Swift: Compiled Inference for Probabilistic Programming Languages

Yi Wu
UC Berkeley
jxwuyi@gmail.com

Lei Li
Toutiao.com
lileicc@gmail.com

Stuart Russell
UC Berkeley
russell@cs.berkeley.edu

Rastislav Bodik
University of Washington
bodik@cs.washington.edu

Abstract

A probabilistic program defines a probability measure over its semantic structures. One common goal of probabilistic programming languages (PPLs) is to compute posterior probabilities for arbitrary models and queries, given observed evidence, using a generic inference engine. Most PPL inference engines—even the compiled ones—incur significant runtime interpretation overhead, especially for contingent and open-universe models. This paper describes Swift, a compiler for the BLOG PPL. Swift-generated code incorporates optimizations that eliminate interpretation overhead, maintain dynamic dependencies efficiently, and handle memory management for possible worlds of varying sizes. Experiments comparing Swift with other PPL engines on a variety of inference problems demonstrate speedups ranging from 12x to 326x.

1 Introduction

Probabilistic programming languages (PPLs) aim to combine sufficient expressive power for writing real-world probability models with efficient, general-purpose inference algorithms that can answer arbitrary queries with respect to those models. One underlying motive is to relieve the user of the obligation to carry out machine learning research and implement new algorithms for each problem that comes along. Another is to support a wide range of cognitive functions in AI systems and to model those functions in humans.

General-purpose inference for PPLs is very challenging; they may include unbounded numbers of discrete and continuous variables, a rich library of distributions, and the ability to describe uncertainty over functions, relations, and the existence and identity of objects (so-called open-universe models). Existing PPL inference algorithms include likelihood weighting (LW) [Milch et al., 2005b], parental Metropolis–Hastings (PMH) [Milch and Russell, 2006; Goodman et al., 2008], generalized Gibbs sampling (Gibbs) [Arora et al., 2010], generalized sequential Monte Carlo [Wood et al., 2014], Hamiltonian Monte Carlo (HMC) [Stan Development Team, 2014], variational methods [Minka et al., 2014; Kucukelbir et al., 2015] and a form of approximate Bayesian computation [Mansinghka et al., 2013]. While better algorithms are certainly possible, our focus in this paper is on achieving orders-of-magnitude improvement in the execution efficiency of a given algorithmic process.

A PPL system takes a probabilistic program (PP) specifying a probabilistic model as its input and performs inference to compute the posterior distribution of a query given some observed evidence. The inference process does not (in general) execute the PP, but instead executes the steps of an inference algorithm (e.g., Gibbs) guided by the dependency structures implicit in the PP. In many PPL systems the PP exists as an internal data structure consulted by the inference algorithm at each step [Pfeffer, 2001; Lunn et al., 2000; Plummer, 2003; Milch et al., 2005a; Pfeffer, 2009]. This process is in essence an interpreter for the PP, similar to early Prolog systems that interpreted the logic program. Particularly when running sampling algorithms that involve millions of repetitive steps, the overhead can be enormous. A natural solution is to produce model-specific compiled inference code, but, as we show in Sec. 2, existing compilers for general open-universe models [Wingate et al., 2011; Yang et al., 2014; Hur et al., 2014; Chaganty et al., 2013; Nori et al., 2014] miss out on optimization opportunities and often produce inefficient inference code.

The paper analyzes the optimization opportunities for PPL compilers and describes the Swift compiler, which takes as input a BLOG program [Milch et al., 2005a] and one of three inference algorithms (LW, PMH, Gibbs) and generates target code for answering queries. Swift includes three main contributions: (1) elimination of interpretative overhead by joint analysis of the model structure and inference algorithm; (2) a dynamic slice maintenance method (FIDS) for incremental computation of the current dependency structure as sampling proceeds; and (3) efficient runtime memory management for maintaining the current-possible-world data structure as the number of objects changes. Comparisons between Swift and other PPLs on a variety of models demonstrate speedups ranging from 12x to 326x, leading in some cases to performance comparable to that of hand-built model-specific code.

To the extent possible, we also analyze the contributions of each optimization technique to the overall speedup.

Although Swift is developed for the BLOG language, the overall design and the choices of optimizations can be applied to other PPLs and may bring useful insights to similar AI
systems for real-world applications.

## 2 Existing PPL Compilation Approaches

In a general purpose programming language (e.g., C++), the compiler compiles exactly what the user writes (the program). By contrast, a PPL compiler essentially compiles the inference algorithm, which is written by the PPL developer, as applied to a PP. This means that different implementations of the same inference algorithm for the same PP result in completely different target code.

As shown in Fig. 1, a PPL compiler first produces an intermediate representation combining the inference algorithm (I) and the input model (PM) as the inference code (PI), and then compiles PI to the target code (PT). Swift focuses on optimizing PI by analyzing the static properties of the input PP, which are complementary to Swift.

### 3 Background

This paper focuses on the BLOG language, although our approach also applies to other PPLs with equivalent semantics [McAllester et al., 2008; Wu et al., 2014]. Other PPLs can be also converted to BLOG via static single assignment form (SSA form) transformation [Cytron et al., 1989; Hur et al., 2014].

### 3.1 The BLOG Language

The BLOG language [Milch et al., 2005a] defines probability measures over first-order (relational) possible worlds; in this sense it is a probabilistic analogue of first-order logic. A BLOG program declares types for objects and defines distributions over their numbers, as well as defining distributions for the values of random functions applied to objects. Observation statements supply evidence, and a query statement specifies the posterior probability of interest. A random variable in BLOG corresponds to the application of a random function to specific objects in a possible world. BLOG naturally supports open universe probability models (OUPMs) and context-specific dependencies.

Fig. 2 demonstrates the open-universe urn-ball model. In this version the query asks for the color of the next random pick from an urn given the colors of balls drawn previously. Line 4 is a number statement stating that the number variable #Ball, corresponding to the total number of balls, is uniformly distributed between 1 and 20. Lines 5–6 declare a random function color(), applied to balls, picking Blue or Green with a biased probability. Lines 7–8 state that each draw chooses a ball at random from the urn, with replacement.

Fig. 3 describes another OUPM, the infinite Gaussian mixture model (∞-GMM). The model includes an unknown
The Swift Compiler

number of clusters as stated in line 3, which is to be inferred from the data.

3.2 Generic Inference Algorithms

All PPLs need to infer the posterior distribution of a query given the observed evidence. The great majority of PPLs use Monte Carlo sampling methods such as likelihood weighting (LW) and Metropolis–Hastings MCMC (MH). LW samples unobserved random variables sequentially in topological order, conditioned on evidence and sampled values earlier in the sequence; each complete sample is weighted by the likelihood of the evidence variables appearing in the sequence. The MH algorithm is summarized in Alg. 1. M denotes a BLOG model, E its evidence, Q a query, N the number of samples, w a possible world, X(w) the value of random variable X in w, Pr[X(w)] denotes the conditional probability of X in w, and Pr[w] denotes the likelihood of possible world w. In particular, when the proposal distribution g in Alg.1 samples a variable conditioned on its parents, the algorithm becomes the parental MH algorithm (PMH). Our discussion in the paper focuses on LW and PMH but our proposed solution applies to other Monte Carlo methods as well.

4 The Swift Compiler

Swift has the following design goals for handling open-universe probability models.

- Provide a general framework to automatically and efficiently (1) track the exact Markov blanket for each random variable and (2) maintain the minimum set of random variables necessary to evaluate the query and evidence (the dynamic slice [Agrawal and Horgan, 1990], or equivalently the minimum self-supporting partial world [Milch et al., 2005a]).

- Provide efficient memory management and fast data structures for Monte Carlo sampling algorithms to avoid interpretive overhead at runtime.

For the first goal, we propose a novel framework, FIDS (Framework for Incrementally updating Dynamic Slices), for generating the inference code (\(P_T\) in Fig. 1). For the second goal, we carefully choose data structures in the target C++ code (\(P_T\)).

For convenience, r.v. is short for random variable. We demonstrate examples of compiled code in the following discussion: the inference code (\(P_T\)) generated by FIDS in Sec. 4.1 is in pseudo-code while the target code (\(P_T\)) by Swift shown in Sec. 4.2 is in C++. Due to limited space, we show the detailed transformation rules only for the adaptive contingency updating (ACU) technique and omit the others.

4.1 Optimizations in FIDS

Our discussion in this section focuses on LW and PMH, although FIDS can also be applied to other algorithms. FIDS includes three optimizations: dynamic backchaining (DB), adaptive contingency updating (ACU), and reference counting (RC). DB is applied to both LW and PMH while ACU and RC are specific to PMH.

Dynamic backchaining

Dynamic backchaining (DB) [Milch et al., 2005a] constructs a dynamic slice incrementally in each LW iteration and samples only those variables necessary at runtime. DB in Swift is an example of compiling lazy evaluation: in each iteration, DB backtracks from the evidence and query and samples an r.v. only when its value is required during inference.

For every r.v. declaration \(\text{random } X \sim C_X\); in the input PP, FIDS generates a getter function \(\text{get}_X()\), which is used to (1) sample a value for \(X\) from its declaration \(C_X\), and (2) memoize the sampled value for later references in the current iteration. Whenever an r.v. is referred to during inference, its getter function will be evoked.

Compiling DB is the fundamental technique for Swift. The key insight is to replace dependency look-ups and method invocations from some internal PP data structure with direct machine address accessing and branching in the target executable file.

For example, the inference code for the getter function of \(x(d)\) in the \(\infty\)-GMM model (Fig. 3) is shown below.

```cpp
double get_x(Data d) {
    // some code memoizing the sampled value memoization;
    // if not sampled, sample a new value
    val = sample_gaussian(get_mu(get_z(d)), 1);
    return val;
}
```

Memoization denotes some pseudo-code snippet for performing memoization. Since \(\text{get}_\mu()\) is called before \(\text{get}_\mu()\), only those \(\mu(u)\) corresponding to non-empty clusters will be sampled. Notably, with the inference code above, no explicit dependency look-up is ever required at runtime to discover those non-empty clusters.

When sampling a number variable, we need to allocate memory for the associated variables. For example, in the urn-ball model (Fig. 2), we need to allocate memory for color(b)
after sampling \#Ball. The corresponding generated inference code is shown below.

```c
int get_num_Ball() {  
    // some code for memoization
    memoization;
    // sample a value
    val = sample_uniformInt(1, 20);
    // some code allocating memory for color
    allocate_memory_color(val);
    return val;

    allocate_memory_color(val) denotes some pseudo-code segment for allocating val chunks of memory for the values of color(b).
```

**Reference Counting**

Reference counting (RC) generalize the idea of DB to incrementally maintain the dynamic slice in PMH. RC is an efficient compilation strategy for the interpretive BLOG to dynamically maintain references to variables and exclude those without any references from the current possible world with minimal runtime overhead. RC is also similar to dynamic garbage collection in programming language community.

For an r.v. being tracked, say \( X \), RC maintains a reference counter \( \text{cnt}(X) \) defined by \( \text{cnt}(X) = |\text{Ch}_w(X)| \). \( \text{Ch}_w(X) \) denote the children of \( X \) in the possible world \( w \). When \( \text{cnt}(X) \) becomes zero, \( X \) is removed from \( w \); when \( \text{cnt}(X) \) becomes positive, \( X \) is instantiated and added back to \( w \). This procedure is performed recursively.

Tracking references to every r.v. might cause unnecessary overhead. For example, for classical Bayes nets, RC never excludes any variables. Hence, as a trade-off, Swift only counts references in open-universe models, particularly, to those variables associated with number variables (e.g., \( \text{color}(b) \) in the urn-ball model).

Take the urn-ball model as an example. When resampling \( \text{drawn}(d) \) and accepting the proposed value \( v \), the generated code for accepting the proposal will be

```c
void inc_cnt(X) {  
    if (cnt(X) == 0) W.add(X);
    cnt(X) = cnt(X) + 1;
}
void dec_cnt(X) {  
    cnt(X) = cnt(X) - 1;
    if (cnt(X) == 0) W.remove(X);
}
void accept_value_drawn(Draw d, Ball v) {  
    // code for updating dependencies omitted
    dec_cnt(color(\text{val_drawn}(d)));
    val_drawn(d) = v;
    inc_cnt(color(v));
}
```

The function \( \text{inc}_\text{cnt}(X) \) and \( \text{dec}_\text{cnt}(X) \) update the references to \( X \). The function \( \text{accept}_\text{value}_\text{drawn}(d, v) \) is specialized to r.v. \( \text{drawn}(d) \): Swift analyzes the input program and generates specialized codes for different variables.

**Adaptive contingency updating**

The goal of ACU is to incrementally maintain the Markov Blanket for every r.v. with minimal efforts.

\( C_X \) denotes the declaration of r.v. \( X \) in the input PP; \( \text{Par}_w(X) \) denote the parents of \( X \) in the possible world \( w \); \( w[X \leftarrow v] \) denotes the new possible world derived from \( w \) by only changing the value of \( X \) to \( v \). Note that deriving \( w[X \leftarrow v] \) may require instantiating new variables not existing in \( w \) due to dependency changes.

Computing \( \text{Par}_w(X) \) for \( X \) is straightforward by executing its generation process, \( C_X \), within \( w \), which is often inexpensive. Hence, the principal challenge for maintaining the Markov blanket is to efficiently maintain a children set \( \text{Ch}(X) \) for each r.v. \( X \) with the aim of keeping identical to the true set \( \text{Ch}_w(X) \) for the current possible world \( w \).

With a proposed possible world (PW) \( w' = w[X \leftarrow v] \), it is easy to add all missing dependencies from a particular r.v. \( U \). That is, for an r.v. \( U \) and every r.v. \( V \in \text{Par}_w(U) \), since \( U \) must be in \( \text{Ch}_w(V) \), we can keep \( \text{Ch}(V) \) up-to-date by adding \( U \) to \( \text{Ch}(V) \) if \( U \notin \text{Ch}(V) \). Therefore, the key step is tracking the set of variables, \( \text{Δ}(w[X \leftarrow v]) \), which will have a set of parents in \( w[X \leftarrow v] \) different from \( w \).

\[
\text{Δ}(W[X \leftarrow v]) = \{ Y : \text{Par}_w(Y) \neq \text{Par}_w[w[X \leftarrow v](Y)] \}
\]

Precisely computing \( \Delta(w[X \leftarrow v]) \) is very expensive at runtime but computing an upper bound for \( \Delta(w[X \leftarrow v]) \) does not influence the correctness of the inference code. Therefore, we proceed to find an upper bound that holds for any value \( v \). We call this over-approximation the contingent set \( \text{Cont}_w(X) \).

Note that for every r.v. \( U \) with dependency that changes in \( w[X \leftarrow v] \), \( X \) must be reached in the condition of a control statement on the execution path of \( U \) within \( w \). This implies \( \Delta(w[X \leftarrow v]) \subseteq \text{Ch}_w(X) \). One straightforward idea is set \( \text{Cont}_w(X) = \text{Ch}_w(X) \). However, this leads to too much overhead. For example, \( \Delta(\cdot) \) should be always empty for models with fixed dependencies.

Another approach is to first find out the set of switching variables \( S_X \) for every r.v. \( X \), which is defined by

\[
S_X = \{ Y : \forall w, v \text{ Par}_w(X) \neq \text{Par}_w[w[Y \leftarrow v](X)] \},
\]

This brings an immediate advantage: \( S_X \) can be statically computed at compile time. However, computing \( S_X \) is NP-Complete\(^1\).

Our solution is to derive the approximation \( \hat{S}_X \) by taking the union of free variables of if/else conditions, function arguments, and set expression conditions in \( C_X \), and then set

\[
\text{Cont}_w(X) = \{ Y : Y \in \text{Ch}_w(X) \land X \in \hat{S}_Y \}.
\]

\( \text{Cont}_w(X) \) is a function of the sampling variable \( X \) and the current PW \( w \). Likewise, for every r.v. \( X \), we maintain a runtime contingent set \( \text{Cont}(X) \) identical to the true set \( \text{Cont}_w(X) \) under the current PW \( w \). \( \text{Cont}(X) \) can be incrementally updated as well.

Back to the original focus of ACU, suppose we are adding new the dependencies in the new PW \( w' = w[X \leftarrow v] \). There are three steps to accomplish the goal: (1) enumerating all the variables \( U \in \text{Cont}(X) \), (2) for all \( V \in \text{Par}_w(U) \), add \( U \) to \( \text{Ch}(V) \), and (3) for all \( V \in \text{Par}_w(U) \cap \hat{S}_V \), add \( U \) to \( \text{Cont}(V) \). These steps can be also repeated in a similar way to remove the vanished dependencies.

Take the \( \infty \)-GMM model (Fig. 3) as an example, when resampling \( z(d) \), we need to change the dependency of r.v. \( x(d) \)

\^1Since the declaration \( C_X \) may contain arbitrary boolean formulas, one can reduce the 3-SAT problem to computing \( S_X \).
Here $|w|$ denotes the total number of random variables existing in $w$, which can be maintained via RC.

Finally, the computation time of ACU is strictly shorter than that of acceptance ratio $\alpha$ (Eq. 2).

4.2 Implementation of Swift

Lightweight memoization

Here we introduce the implementation details of the memoization code in the `getter` function mentioned in section 4.1.

Objects will be converted to integers in the target code. Swift analyzes the input PP and allocates static memory for memoization in the target code. For open-universe models where the number of random variables are unknown, the dynamic table data structure (e.g., vector in C++) is used to reduce the amount of dynamic memory allocations; we only increase the length of the array when the number becomes larger than the capacity.

One potential weakness of directly applying a dynamic table is that, for models with multiple nested number statements (e.g. the aircraft tracking model with multiple aircrafts and blips for each one), the memory consumption can be large. In Swift, the user can force the target code to clear all the unused memory every fixed number of iterations via a compilation option, which is turned off by default.

The following code demonstrates the target code for the number variable `#Ball` and its associated `color(b)`’s in the urn-ball model (Fig. 2) where `iter` is the current iteration number.

```swift
vector<int> val_color, mark_color;
int get_color(int b) {
  if (mark_color[b] == iter) return val_color[b]; // memoization
  mark_color[b] = iter; // mark the flag
  val_color[b] = ...; // sampling code omitted
  return val_color[b];
}

int val_num_B, mark_num_B;
int get_num_Ball() {
  if (mark_num_B == iter) return val_num_B;
  mark_num_B = ...; // sampling code omitted
  val_num_B = ...;
  return val_num_B;
}
```

Note that when generating a new PW, by simply increasing the counter `iter`, all the memoization flags (e.g., `mark_color` for `color(b)`) will be automatically cleared.

Lastly, in PMH, we need to randomly select a r.v. to sample per iteration. In the target C++ code, this process is accomplished via polymorphism by declaring an abstract class with a `resample()` method and a derived class for every r.v. in the PP. An example code snippet for the $\infty$-GMM model is shown below.

```cpp
class MH_OBJ { public: //abstract class
  virtual void resample() = 0; // resample step
  virtual void add_to_Ch() = 0; // for ACU
  virtual void accept_value() = 0; // for ACU
  virtual double get_likeli() = 0;
}; // some methods omitted
// maintain all the r.v.s in the current PW
std::vector< MH_OBJ > active_vars;
```
Efficient proposal manipulation
Although one PMH iteration only samples a single variable, generating a new PW may still involve (de-)instantiating an arbitrary number of random variables due to RC or sampling a number variable. The general approach commonly adopted by many PPL systems is to construct the proposed possible world in dynamically allocated memory and then copy it to the current world [Milch et al., 2005a; Wingate et al., 2011], which suffers from significant overhead.

In order to generate and accept new PWs in PMH with negligible memory management overhead, we extend the lightweight memoization to manipulate proposals: Swift statically allocates an extra memoized cache for each random variable, which is dedicated to storing the proposed value for that variable. During resampling, all the intermediate results are stored in the cache. When accepting the proposal, the proposed value is directly loaded from the cache; when rejection, no action is needed due to our memoization mechanism.

Here is an example target code fragment for $mu(c)$ in the $\infty$-GMM model. proposed_vars is an array storing all the variables to be updated if the proposal gets accepted.

```cpp
// derived classes for r.v.s in the PP
class Var_mu: public MH_OBJ{
public://for mu(c)
    double val; // value for the var
    double cached_val; //cache for proposal
    int mark, cache_mark; // flags

    std::set<Var_mu*> Ch, Cont; // sets for ACU
    double getval() {...} //sample & memoize
    double getcache() {...} //generate proposal
    ...
}; // some methods omitted

std::vector<Var_mu*> proposed_vars;

class Var_z: public MH_OBJ{
... }; //for z(d)

Var_z* z[20]; // fixed-size array
```

Supporting new algorithms
We demonstrate that FIDS can be applied to new algorithms by implementing a translator for the Gibbs sampling [Arora et al., 2010] (Gibbs) in Swift.

Gibbs is a variant of MH with the proposal distribution $g(\cdot)$ set to be the posterior distribution. The acceptance ratio $\alpha$ is always 1 while the disadvantage is that it is only possible to explicitly construct the posterior distribution with conjugate priors. In Gibbs, the proposal distribution is still constructed from the Markov blanket, which again requires maintaining the children set. Hence, FIDS can be fully utilized.

However, different conjugate priors yield different forms of posterior distributions. In order to support a variety of proposal distributions, we need to (1) implement a conjugacy checker to check the form of posterior distribution for each r.v. in the PP at compile time (the checker has nothing to do with FIDS); (2) implement a posterior sampler for every conjugate prior in the runtime library of Swift.

5 Experiments
In the experiments, Swift generates target code in C++ with C++ standard <random> library for random number generation and the armadillo package [Sanderson, 2010] for matrix computation. The baseline systems include BLOG (version 0.9.1), Figaro (version 3.3.0), Church (webchurch), Infer.NET (version 2.6), BUGS (winBUGS 1.4.3) and Stan (cmdStan 2.6.0).

5.1 Benchmark models
We collect a set of benchmark models\footnote{Most models are simplified to ensure that all the benchmark systems can produce an output in reasonably short running time. All of these models can be enlarged to handle more data without adding extra lines of BLOG code.} which exhibit various capabilities of a PPL (Tab. 1), including the burglary model (Bur), the hurricane model (Hur), the urn-ball model (U-B(x,y)) denotes the urn-ball model with at most $x$ balls and $y$ draws, the TrueSkill model [Herbrich et al., 2007] (T-K), 1-dimensional Gaussian mixture model (GMM, 100 points with 4 clusters) and the infinite GMM model ($\infty$-GMM). We also include a real-world dataset: handwritten digits [LeCun et al., 1998] using the PPCA model [Tipping and Bishop, 1999]. All the models can be downloaded from the GitHub repository of BLOG.

5.2 Speedup by FIDS within Swift
We first evaluate the speedup promoted by each of the three optimizations in FIDS individually.

Dynamic Backchaining for LW
We compare the running time of the following versions of code: (1) the code generated by Swift (“Swift”); (2) the modified compiled code with DB manually turned off (“No DB”), which sequentially samples all the variables in the PP; (3) the hand-optimized code without unnecessary memoization or recursive function calls (“Hand-Opt”).
Table 1: Features of the benchmark models. R: continuous scalar or vector variables. CT: context-specific dependencies (contingency). CC: cyclic dependencies in the PP (while in any particular PW the dependency structure remains acyclic). OU: open-universe. CG: conjugate priors.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Bur</th>
<th>Hur</th>
<th>U-B(20,2)</th>
<th>U-B(20,8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DB</td>
<td>0.768</td>
<td>1.288</td>
<td>2.952</td>
<td>5.040</td>
</tr>
<tr>
<td>Swift</td>
<td>0.782</td>
<td>1.115</td>
<td>1.755</td>
<td>4.601</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.982</td>
<td>1.155</td>
<td>1.682</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Running Time (s): Swift vs. Hand-Optimized

Table 2: Performance on LW with $10^7$ samples for Swift, the version without DB and the hand-optimized version.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Bur</th>
<th>Hur</th>
<th>U-B(20,2)</th>
<th>U-B(20,8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No DB</td>
<td>$3 \times 10^7$</td>
<td>$3 \times 10^7$</td>
<td>$1.35 \times 10^8$</td>
<td>$1.95 \times 10^8$</td>
</tr>
<tr>
<td>Swift</td>
<td>$3 \times 10^7$</td>
<td>$2.0 \times 10^7$</td>
<td>$4.82 \times 10^7$</td>
<td>$1.42 \times 10^8$</td>
</tr>
<tr>
<td>Calls Saved</td>
<td>0%</td>
<td>33.3%</td>
<td>64.3%</td>
<td>27.1%</td>
</tr>
</tbody>
</table>

Table 3: Performance of PMH in Swift and the version utilizing static dependencies on benchmark models.

Reference Counting for PMH

RC only applies to open-universe models in Swift. Hence we focus on the urn-ball model with various model parameters. The urn-ball model represents a common model structure appearing in many applications with open-universe uncertainty. This experiment reveals the potential speedup by RC for real-world problems with similar structures. RC achieves greater speedup when the number of balls and the number of observations become larger.

We compare the code produced by Swift with RC (“Swift”) and RC manually turned off (“No RC”). For “No RC”, we traverse the whole dependency structure and reconstruct the dynamic slice every iteration. Fig. 5 shows the running time of both versions. RC is indeed effective – leading to up to 2.3x speedup.

Adaptive Contingency Updating for PMH

We compare the code generated by Swift with ACU (“Swift”) and the manually modified version without ACU (“Static Dep”) and measure the number of calls to the likelihood function in Tab. 3. The version without ACU (“Static Dep”) demonstrates the best efficiency that can be achieved via compile-time analysis without maintaining dependencies at runtime. This version statically computes an upper bound of the children set by $\overline{Ch}(X) = \{ Y : X \text{ appears in } C_Y \}$ and again uses Eq.(2) to compute the acceptance ratio. Thus, for models with fixed dependencies, “Static Dep” should be faster than “Swift”.

In Tab. 3, “Swift” is up to 1.7x faster than “Static Dep”, thanks to up to 81% reduction of likelihood function evaluations. Note that for the model with fixed dependencies (Bur), the overhead by ACU is almost negligible: 0.2% due to traversing the hashmap to access to child variables.

5.3 Swift against other PPL systems

We compare Swift with other systems using LW, PMH and Gibbs respectively on the benchmark models. The running time is presented in Tab. 4. The speedup is measured between Swift and the fastest one among the rest. An empty entry means that the corresponding PPL fails to perform inference on that model. Though these PPLs have different host languages (C++, Java, Scala, etc.), the performance difference resulting from host languages is within a factor of 2\(^3\).

5.4 Experiment on real-world dataset

We use Swift with Gibbs to handle the probabilistic principal component analysis (PPCA) [Tipping and Bishop, 1999], with real-world data: 5958 images of digit “2” from the handwritten digits dataset (MNIST) [LeCun et al., 1998]. We compute 10 principal components for the digits. The training and testing sets include 5958 and 1032 images respectively, each with 28x28 pixels and pixel value rescaled to $[0, 1]$.

Since most benchmark PPL systems are too slow to handle this amount of data, we compare Swift against other two

\(^3\)http://benchmarksgame.alioth.debian.org/
<table>
<thead>
<tr>
<th>PPL</th>
<th>Bur</th>
<th>Hur</th>
<th>U-B (20,10)</th>
<th>T-K</th>
<th>GMM</th>
<th>~-GMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW with 10^6 samples</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Church</td>
<td>9.6</td>
<td>22.9</td>
<td>179</td>
<td>57.4</td>
<td>1627</td>
<td>1038</td>
</tr>
<tr>
<td>Figaro</td>
<td>15.8</td>
<td>24.7</td>
<td>176</td>
<td>48.6</td>
<td>997</td>
<td>235</td>
</tr>
<tr>
<td>BLOG</td>
<td>8.4</td>
<td>11.9</td>
<td>189</td>
<td>49.5</td>
<td>998</td>
<td>261</td>
</tr>
<tr>
<td>Swift</td>
<td>0.04</td>
<td>0.08</td>
<td>0.54</td>
<td>0.24</td>
<td>6.8</td>
<td>1.1</td>
</tr>
<tr>
<td>Speedup</td>
<td>196</td>
<td>145</td>
<td>326</td>
<td>202</td>
<td>147</td>
<td>214</td>
</tr>
<tr>
<td>PMH with 10^6 samples</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Church</td>
<td>12.7</td>
<td>25.2</td>
<td>246</td>
<td>173</td>
<td>3703</td>
<td>1057</td>
</tr>
<tr>
<td>Figaro</td>
<td>10.6</td>
<td>-</td>
<td>59.6</td>
<td>17.4</td>
<td>151</td>
<td>62.2</td>
</tr>
<tr>
<td>BLOG</td>
<td>6.7</td>
<td>18.5</td>
<td>30.4</td>
<td>38.7</td>
<td>72.5</td>
<td>68.3</td>
</tr>
<tr>
<td>Swift</td>
<td>0.11</td>
<td>0.15</td>
<td>0.4</td>
<td>0.32</td>
<td>0.76</td>
<td>0.60</td>
</tr>
<tr>
<td>Speedup</td>
<td>61</td>
<td>121</td>
<td>75</td>
<td>54</td>
<td>95</td>
<td>103</td>
</tr>
<tr>
<td>Gibbs with 10^6 samples</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUGS</td>
<td>87.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>84.4</td>
<td>-</td>
</tr>
<tr>
<td>Infer.NET</td>
<td>1.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>77.8</td>
<td>-</td>
</tr>
<tr>
<td>Swift</td>
<td>0.12</td>
<td>0.19</td>
<td>0.34</td>
<td>-</td>
<td>0.42</td>
<td>0.86</td>
</tr>
<tr>
<td>Speedup</td>
<td>12</td>
<td>∞</td>
<td>-</td>
<td>∞</td>
<td>185</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 4: Running time (s) of Swift and other PPLs on the benchmark models using LW, PMH and Gibbs.

We also ran Stan with 50 and 100 tuning steps, which took more than 3 days to finish 130 iterations (including tuning). However, the results are almost the same as that with 9 tuning steps.

Swift quickly generates visually meaningful outputs with around 10^6 Gibbs iterations in 5 seconds. Infer.NET takes 13.4 seconds to finish the first iteration and converges to a result with the same perplexity after 25 iterations and 150 seconds. The overall convergence of Gibbs w.r.t. the running time significantly benefits from the speedup by Swift.

For Stan, its no-U-turn sampler [Homan and Gelman, 2014] suffers from significant parameter tuning issues. We also tried to manually tune the parameters, which does not help much. Nevertheless, Stan does work for the simplified PPCA model with 1-dimensional data. Although further investigation is still needed, we conjecture that Stan is very sensitive to its parameters in high-dimensional cases. Lastly, this experiment also suggests that those parameter-robust inference engines, such as Swift with Gibbs and Infer.NET with VMP, would be preferred at practice when possible.

6 Conclusion and Future Work

We have developed Swift, a PPL compiler that generates model-specific target code for performing inference. Swift uses a dynamic slicing framework for open-universe models to incrementally maintain the dependency structure at runtime. In addition, we carefully design data structures in the target code to avoid dynamic memory allocations when possible. Our experiments show that Swift achieves orders of magnitudes speedup against other PPL engines.

The next step for Swift is to support more algorithms, such as SMC [Wood et al., 2014], as well as more samplers, such as the block sampler for (nearly) deterministic dependencies [Li et al., 2013]. Although FIDS is a general framework that can be naturally extended to these cases, there are still implementation details to be studied.

Another direction is partial evaluation, which allows the compiler to reason about the input program to simplify it. Shah et al. [2016] proposes a partial evaluation framework for the inference code (P in Fig. 1). It is very interesting to extend this work to the whole Swift pipeline.

Acknowledgement

We would like to thank our anonymous reviewers, as well as Rohin Shah for valuable discussions. This work is supported by the DARPA PPAML program, contract FA8750-14-C-0011.

References


Figure 6: Log-perplexity w.r.t running time(s) on the PPCA model with visualized principal components. Swift converges faster.


