results show a significant performance improvement over single-precision float on three platforms – 11.5x (Arduino), 3.8x (PocketBeagle), and 2.2x (Raspberry Pi) with virtually no accuracy losses.

Acknowledgments

This work was supported in part by NSF Grants No. 1846354 and 2008883, Sloan Foundation, and a Facebook gift.

References

[1] IEEE standard for floating-point arithmetic. *ANSI/IEEE Std 754-1985*.
[2] B. Carpenter. Edward vs Stan performance. In *The Stan Forum*, 2017.
[3] B. Carpenter et al. Stan: A probabilistic programming language. *JSS'17*.
[4] S. Ferson et al. Constructing probability boxes and Dempster-Shafer structures. Technical report, Sandia National Lab., 2015.
[5] T. Gehr et al. PSI: Exact symbolic inference for probabilistic programs. In *CAV*, 2016.
[6] A. Gordon et al. Probabilistic programming. In *ICSE FoSE*. 2014.
[7] B. Jeannet et al. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.
[8] S. Jha et al. Synthesis of optimal fixed-point implementations of numerical software routines. 2013.
[9] G. Ko et al. Accelerating Bayesian inference on structured graphs using parallel Gibbs sampling. In *FPL*, 2019.
[10] J. Laurel et al. Continualization of probabilistic programs with correction. In *ESOP*, 2020.
[11] S. Liu et al. An unbiased MCMC FPGA-based accelerator in the land of custom precision arithmetic. *IEEE Trans. on Comp.*, 2016.
[12] O. Mengshoel et al. Sensor validation using Bayesian networks. i-SAIRAS, 2008.
[13] B. Paige et al. A compilation target for probabilistic programming languages. In *ICML*, 2014.
[14] N. Piatkowski et al. Integer undirected graphical models for resource-constrained systems. *Neurocomputing*, 2016.
[15] D. Ritchie et al. Deep amortized inference for probabilistic programs. *arXiv*, 2016.
[16] J. Schumann et al. Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *International Journal of Prognostics and Health Management*, 2015.
[17] N. Shah et al. ProbLP: A framework for low-precision probabilistic inference. In *DAC*, 2019.
[18] D. Thomas et al. Gaussian random number generators. *ACM Computing Surveys (CSUR)*, 39(4), 2007.
[19] Q. Xu et al. Approximate computing. *IEEE Design & Test*, 2015.
[20] J. Yang et al. AutoPPL; https://jamesyang007.github.io/autoppl/. 2020.
[21] X. Zhang et al. Statistical robustness of Markov chain Monte Carlo accelerators. ASPLOS, 2021.