Generating Efficient MCMC Kernels from Probabilistic Programs

Lingfeng Yang **Pat Hanrahan** **Noah D. Goodman**
Stanford University

Abstract

Universal probabilistic programming languages (such as CHURCH [6]) trade performance for abstraction: any model can be represented compactly as an arbitrary stochastic computation, but costly online analyses are required for inference. We present a technique that recovers hand-coded levels of performance from a universal probabilistic language, for the Metropolis-Hastings (MH) MCMC inference algorithm. It takes a CHURCH program as input and *traces* its execution to remove computation overhead. It then analyzes the trace for each proposal, using *slicing*, to identify the minimal computation needed to evaluate the MH acceptance probability. Generated incremental code is much faster than a baseline implementation (up to 600x) and usually as fast as hand-coded MH kernels.

1 Introduction

In order to achieve high performance, machine learning practitioners typically intertwine the model representation and inference code. While this allows strong control over the underlying software and hardware, it is difficult to change the model, switch between inference algorithms, or express complex models.

To alleviate this, we can separate the model representation and inference code. In systems like JAGS [1], BLOG [2] and FACTORIE [3], graphical models serve as the representation. Inference algorithms can be switched without changing the model, and complex models can be expressed using abstractions such as plate models and template factors. Finally, many optimizations are possible in this setting, such as FAC-

TORIE's `DiffList` mechanism, which restricts computation to a particular Markov blanket at a time.

However, this reliance on an underlying graphical model limits the space of distributions that can be represented. In particular, these systems do not support models with recursive dependencies or non-trivial deterministic dependencies. For instance, FACTORIE requires user-specified proposals that explicitly maintain deterministic relationships [3]. With a graphical model, it is also necessary to explicitly declare the dependencies between random variables, which can become cumbersome.

A more intuitive and general representation is the *universal probabilistic language* (UPL). UPLs allow specification of any computable distribution [4,5] as a program with random choice primitives. The space of expressible models is essentially the space of expressible programs, including recursion and higher-order design patterns. Combinations of arbitrary stochastic and deterministic dependence is allowed as a matter of the natural course of program execution, not through explicit user declaration. Posterior inference is expressed using query functions that represent conditional probabilities. Examples are CHURCH [6], HANSEI [7], IBAL [8], and Infer.NET Fun [9].

Unsurprisingly, inference in these languages can be very slow. The set of random choices may change arbitrarily and execution of the program must be closely observed to track them. In particular, Metropolis-Hastings (MH) for universal probabilistic languages can be formulated in a straightforward way by re-running the program from scratch when a proposal is made to a random choice, using an expensive naming scheme based on the current stack trace to track random choices [10].

In this paper we show that by adapting techniques from the programming languages literature, *tracing* and *slicing*, we can bridge the gap between generality and efficiency. We extend LIGHTWEIGHT-MH [10], a general, robust MCMC algorithm for universal probabilistic languages. During inference on a given program, our technique records the execution

```
(letrec ([num-sites (randint 3 5)]
         [sites (repeat num-sites flip)]
         [constr (for-each
                    (lambda (xy)
                      (soft-eq (car xy) (cadr xy)))
                    (bigram sites))])
  sites)
```
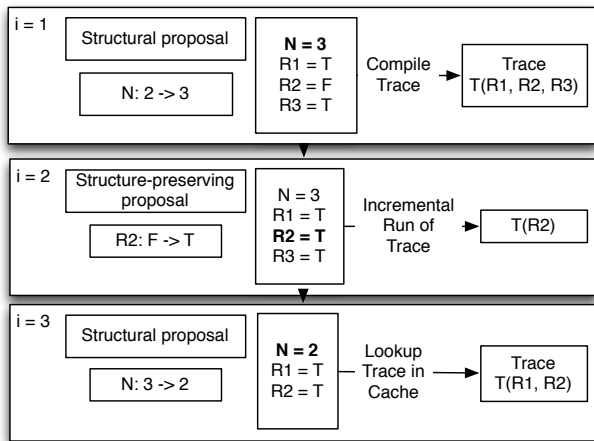
Figure 1: An example model.

Figure 2: A few iterations of our technique on the example model.

of the language's primitive operations, both stochastic and deterministic. This builds a trace of the density computation. By restricting this trace to *structure-preserving* choices that do not influence existence of random choices, we remove much of the overhead from LIGHTWEIGHT-MH. This results in an order of magnitude speedup versus unoptimized LIGHTWEIGHT-MH and performance similar to that of hand-written code (and sometimes faster).

We then further optimize these traces using *slicing* to extract the smallest sub-trace that needs to be run when a single random choice changes—thus performing the minimum amount of work per proposal. With this optimization, MCMC kernel performance can be more than an order of magnitude faster than that of the tracing interpreter alone, in total up to 600 times the performance of unoptimized LIGHTWEIGHT-MH.

2 Overview

In this section we describe the inefficiencies of LIGHTWEIGHT-MH. Then, we show how to use tracing and slicing techniques to generate code for a LIGHTWEIGHT-MH kernel specialized to a model. The input program is run through our tracing interpreter, generating traces that compute the density with less

work, thus allowing each MCMC iteration to perform less work. Then, slicing produces sub-segments of these traces that further reduce the work required. The result is a faster MCMC kernel.

Figure 2 shows a 1D Ising model, with unknown number of sites and pairwise compatibility factors between consecutive sites. Many widely-used models in machine learning follow this basic pattern. The num-sites random variable samples the number of variables using the randint random choice function. sites is a list of random numbers of length num-sites (generated by repeat), each a random boolean value generated by flip (a Bernoulli). soft-eq is the compatibility factor and is defined as a random choice with unit return value. It weights the distribution by probability 1.0 if the inputs are equal, and probability 0.1 otherwise. It is applied to every consecutive pair of sites using bigram, which takes a list (x1 x2 ...) to the list of consecutive pairs ((x1 x2) (x2 x3) ...).

One can perform MH on this model using LIGHTWEIGHT-MH [10]: building and tracking a database of random choices as the program runs, perturbing one of the choices and re-running the program with respect to the updated database to obtain an acceptance ratio. Let N, N' be the current (and proposed) values of num-sites, and x_i, x'_i be the current and proposed values of the individual sites. The acceptance ratio is then

$$J(x_1 \ldots x_N \to x'_1 \ldots x'_{N'}) \frac{\prod_{i=1}^{N'} p(x'_i) \prod_{i=1}^{N'-1} f(x'_i, x'_{i+1})}{\prod_{i=1}^{N} p(x_i) \prod_{i=1}^{N-1} f(x_i, x_{i+1})},$$

where $J(\cdot \to \cdot)$ is the correction for the proposal.

This is inefficient in two ways:

1. To determine which choices in the program correspond to which x_i, the *addressing scheme* is used, which is based on retrieving the stack trace at every random choice. The identity of each choice corresponds to the current stack. This is unnecessary when few or no random choices have been added or removed during the proposal.

2. The entire acceptance ratio is always computed, regardless of which variable was actually perturbed in the proposal.

Now, we describe how our incrementalizing compiler removes the overhead of 1) by *tracing*, and automates the algebraic simplification needed to mitigate 2) by *slicing*. For illustrative purposes, suppose that $N = N' = 3$, and we are proposing a change to x_1.

Tracing preserves structure. Our tracing interpreter generates a trace for the density computation over x_i given a fixed N. The result is a fixed set of program variables that correspond to the x_i, allowing density computation without re-computation of names:

```
X0 <- T-score randint R0 0 1
X1 <- T-score randint R1 0 1
X2 <- T-score randint R2 0 1
X3 <- T-score soft-eq X0 X1
X4 <- T-score soft-eq X1 X2
```

The trace computes the density as follows. `T-score randint ...` corresponds to the $p(x_i)$ factors and `T-score soft-eq` corresponds to the $f(x_i, x_{i+1})$ factors. As these functions are called, they implicitly increment a global score variable. The current state (x_1, x_2, x_3) is assigned to the *input variables* `R0, R1, R2`, and the density is the value of the score variable after being updated by the trace.

Yet, how do we determine which choices to treat as fixed program variables in general? We need to distinguish choices that, when changed, require name re-computation versus those that do not. We shall call these two types of choices *structural* and *structure-preserving* choices, respectively.

Control flow determines structure. A simple concept distinguishes these two types of choices. In a universal probabilistic language, the existence of random choices (and therefore the need for name re-computation) can only be influenced by *control flow constructs*: branching, looping, and recursion. Going back to the example, N was passed as a parameter to `repeat`, whose definition is given here:

```
(define (repeat n f)
  (if (= n 0) '() (cons (f) (repeat (- n 1) f))))
```

We see that N flows into the `if` statement, possibly causing the function `f` to be called, which in our case is used to generate the x_i choices. Therefore, N is structural. When N changes, so does the trace. On the other hand, calls to `flip` influence subsequent `soft-eq` choices but not their existence. This makes the x_i structure-preserving. Note that by this definition, the subclass of programs corresponding to graphical models have no structural choices.

Slicing simplifies scoring. Slicing is based on a conceptually simple optimization: with the proposal to x_1, there is no need to re-compute the entire density in the acceptance ratio:

```
ShredMHLoop(niter, proposals, scorer, db, trace):
for(i = 0, i < niter; i++):
    j = select_proposal_var_index(db)
    if structural?(db, j):
        xj = proposals[j](j)
        prop_hash = hash(db, j, xj)
        if (trace_exists(prop_hash)):
            prop_trace = traces[prop_hash]
        else
            prop_trace = gen_trace(db, j, xj)
        bwfw = q(trace, db, j, xj, prop_trace);
        diffscore = score(prop_trace) - score(trace);
        accept? = logU(0, 1) < bwfw + diffscore;
        if accept?:
            trace = prop_trace
            db = prop_db
    else
        xj_prev = copy(db[j])
        dscore_before = run_slice(trace, j)
        db[j] = proposals[j](db[j])
        dscore_after = run_slice(trace, j)
        bwfw = q(trace)
        diffscore = dscore_after - dscore_before;
        accept? = logU(0, 1) < bwfw + diffscore;
        if not accept?:
            db[j] = xj_prev
            run_slice(trace, j)
```

Figure 3: Pseudocode summarizing our technique.

$$\frac{q(x_1|x_1')}{q(x_1'|x_1)} \frac{p(x_1')f(x_1', x_2)}{p(x_1)f(x_1, x_2)} \frac{\prod_{i=2}^{N} p(x_i) \prod_{i=2}^{N'-1} f(x_i, x_{i+1})}{\prod_{i=2}^{N} p(x_i) \prod_{i=2}^{N'-1} f(x_i, x_{i+1})}$$
$$= \frac{q(x_1|x_1')}{q(x_1'|x_1)} \frac{p(x_1')f(x_1', x_2)}{p(x_1)f(x_1, x_2)};$$

i.e., it simplifies into factors relevant to the changed variable.

With the traces above, automatically performing this optimization amounts to following dependencies of statements. We use *slicing* to do so. Slicing is a general set of techniques for isolating the part of a program dependent on a given variable or value. For a proposal to x_1, we calculate the values dependent on `R0`, the corresponding input variable. This results in the following *slice*, which performs less computation:

```
X0 <- T-score randint R0 0 1
X3 <- T-score soft-eq X0 X1
X4 <- T-score soft-eq X1 X2
```

Generating/switching traces. Each trace only computes the density for a particular set of structure-preserving choices. If a proposal to a structural choice is made, we need to generate or retrieve a new trace. For instance, if a proposal to N changes it from 3 to 4, the next state is $N, (x_1 \ldots x_4)$ and the trace for $N, (x_1 \ldots x_3)$ cannot be used. Figure 2 depicts a scenario of three MH iterations on this model, showing how we alternate between running and existing trace and building (or retrieving) a new trace if needed.

We give pseudocode summarizing our method in Fig-

ure 3, depicting when we generate or retrieve a new trace and when we run an existing trace. `q` is a function that retrieves the proposal correction factor $\frac{q(\text{prev}|\text{next})}{q(\text{next}|\text{prev})}$. `run_slice` is a function that runs the slice corresponding to a certain variable, giving the corresponding density. It is run multiple times to maintain consistency between iterations.

3 Tracing and Slicing

We formally describe our technique in this section. We build on the formulation used in the paper [10] describing LIGHTWEIGHT-MH, which we briefly review here. Each run of the probabilistic program assigns values to a vector of random choices $\mathbf{X} = (x_1 \ldots x_K)$, where K is assumed to upper bound the maximum number of steps in the computation, thus leaving some of the elements in the vector to correspond to non-existent random choices. Let θ_i be the parameters associated with each x_i. The distribution associated with the program is then $P(\mathbf{X}) = \prod_{i=1}^{K} p(x_i|\theta_i, x_1 \ldots x_{i-1})$. To save space, let f_i denote $p(x_i|\theta_i, x_1 \ldots x_{i-1})$.

We split the x_i into *structural* choices x_S that affect existence of other choices and *structure-preserving* choices x_N that do not. Without loss of generality, let $x_S = x_1 \ldots x_S$ and $x_N = x_{S+1} \ldots X_K$ be the structure-preserving choices. Note that this does not violate the conditional dependencies $p(x_i|\theta_i, x_1 \ldots x_{i-1})$. No structure-preserving choice may influence any structural choice, otherwise it would be structural. Then the probability density of program executions is

$$P(\mathbf{X}) = \prod_{i=1}^{S} f_i \prod_{i=S+1}^{N} f_i = P(\mathbf{S})P(\mathbf{N}).$$

3.1 Tracing

Our tracing algorithm dynamically constructs subsets of $P(\mathbf{N})$ that correspond to extant sets of structure-preserving choices. We define our tracing interpreter as acting on a CHURCH-like, call-by-value functional probabilistic language. This is for clarity and precision; in principle, our technique applies to any language admitting an implementation of LIGHTWEIGHT-MH. The CHURCH language syntax is given below as a grammar. `v` are variables, `c` constants, `p` density functions, `op` primitive operations (such as $+, -, \times$, `cons, car`, and `cdr`), and `smp` primitive sampling functions. Brackets denote a list of the included element.

```
e = lambda [v] e | app e [e]
    | if e e e | op [e]
    | v | c | letrec v = e in e
    | S p smp [e] | N p smp [e]
```

This is lambda calculus with structural (`S`) and structure-preserving (`N`) elementary random choice primitives (ERPs) [10]. Each ERP is parameterized by a scoring function, a sampling function, and a list of parameters. Our tracing interpreter is a procedure `T` that turns each syntactic program construct into a value or trace variable. It produces a trace, which takes this form:

```
t = s t | END
s = tv <- T-Score p tv [tv] | tv <- T-PrimOp op [tv]
```

`T-PrimOp` represents a primitive operation. `T-Score` takes a scoring function, the value of a random choice, and parameters as input, returning the value and incrementing a global score variable as a side effect. Executing a trace computes a corresponding probability density.

We now describe how traces are produced from CHURCH programs. Our tracing interpreter, with a few exceptions, works just like a normal interpreter for a call-by-value functional language. See SICP 4.1 [27] for a canonical definition. We first run a pre-process that puts a unique label `lab` at each syntax element for computing ERP addresses [10]. Actual tracing starts at a structure-preserving ERP:

```
T(addr, env, N lab p smp [e]):
    v+ = next_trace_variable();
    i+ = next_trace_variable();
    x+ = [ T(addr, env, e_i), e_i <- [e] ];
    x- = [ trvals[v], v <- x+];
    this_addr = cons(lab, addr);
    if ERP_exists(this_addr) then
        trvals[v+] = ERP_val(this_addr);
        this_score = p(trvals[v+], x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, trvals[v
            +], p-);
    else
        v- = smp(x-);
        trvals[v+] = v-;
        this_score = p(v, x-);
        score = score + this_score;
        update(this_addr, p, smp, this_score, v-, p-);
    add_stmt(v+ = T-score p i+ x+)
    return v+;
```

This behaves identically to the inner loop body of Algorithm 3 in the paper [10] describing LIGHTWEIGHT-MH, except we also add a `T-score` statement to the trace and return a trace variable, not the sampled value. `next_trace_variable` produces a new unique symbol for use as a trace variable. `i+` is an free *input variable* which will be set by the MCMC kernel at each proposal. The result is a direct correspondence between structure-preserving choices and input variables. `[ f(x) , x <- l]` depicts a mapping operation over elements `x` in the list `l`, producing a new list with the transformed elements. The function `add_stmt` appends its argument, a trace statement, to the global trace. `trvals` associates trace variables with the normally interpreted values (if a normal value is input, it acts as the identity function).

For structural ERP's, T works the same way except no tracing is done and all parameters are normally interpreted. This is key to our tracing technique: it has the effect of *partially evaluating* (pre-computing) away the computation associated with $P(\mathbf{S})$ and all naming computation.

The interpreter than resumes, working (mostly) just like a normal interpreter for a call-by-value language, treating the trace variable as another kind of value. In fact, the interpretation of letrec bindings, non-primitive functions lambda and calls app is the same as that of a normal interpreter for CHURCH. For branching and primitive operations, it is necessary to deal with trace variables and their associated values explicitly. Branches are handled as follows.

```
T(addr, env, If e1 e2 e3):
    v1 = T(addr, env, e1);
    if trvals[v1] then
        return T(addr, env, e2);
    else
        return T(addr, env, e3);
```

At this point, we are already in the structure-preserving part $P(\mathbf{N})$ of the distribution. This allows us to assume that if the conditional part of the branch e1 is a trace variable, the branch resulting from using its actual value trvals[v1] is the one always taken.

For primitive operations, we trace if the arguments contain trace variables and evaluate normally if not. This is because it is not necessary to trace things like X3 = 2 + 2; it is faster to use the computed result somewhere else, e.g. X6 = 4 + X5. This is known as *constant folding* in the compilers literature. Let any_trace_var? be a function that checks whether or not a trace variable occurs in a list of values.

```
T(addr, env, op [e]):
    vs = [T(addr, env, e_i), e_i <- [e]];
    if any_trace_var?(vs) then
        v+ = next_trace_variable();
        trvals[v+] = op([trvals[v_i], v_i <- vs]);
        add_stmt(v+ = T-PrimOp op vs);
        return v+;
    else
        return op(vs);
```

Our final step is to translate the traces to a low-level language. The generated traces do not contain any control flow, looping, or recursion. Additionally, they only assign a value to each variable *once*. Such programs are said to be in *static single assignment* (SSA) [11] form. SSA programs are easily translated into a number of low-level languages.

We generate a C++ function that calculates the subset of $P(\mathbf{N})$. Its arguments are the input variables corresponding to each structure-preserving choice, and its body is the C++-translated version of each trace statement specialized to the types used at each point. This is then used in the MCMC loop (Figure 3) as a way

to compute the density. Note that it is also possible to compile to other low-level languages such as LLVM [12] and Terra [13], which can more directly map to machine instructions on common hardware.

3.2 Stochastic memoization

Although it is not necessary for our tracing technique, models with Dirichlet processes [14] are included in the benchmarks used in this paper. We elected to represent them using the CHURCH-specific syntax constructs: DPmem alpha f and mem f [10], the implementation of which is briefly described here.

DPmem takes a concentration parameter and a function representing the base distribution, producing a function whose distribution is a Dirichlet process with the given parameters. mem takes a function, transforming it to only run once and have one return value for each unique incoming argument. We deal with these primitives in a manner that avoids control flow in our intermediate trace language through tagging the sub-traces produced by these primitives with a stack of names of the DPmem/mem-transformed functions currently generating them. These sub-traces are then compiled to C++ traces that conditionally execute based on the corresponding stochastic memoization semantics.

3.3 Slicing

We adapt *slicing*, a technique from programming languages [15], to generate computations that do the least amount of work in accept/rejecting proposals to structure-preserving choices. We define the *slice* $S(v_i)$ as all trace variables (i.e., statements) needing recomputation upon change to v_i.

First, for a given trace variable v_i, v_i may appear on the right hand side of the statements of other trace variables v_j: v_j = T-PrimOp op ... v_i .. or v_j = T-Score op v ... v_i ... In this case, we define v_j as *directly dependent* on v_i. We use $D(v_i, v_j)$ to denote the set of all such pairs of trace variables.

We now define *indirect dependency*. For the T-PrimOp above, it may be necessary to further compute the direct dependencies $D(v_j, v_k)$ of the LHS variable, and so on. For T-score statements it is not, as they merely return the value of some other input variable. Let $D_p(x, y) = \{(x, y) | y = \text{T-PrimOp op vs}, x \in vs\}$ capture this fact. The *indirect dependence relation* $I(\cdot, \cdot)$ is inductively defined as

$$I(x, y) = D(x, y)$$
$$\cup \{(x, z) | (x, z) \in D_p(x, z) \wedge (z, y) \in I(z, y)\}.$$

Then, the slice $S(v_i) = \{v_i\} \cup \{v_j | v_j \in I(v_i, v_j)\}$.

In code generation, we associate a function `S_vi` for each trace variable's slice $S(v_i)$ that executes its corresponding statements in order. We then promote all trace variables to global scope so that they are visible from every slice. We also avoid redundantly generating code. For example, if we already computed $S(v_4)$, and $v_4 \in S(v_2)$, we replace the resulting set of statements with a call to `S_v4`. Computing the density relevant to each structure-preserving choice x_k then amounts to setting the input variable `ik` to x_k and running the corresponding slice `S_ik`.

Our algorithm for computing $S(v_i)$ is the memoized depth-first search suggested by the definition of $I(x, y)$ above and our avoidance of generating redundant code. It produces a directed graph where each node represents a trace variable and each edge a direct dependency. The set of descendants of a node is the slice.

4 Results

We endeavor to answer the following questions:

1. What is the effect of tracing and slicing on kernel efficiency?

2. What is the cost of tracing and slicing?

3. In an absolute sense, how fast are the resulting kernels?

We analyzed a variety of widely-used models in machine learning, first running the following closed-universe models:

`linear-regression`: This performs linear regression on a set of 100 (x, y) data points using Gaussian priors on the slope and intercept of the unknown line. There is nearly no independent structure.

`hierarchical-regression`: This is an adaptation of the Rats example model in Volume I of the OpenBUGS examples repository [16]. 30 rats are modeled, each with five data points describing weight per week. There is much independent structure; the estimated slope and intercept of a particular rat can be updated independently of the others.

`mixture`: A Dirichlet-multinomial mixture model classifying four objects into three categories. This is a small model.

`infinite-mixture`: Same as the mixture model above, except for a Dirichlet process (DP) prior on the set of categories.

`topic-lda`: A topic model with two topics, 21 documents and a vocabulary size of 4.

`topic-hdp`: Same as the topic model above, but there is a DP prior on topics [17].

`citation-matching`: A citation matching model with a DP prior on papers. It is run on 5 citations. There are pairwise citation-paper similarity factors and paper-paper dissimilarity factors.

We begin our analysis with closed-universe models because they yield just one trace, directly representing the efficiency of the generated kernels in iterations per second. No extra time is spent in compiling and switching between multiple traces, as would be the case in open-universe models. We include a separate analysis of open-universe models later that factors in compilation time.

We ran the benchmarks with four different MH implementations: CHURCH's LIGHTWEIGHT-MH, our tracing interpreter, SHRED (tracing and slicing), and hand-coding. The OS/hardware used was Mac OS X 10.9 running on a 2.3 GHz Intel Core i7 with 8 GB RAM. Ikarus, a native-code Scheme compiler, was used to implement LIGHTWEIGHT-MH and our tracing/slicing algorithms. Hand-coded models were programmed in C++. For probabilistic language implementations, we ran each model for 10^5 iterations. Hand-coded models were run for 10^6 iterations each. The supplemental materials include specifications of the benchmarks and source code for hand-coded models.

Effect of tracing. Figure 4(a) compares our tracing interpreter with CHURCH and hand-coding. We see that tracing results in an average speedup of 32x, well over an order of magnitude. `hierarchical-regression` is sped up by 68x.

However, `mixture` is sped up by only 17x. We attribute this to the tracing interpreter primarily removing overhead of the addressing scheme. `hierarchical-regression` contains many more random variables that go through a much longer loop than `mixture`, resulting in more addressing overhead. We also observe that larger models such as `topic-lda` achieve fewer iterations per second than the simpler models. This is a consequence of the amount of computation needed per iteration.

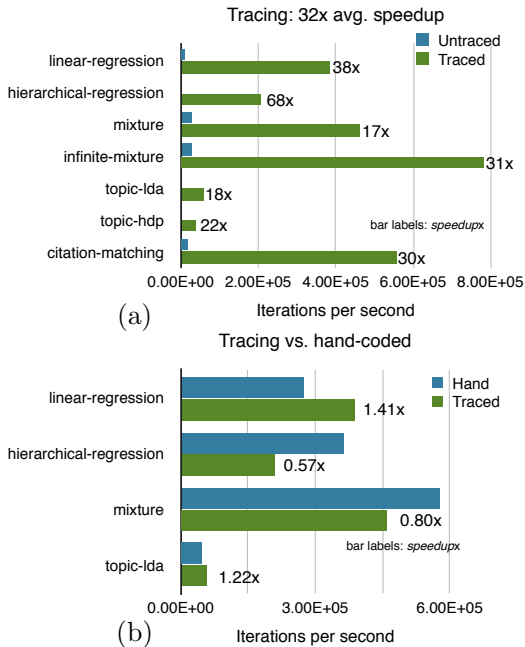Figure 4(b) shows performance relative to hand-coding. We see that the code we generate using tracing

Figure 4: MCMC kernel performance (in iterations/second) of tracing versus (a) CHURCH running LIGHTWEIGHT-MH and (b) hand-coding.
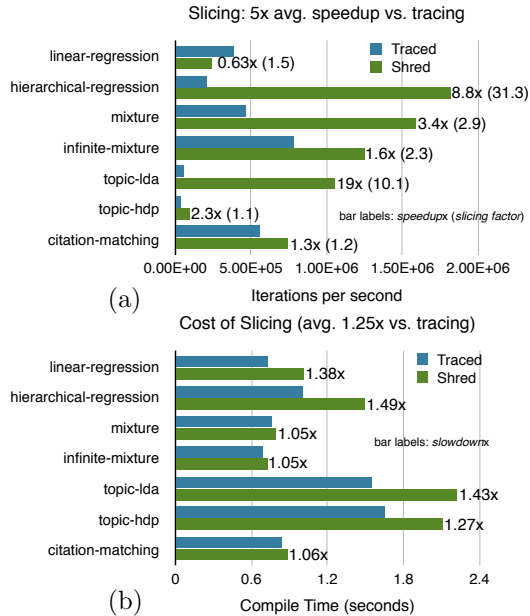

Figure 5: Effect of SHRED, our slicing-based incrementalizing scheme. (a) compares performance versus tracing and (b) compares cost in compilation time.

is on par with that of hand-coding, being close to the same speed on average.

Effect of slicing. First, in analyzing performance of our incrementalizing compiler, it is illustrative to quantify the *expected* speedup in terms of the independent structure available in the model: the *slicing factor*. It is the trace length divided by the average length of a slice (averaged over all variables). A slicing factor of 1 means MH iterations using the full trace take just as long as iterations using the slices, on average, and no speedup is expected. A slicing factor of 10 means that the trace is 10 times longer than the average slice, and therefore using the slices to be around 10 times faster.

Figure 5(a) shows speedups of slicing versus tracing, along with slicing factors. In most cases, speedup is on the order of the slicing factor. Slicing benefits models with lots of independent structure such as hierarchical-regression and topic-lda the most, with speedups of 8.8x and 19x respectively. We attribute the decreased performance of linear-regression to its lack of independent structure and running slices more than once per iteration (see Figure 3) to ensure consistency.

Cost of slicing. In Figure 5(b), we see that slicing's contribution to compile time is always within a factor of two. When considering the speedup obtained by

slicing, this is well worth it. For instance, the speedup for lda-topic means that a trade of 43% longer compile time results a 1200% faster kernel.

Overall, our technique generates MCMC kernels that are dramatically faster than the unoptimized LIGHTWEIGHT-MH. This is especially true when models exhibit lots of independent structure, allowing slicing to isolate smaller computations: In the hierarchical-regression and topic-lda benchmarks, we achieve speedups of 598x and 342x relative to the untraced version.

Open-universe models. We now evaluate our technique's overhead in generating and switching between traces. We ran two open-universe models: 1) model selection, choosing between the sum versus the product of two Gaussian variables, and 2) polynomial regression for polynomial degrees 1 through 4 on 9 (x, y) data points. We compared SHRED against hand-coding and LIGHTWEIGHT-MH("lwmh"), evaluating both iterations per second and total runtime.

Iter/s, Runtime (s)	lwmh	Shred	Hand
model selection	8.5e4	1.1e6	8.8e6
	12	2.1	0.11
polynomial regression	5.2e4	7.5e5	2.7e6
	19	3.8	0.37

We see that although our technique results in a clear speedup of the kernel by around an order of magnitude, hand-coded versions of the open-universe models are still much faster, by around an order of magnitude. We attribute this to the need to repeatedly save and load traces whenever a structural change occurs.

5   Related Work

Probabilistic programming languages.   Our system provides a much more efficient implementation of Metropolis-Hastings-based queries for the CHURCH probabilistic programming language [6]. Other inference algorithms for CHURCH can be better for some classes of programs [10, 18]. Our techniques could likely be used to accelerate these other CHURCH algorithms, and also algorithms for a variety of other universal probabilistic programming languages (such as HANSEI [7] and FIGARO [19]).

Other probabilistic programming languages take a different approach, using a simpler, non-Turing-complete language specialized for certain kinds of probabilistic computations. BUGS and JAGS [20, 1] focus on simpler probabilistic models with fixed control flow. Our techniques could apply to these languages as well.

Just-in-time compilation. Our technique has some aspects in common with Just-in-time (JIT) compilation, where we opportunistically replace segments of a running program with optimized versions of it to improve performance. JIT has a long history [21], with trace-based JIT compilation receiving renewed interest. Much of the work focuses on applying such techniques to JavaScript, a widely used language for specifying client-side scripts on web pages. In particular, by specializing the code in loops to the types actually used in them at runtime, speedups of an order of magnitude are possible [22].

Incremental computation.   The discipline of *incremental computation* aims at minimizing computation in the situation of a function receiving a series of changing inputs. We do not address incremental computation in the same generality as those who have worked on self-adjusting computation [23]. Rather, we focus incrementalizing only our traces, which do not contain control flow and consist of a static sequence of function calls. We use a program slicing-like [15] method to compute the set of statements affected by a proposal to a random choice.

6   Discussion and Future Work

We have shown that compilation techniques—tracing and slicing—can generate MCMC kernels whose performance is on par with hand-written code. While the initial compilation overhead, which is sometimes in seconds, is not justified for simple models that are only run once, it is very useful in even somewhat complex models. Nevertheless, improving compilation efficiency is a viable avenue of future work. Adapting further techniques from the compilers literature such as hot path detection and trace trees [14] would allow online detection of such complexity, adaptively selecting paths to compile based on execution frequency. For MCMC, this would mean only tracing paths where MCMC will spend the most iterations. In addition, for some programs it is not necessary to generate the entire trace to determine the form of the slices. For example, for an Ising line model on a thousand sites, there are only three different forms the slice can take, not a thousand.

Our technique also motivates further work that simply use the optimized code. Two such ways include compiling ahead of time for an unknown data set and running several copies of the same compiled code in parallel, which both further amortize compilation time. We are exploring the generation of probabilistic hardware, which has extreme "compilation overhead" but lends itself well to these use cases. Moreover, there are inference algorithms such as locally-annealed reversible jump [25] that rely on running on a fixed set of variables for many iterations.

The original LIGHTWEIGHT-MH paper [10] highlighted the possibility of using general code transformations to improve performance of probabilistic inference. We have largely succeeded, and we attribute this to the fact that the model representation is programmatic, allowing application and adaptation of general compilation techniques. Our technique could be the first of many ways in which such general techniques can be adapted to perform efficient inference.

7 References

[1] Just Another Gibbs Sampler. http://mcmc-jags.sourceforge.net.

[2] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic Models with Unknown Objects. In proceedings of IJCAI 2005, pp. 1352–1359.

[3] Andrew McCallum, Karl Schultz, Sameer Singh. "FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs". In Advances on Neural Information Processing Systems (NIPS), 2009.

[4] C. E. Freer and D. M. Roy. Posterior Distributions are Computable from Predictive Distributions. In proceedings of AISTATS 2011, pp. 233–240.

[5] N. Ramsey and A. Pfeffer. Stochastic Lambda Calculus and Monads of Probability Distributions. In proceedings of POPL 2002, Vol. 37 No. 1, pp. 154–165.

[6] Goodman, Noah. Mansinghka, Vikash K. Roy, Daniel M. Bonawitz, Keith. Tenenbaum, Joshua B. Church: a Language for Generative Models. In proceedings of UAI 2008, pp. 220–229.

[7] O. Kiselyov and C. Shan. Embedded Probabilistic Programming. IFIP working conference on domain-specific languages. Walid Taha, editor. LNCS 5658, Springer, 2009, pp. 360–384.

[8] A. Pfeffer. IBAL: a probabilistic rational programming language. In proceedings of IJCAI 2001, Vol. 1, pp. 733–740.

[9] S. Bhat, J. Borgstroem, A. D. Gordon, C. Russo. Deriving Probability Density Functions from Functional Programs. In proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp 510–524. 2013.

[10] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In proceedings of AISTATS 2011, pp. 1–9.

[11] A. Aho, M. S. Lam, R. Seth, and J. D. Ullman. Compilers: Principles, Techniques, and Tools (2nd edition). Prentice Hall, 2006.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation.

[13] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan and J. Vitek. Terra: A Multi-Stage Language for High-Performance Computing. PLDI 2013.

[14] Y. W. Teh. Dirichlet processes. Encyclopedia of Machine Learning 2010.

[15] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, Vol SE-10 No. 4, July 1984.

[16] D. Spiegelhalter, D., A. Thomass, N. Best and D. Lunn. OpenBUGS user manual, version 3.0.2. MRC Biostatistics Unit, Cambridge 2007.

[17] Y. W. Teh, M. I. Jordan, M. J. Beal and D. M Blei. Hierarchical Dirichlet Processes. Journal of the American Statistical Association, Vol. 101, Issue 476, 2006, pp. 1566–1581.

[18] A. Stuhlmüller and N. D. Goodman. A Dynamic Programming Algorithm for Inference in Recursive Probabilistic Programs. In proceedings of the Second Statistical Relational AI workshop at UAI 2012.

[19] A. Pfeffer. Figaro: An Object-Oriented Probabilistic Programming Language. Charles River Analytics Technical Report 2009.

[20] A. Thomas. BUGS: a Statistical Modeling Package. RTA/BCS Modular Languages Newsletter, Vol. 2, pp. 36–38. 1994.

[21] J. Aycock. A Brief History of Just-in-Time. ACM Computing Surveys 2003, Vol. 35, pp. 97–113.

[22] A. Gal, B. Eich, M. Shaver, D. Anderson, and D. Mandelin. Haghighat, Mohammad R. Kaplan, Blake. Hoare, Graydon. Zbarsky, Boris. Orendorff, Jason. Ruderman, Jesse. Smith, Edwin. Reitmaier, Rick. Bebenita, Michael. Chang, Mason. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In proceedings of PLDI 2009, pp 465–478.

[23] U. Acar. Self-Adjusting Computation. Ph. D thesis (2005).

[24] A. Gal, C. W. Probst, M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. VEE 2006.

[25] Y-T. Yeh, L. Yang, M. Watson, N. Goodman and P. Hanrahan. Synthesizing Open Worlds with Constraints using Locally Annealed Reversible Jump MCMC. In proceedings of SIGGRAPH 2012.