C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching

Daniel Ritchie
Stanford University

Andreas Stuhlmüller
Stanford University

Noah D. Goodman
Stanford University

Abstract

Lightweight, source-to-source transformation approaches to implementing MCMC for probabilistic programming languages are popular for their simplicity, support of existing deterministic code, and ability to execute on existing fast runtimes [1]. However, they are also inefficient, requiring a complete re-execution of the program on every Metropolis Hastings proposal. We present a new extension to the lightweight approach, C3, which enables efficient, incrementalized re-execution of MH proposals. C3 is based on two core ideas: transforming probabilistic programs into continuation passing style (CPS), and caching the results of function calls. It is particularly effective at speeding up recursive programs with many local latent variables. We show that on several common such models, C3 reduces proposal runtime by 20-100x, in some cases reducing runtime complexity from linear in model size to constant. We also demonstrate nearly an order of magnitude speedup on a complex inverse procedural modeling application.

1 Introduction

Probabilistic programming languages (PPLs) are a powerful, general-purpose tool for developing probabilistic models. A PPL is a programming language augmented with random sampling statements; programs written in a PPL correspond to generative priors. Performing inference on such programs amounts to reasoning about the space of execution traces which satisfy some condition on the program output. Many different PPL systems have been proposed, such as BLOG [2], Figaro [3], Church [4], Venture [5], Anglican [6], and Stan [7].

There are many possible implementations of PPL inference. One popular choice is the 'Lightweight MH' framework [1]. Lightweight MH uses a source-to-source transformation to turn a probablistic program into a deterministic one, where random choices are uniquely identified by their structural position in the program execution trace. Random choice values are then stored in a database indexed by these structural 'addresses.' To perform a Metropolis-Hastings proposal, Lightweight MH changes the value of a random choice and re-executes the program, looking up the values of other random choices in the database to reuse them when possible. Lightweight MH is simple to implement and allows PPLs to be built atop existing deterministic languages. Users can thus leverage existing libraries and fast compilers/runtimes for these 'host' languages. For example, Stochastic Matlab can access Matlab's rich matrix and image manipulation routines [1], WebPPL runs on Google's highly-optimized V8 Javascript engine [8], and Quicksand's host language compiles to fast machine code using LLVM [9].

Unfortunately, Lightweight MH is also inefficient: when an MH proposal changes a random choice, the entire program re-executes to propagate this change. This is rarely necessary: for many models, most proposals affect only a small subset of the program execution trace. To update the trace, re-execution is needed only where values can change. Under Lightweight MH, random choice values are preserved and reused when possible, limiting the effect of a proposal to a subset of the changed variable's Markov blanket (sometimes a much smaller subset, due to context-specific independence [10]). Custom PPL interpreters can leverage this property to incrementalize proposal re-execution [5], but implementing such interpreters is complicated, and using them makes it difficult or im-

Appearing in Proceedings of the 19^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2016, Cadiz, Spain. JMLR: W&CP volume 51. Copyright 2016 by the authors.

```
// Hidden Markov Model
var hmm = function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = hmm(n-1, obs);
    var state = transition(prev);
    observation(state, obs[n]);
    query.add(n, state);
    return state;
  }
};
hmm(100, observed_data);
return query;
```

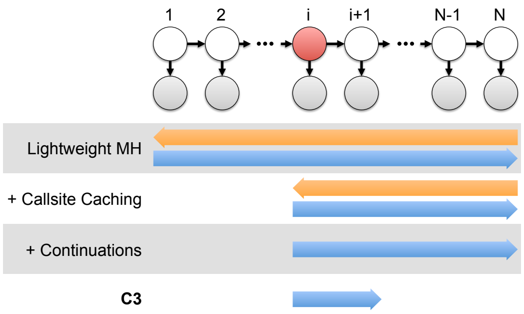


Figure 1: *(Left)* A simple HMM program in the WebPPL language; the hightlighted lines involving `query` are the only modifications necessary to use our method with this program. *(Right)* Illustrating the re-execution behavior of different MH implementations in response to a proposal to the random choice c_i shaded in red. Lightweight MH re-executes the entire `hmm` program, invoking (orange bar) and then unwinding (blue bar) the full chain of recursive calls. Callsite caching allows re-execution to skip all recursive calls under `hmm(i-1, obs)`. With continuations, re-execution only has to unwind from the continuation of choice c_i. Combining callsite caching and continuations allows re-execution to terminate upon returning from `hmm(i+1, obs)`, since its return value does not change.

possible to leverage libraries and fast runtimes for existing deterministic languages.

In this paper, we present a new implementation technique for MH proposals on probabilistic programs that gives the best of both worlds: incrementalized proposal execution using a lightweight, source-to-source transformation framework. It is especially successful at accelerating recursive probabilistic programs with many local latent variables. Our method, C3, is based on two core ideas:

1. *Continuations*: Converting the program into continuation-passing style to allow program re-execution to begin anywhere.

2. *Callsite caching*: Caching function calls to avoid re-execution when function inputs or ouputs have not changed.

We first describe how to implement C3 in any functional PPL with first-class functions. Our implementation is integrated into the open-source WebPPL probabilistic programming language [8]; it requires only small changes to how WebPPL programs are normally written. We then compare C3 to Lightweight MH, showing that it gives orders of magnitude speedups on common models such as HMMs, topic models, Gaussian mixtures, and hierarchical linear regression. In some cases, C3 reduces runtimes from linear in model size to constant. We also demonstrate that C3 is nearly an order of magnitude faster on a complex inverse procedural modeling example from computer graphics.

2 Approach

To illustrate our approach, we use a simple example: a binary state Hidden Markov Model program written in WebPPL (Figure 1 Left). This program recursively samples latent states (inside the `transition` function), conditioning on the observations in the `obs` list (inside the `observation` function). When invoked, `hmm(N, obs)` generates a linear chain of latent and observed random variables (Figure 1 Right). The values of the latent state variables are stored in the special `query` table; we will show later how this small modification allows our method to be used with this program.

Consider how Lightweight MH performs a proposal on this program. It first runs the program once to initialize the database of random choices. It then selects a choice c_i uniformly at random from this database (the red circle in Figure 1 Right) and changes its value. This change necessitates a constant-time update to the score of c_{i+1}. However, Lightweight MH re-executes the *entire* program, invoking a chain of recursive calls to `hmm` (the orange bar in Figure 1 Right) and then unwinding those calls (the blue bar). This process requires $2N$ such call visits for an HMM with N states.

One strategy for speeding up re-execution is to cache function calls and reuse their results if they are invoked again with unchanged inputs. We call this scheme, which is a generalization of Lightweight MH's random choice reuse policy, *callsite caching*. With this strategy, the recursive re-execution of `hmm` must still traverse all ancestors of choice c_i but can stop at `hmm(i, obs)`: it can reuse the result of `hmm(i-1, obs)`, since the inputs have not changed. As shown in Figure 1 Right, using

```
// Initial HMM code
var hmm = function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = hmm(n-1, obs);
    var state = transition(prev);
    observation(state, obs[n]);
    return state;
  }
};
```

```
// After caching transform
var hmm = function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = cache(hmm, n-1, obs);
    var state = cache(transition, prev);
    cache(observation, state, obs[n]);
    return state;
  }
};
```

```
// After function tagging transform
var hmm = tag(function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = cache(hmm, n-1, obs);
    var state = cache(transition, prev);
    cache(observation, state, obs[n]);
    return state;
  }
}, ’1’, [hmm, transition, observation]);
```

Figure 2: Source code transformations used by C3. *(Left)* Original HMM code. *(Middle)* Code after applying the caching transform, wrapping all callsites with the `cache` intrinsic. *(Right)* Code after applying the function tagging transform, where all functions are annotated with a lexically-unique ID and the values of their free variables. An example CPS-transformed program can be found in the supplemental materials.

callsite caching can result in less re-execution, but it still requires $\sim 2N$ `hmm` call visits on average.

Now suppose we instead convert the program into continuation passing style. CPS re-organizes a program to make all data and control flow explicit—instead of returning, functions invoke a ‘continuation’ function which represents the remaining computation to be performed [11]. For our HMM example, by storing the continuation at c_i, computation can resume from the point where this random choice is made, which corresponds to unwinding the stack from `hmm(i, obs)` up to `hmm(N, obs)`. Looking at the ‘Continuations’ row of Figure 1, this is a significant improvement over Lightweight MH and is also better than callsite caching. However, it still requires $\sim N$ call visits.

Our main insight is that we can achieve the desired runtime by combining callsite caching with continuations—we call the resulting system **C3**. With C3, re-execution can not only jump directly to choice c_i by invoking its continuation, but it can actually terminate almost immediately: the cache also contains the return values of all function calls, and since the return value of `hmm(i+1, obs)` has not changed, all subsequent computation will not change either. C3 unwinds only two recursive `hmm` calls, giving the desired constant-time update. Despite this early termination, the values of all hidden states are still available in the special `query` table (see Section 4.3). Thus C3 is more than the sum of its parts: by combining caching with CPS, it enables incrementalization benefits that neither component can deliver independently.

In the sections that follow, we describe how to implement C3 in a functional PPL. Specifically, we describe how to transform the program source at compile-time (Section 3) to make requisite data available to the runtime caching mechanism (Section 4).

3 Compile-time Source Transformations

Lightweight MH transforms the source code of probabilistic programs to compute random choice addresses; the transformed code can then be executed on existing runtimes for the host deterministic language. C3 fits into this framework by adding three additonal source transformations: caching, function tagging, and a standard continuation passing style transform for functional languages.

Caching This transform wraps every function callsite with a call to an intrinsic `cache` function (Figure 2 Middle). This function performs run-time callsite cache lookups, as described in Section 4. We initially left this step to the user, requiring them to hand-annotate calls that should be cached. However, we found that with a simple automatic cache adaptation scheme, the automatic transformation can achieve performance close to that of the optimal hand-annotation without additional user overhead (see Section 4.4)

Function tagging This transform analyzes the body of each function and tags the function with both a lexically-unique ID as well as the values of its free variables (Figure 2 Right). In Section 4, we describe how C3 uses this information to decide whether a function call must be re-executed.

The final source transformation pipeline is: caching → function tagging → address computation → CPS. Standard compiler optimizations such as inlining, constant folding, and common subexpression elimination can then be applied. In fact, the host language compiler often already performs such optimizations, which is an additional benefit of the lightweight transformational approach.

```
1   // Arguments added by compiler:
2   // a: current address
3   // k: current continuation
4   function cache(a, k, fn, args) {
5     // Global function call stack
6     var currNode = nodeStack.top();
7     var node = find(a, currNode.children);
8     if (node === null) {
9       node = FunctionNode(a);
10      // Insert maintains execution order
11      insert(node, currNode.children,
12             currNode.nextChildIndex);
13    }
14    execute(node, k, fn, args);
15  }
16
17  // rc: a random choice node
18  function propagate(rc) {
19    // Restore call stack up to rc.parent
20    restore(nodeStack, rc.parent);
21    // Changes to rc may make siblings unreachable
22    markUnreachable(rc.parent.children, rc.index);
23    // Continue executing
24    rc.parent.nextChildIndex = rc.index + 1;
25    rc.k(rc.val);
26  }
27
28  function execute(node, k, fn, args) {
29    node.reachable = true; node.k = k;
30    node.index = node.parent.nextChildIndex;
31    // Check for input changes
32    if (!fnEquiv(node.fn, fn) || !equal(node.args, args)) {
33      this.fn = fn; this.args = args;
34      // Mark all children as initially unreachable
35      markUnreachable(this.children, 0);
36      // Call fn with special continuation
37      node.nextChildIndex = 0;
38      nodeStack.push(node);
39      node.entered = true;
40      fn(args, function(retval) {
41        node = nodeStack.pop();
42        // Remove unreachable children
43        removeUnreachables(node.children);
44        // Terminate early on proposals where
45        //    retval does not change
46        var rveq = equal(retval, this.retval);
47        if (!node.entered && rveq) kexit();
48        else {
49          node.entered = false;
50          // retval change may make siblings unreachable
51          if (!rveq)
52            markUnreachable(node.parent.children,
53                            node.index);
54          // Continue executing
55          node.retval = retval;
56          node.parent.nextChildIndex++;
57          k(node.retval);
58        }
59      });
60    } else {
61      node.parent.nextChildIndex++;
62      k(node.retval);
63    }
64  }
```

Figure 3: The main subroutines governing C3's callsite cache. Function calls are wrapped with `cache`, which retrieves (or creates) a cache node for a given address `a`. It calls `execute`, which examines the function call's inputs for changes and runs the call if needed. Finally, MH proposals use `propagate` to resume re-execution of the program from a particular random choice node which has been changed.

4 Runtime Caching Implementation

When performing an MH proposal, callsite caching aims to avoid re-executing functions and to enable early termination from them as often as possible. In this section, we describe how C3 efficiently implements both of these types of computational 'short-circuiting' for probabilistic functional programs. Figure 3 provides high-level code for the main subroutines which govern the caching system.

4.1 Cache Representation

We first require an efficient cache structure to minimize overhead introduced by performing a cache access on every function call. C3 uses a tree-structured cache: it stores one node for each function call, where a node's children correspond to the function's callees. Random choices are stored as leaf nodes. Thus, the size of the cache is proprtional to the runtime of one complete execution of the program.

C3 also maintains a stack of nodes which tracks the program's call stack (`nodeStack` in Figure 3). During cache lookups, the desired node, if it exists, must be a child of the node on the top of this stack. Exploiting this property accelerates lookups, which would otherwise proceed from the cache root. Altogether, this structure provides expected constant time lookups, additions, and deletions. In addition, by storing a node's children in execution order, C3 can efficiently determine when child nodes have become 'stale' (i.e. unreachable) due to control flow changes and should be removed. A child node is marked unreachable when its parent begins or resumes execution (`execute` line 35; `propagate` line 22) and marked reachable when it is executed (`execute` line 20). Any children left marked unreachable when the parent exits are removed from the cache (`execute` line 43).

4.2 Short-Circuit On Function Entry

As described in Section 3, every function call is wrapped in a call to `cache`, which retrieves (or creates) a cache node for the current address. C3 then evaluates whether the node's associated function call must be re-evaluated or if its previous return value can be re-used (the `execute` function). Reuse is possible when the following two criteria are satisfied:

1. The function's arguments are equivalent to those from the previous execution.

2. The *function itself* is equivalent to that from the previous execution.

The first criterion can be verified with conservative equality testing; C3 uses shallow value equality testing, though deeper equality tests could result in more reuse for structured argument types. Deep equality testing is more expensive, though this can be mitigated using data structure techniques such as hash consing [12] or compiler optimizations such as global value numbering [13].

The second criterion is necessary because C3 operates on languages with first-class functions, so the identity of the caller at a given callsite is a runtime variable. Checking whether the two functions are exactly equal (i.e. refer to the same closure) is too conservative, however. Instead, C3 leverages the information provided by the function tagging transform from Section 3: two functions are equivalent if they have the same lexical ID (i.e. came from the same source location) and if the values of their free variables are equal. C3 applies this check recursively to any function-valued free variables, and it also memoizes the result, as program execution traces often feature many applications of the same function. This scheme is especially critical to obtain reuse in programs that feature anonymous functions, as those manifest as different closures for each program execution.

4.3 Short-Circuit On Function Exit

When C3 re-executes the program after changing a random choice (using the `propagate` function), control may eventually return to a function call whose return value has not changed. In this case, since all subsequent computation will have the same result, C3 can terminate execution early by invoking the exit continuation `kexit`. During function exit, C3's `execute` function detects if control is returning from a proposal by checking if the call is exiting without having first been entered (line 47). This condition signals that the current re-execution originated at some descendant of the exiting call, i.e. a random choice node.

Early termination is complicated by inference queries whose size depends on model size: for example, the sequence of latent states in an HMM. In lightweight PPL implementations, inference typically computes the marginal distribution on program return values. Thus, a naïve HMM implementation would construct and return a list of latent states. However, this implementation makes early termination impossible, as the list must be recursively reconstructed after a change to any of its elements.

For these scenarios, C3 offers a solution in the form of a global `query` table to which the program can write values of interest. Critically, `query` has a *write-only* interface: since the program cannot read from `query`, a write to it cannot introduce side-effects in subsequent compuation, and thus the semantics of early termination are preserved. Programs that use `query` can then simply return it to infer the marginal distribution over its contents.

4.4 Automatic Cache Adaptation

It is not always optimal to cache *every* callsite: caching introduces overhead, and some function calls almost always change on each invocation. C3 detects such callsites and stops caching them in a heuristic process we call *adaptive caching*. A callsite is un-cached if, after at least N proposals, execution has reached it M times without resulting in either short-circuit-on-entry or short-circuit-on-exit. We use $N = 10, M = 50$ for the results presented in this paper. With these settings, we have observed modest reductions in both cache size (20–65%) and running time (10–45%) for the examples presented in this paper. These results are also close to those given by the optimal hand-annotation of `cache` statements. For example, on the LDA example presented in Section 5, the automatically-adapted program has runtime within 7% of the optimally hand-annotated program. A small, constant running time overhead remains for uncached callsites, as calling them still triggers a table lookup to determine their caching status. Future work could explore efficiently re-compiling the program to remove `cache` calls around such callsites.

4.5 Optimizations

C3 takes care to ensure that the amount of work it performs in response to a proposal is only proportional to the amount of the program execution trace affected by that proposal. First, it maintains references to all random choices in a hash table, which provides expected constant time additions, deletions, and random element lookups. This table allows C3 to perform uniform random proposal choice in constant time, rather than the linear time cost of scanning through the entire cache.

Second, proposals may be rejected, which necessitates copying the cache in case its prior state must be restored on rejection. C3 avoids copying the entire cache using a copy-on-write scheme with similar principles to transactional memory [14]: modifications to a cache node's properties are staged and only committed if the proposal is accepted. Thus, C3 only copies as much of the cache as is actually visited during proposal re-execution.

Finally, continuations which never return may overflow the call stack for long-running programs. Our implementation avoids this problem via a standard *tram-*

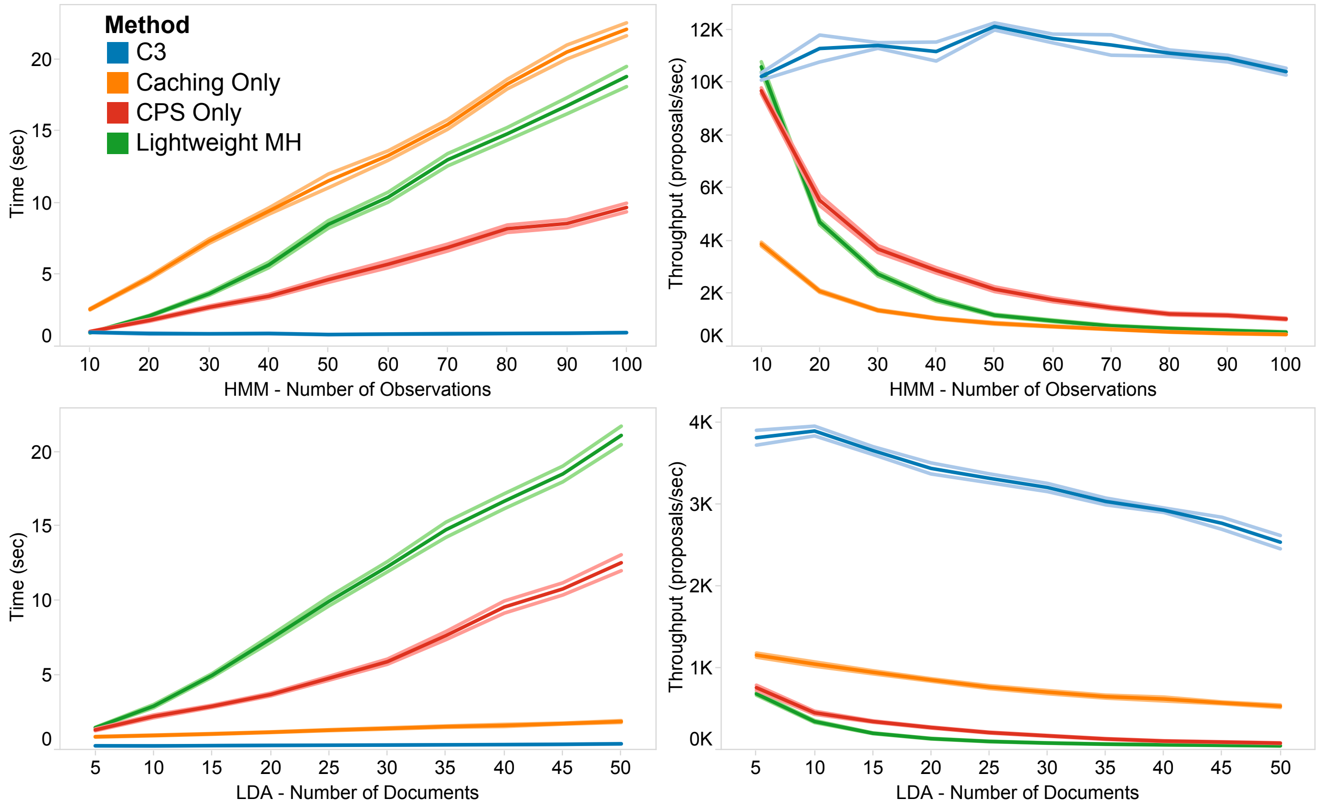


Figure 4: Comparing the performance of C3 with other MH implementations. *(Top)* Performing 10000 MH iterations on an HMM program. *(Bottom)* Performing 1000 MH iterations on an LDA program. *(Left)* Wall clock time elapsed, in seconds. *(Right)* Sampling throughput, in proposals per second. 95% confidence bounds are shown in a lighter shade. Only C3 exhibits constant asymptotic complexity for the HMM; other implementations take linear time, exhibiting decreasing throughput.

polining optimization: instead of directly invoking its continuation, a CPS'ed function returns a thunk (i.e. a nullary function) which encapsulates the continuation. The program repeatedly calls the series of returned thunks in a loop, thus executing the program with only one function call on the stack at any time.

5 Experimental Results

We now investigate the runtime performance characteristics of C3. We compare C3 to Lightweight MH, as well as to systems that use only callsite caching and only continuations. This allows us to investigate the incremental benefit provided by each of C3's components. The source code for all models used in this section is available in the supplemental materials, and our implementation of C3 itself is available as part of the WebPPL probabilistic programming language [8]. All timing data was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.10.2.

We first evaluate these systems on two standard generative models: a discrete-time Hidden Markov Model and a Latent Dirichlet Allocation model. We use synthetic data, since we are interested purely in the computational efficiency of different implementations of the same statistical inference algorithm. The HMM program uses 10 discrete latent states and 10 discrete observable states and returns the sequence of latent states. We condition it on a random sequence of observations, of increasing length from 10 to 100, and run each system for 10000 MH iterations, collecting a sample every 10 iterations. The LDA program uses 10 topics, a vocabulary of 100 words, and 20 words per document. It returns the distribution over words for each topic. We condition it on a set of random documents, increasing in size from 5 to 50, and run each system for 1000 MH iterations.

Figure 4 shows the results of this experiment; all quantities are averaged over 20 runs. We show wall clock time in seconds (left) and throughput in proposals per second (right). For the HMM, C3's runtime is constant regardless of model size, whereas *Lightweight MH* and *CPS Only* exhibit the expected linear runtime (approximately $2N$ and N, respectively). As dis-

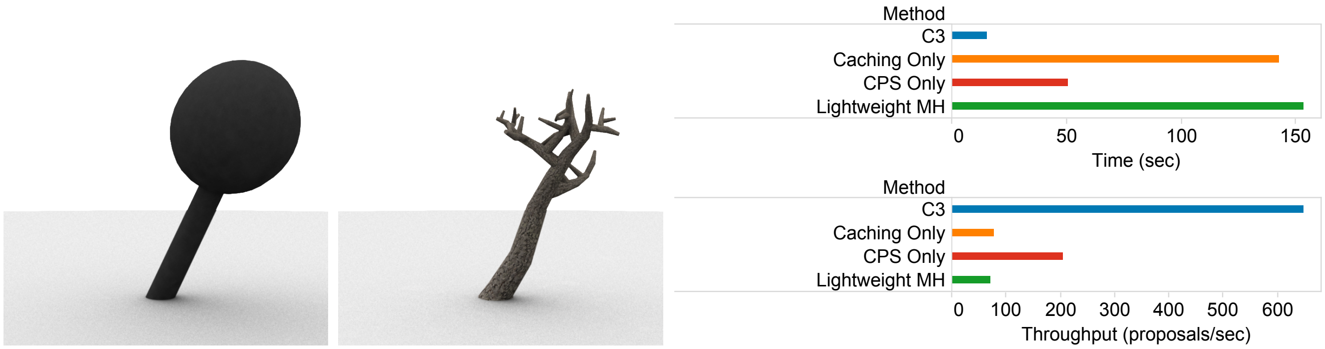


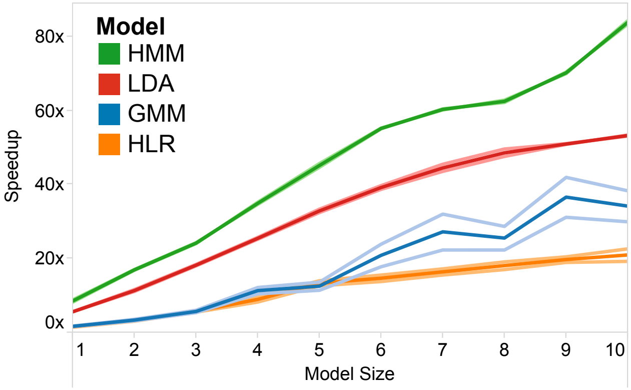
Figure 5: Comparing C3 and Lightweight MH on an inverse procedural modeling program. *(Left)* Desired tree shape. *(Middle)* Example output from inference over a tree program given the desired shape. *(Right)* Performance characteristics of different MH implementations. C3 delivers nearly an order of magnitude speedup.

cussed in Section 2, *Caching Only* has the same complexity as *Lightweight MH* but is a constant factor slower due to caching overhead. For the LDA model, *Lightweight MH* and *CPS Only* all exhibit asymptotic complexity comparable with their performance on the HMM. However, *Caching Only* performs significantly better. The LDA program is structured with nested loops; caching allows re-execution to skip entire inner loops for many proposals. *Caching Only* must still re-execute all ancestors of a changed random choice, though, so it is slower than C3, which jumps directly to the change point. C3 does not achieve exactly constant runtime for LDA because a small percentage of its proposals affect hierarchical variables, requiring more re-execution. This is a characteristic of hierarchical models in general; in this specific case, conjugacy could be leveraged to integrate out higher-level variables.

We also evaluate these systems on an inverse procedural modeling program. Procedural models are programs that generate random 3D models from the same family. *Inverse* procedural modeling infers executions of such a program that resemble a target output shape [15]. We use a simple grammar-like program for tree skeletons presented in prior work, conditioning its output to be volumetrically similar to a target shape [16]. We run each system for 2000 MH iterations.

Figure 5 shows the results of this experiment. C3 achieves the best performance, delivering nearly an order of magnitude speedup over Lightweight MH. Using caching only does not help in this example, since re-executing the program from its beginning reconstructs all of the recursive procedural modeling function's structured inputs, whose equality is not captured by our cache's shallow equality tests.

Finally, the figure below shows the results of a wider evaluation: for four models, we plot the speedup obtained by C3 over Lightweight MH (in relative throughput) as model size increases. The four models are: the HMM and LDA models from Figure 4, a one-dimensional finite Gaussian mixture model (GMM), and a hierarchical linear regression model (HLR) [17]. The 1-10 normalized Model Size parameter maps to a natural scale parameter for each of the four models; details are available in the supplemental materials. While C3 offers only small benefits over Lightweight MH for small models, it achieves dramatic speedups of 20-100x for large models.



6 Related Work

The ideas behind C3 have connections to other areas of active research. First, incrementalizing MCMC proposals for PPLs falls under the umbrella of *incremental computation* [18]. Much of the active work in this field seeks to build general-purpose languages and compilers to incrementalize any program [19]. However, there are also systems such as ours which seek simpler solutions to domain-specific incrementalization problems. In particular, C3's callsite caching mechanism was inspired in part by recent work in computer graphics on hierarchical render caches [20].[1]

C3 bears similarity to the CPS-based self-adjusting computation system developed by Ley-Wild and colleagues [21]. Both this system and C3 use continuations to approximate dynamic data dependencies, and both use some form of function caching to avoid re-executing unchanged computations. Their system aims for generality, using a compiler infrastructure that supports arbitrary changeable data and computation reuse. The restricted needs of our application—MH proposal computation—allow C3 to use a simpler strategy: only random choices are changeable, and computation is only reused from the previously-accepted program execution. This approach is consistent with our goal of keeping the system lightweight. It also has efficiency benefits: since Ley-Wild's system allows arbitrary changes to data, its change propagation mechanism must examine all of the previous execution's reads and writes to changeable data to ensure that they are consistent with the data's current value. In contrast, C3 knows that a proposal makes exactly one change (i.e. to a random choice value), so it can start change propagation from the continuation at that point, as well as terminate change propagation as soon as any function's return value is unchanged. This ability, along with efficient random choice lookup and cache copy-on-write, enables asymptotically constant-time proposals when the model's dependence structure supports them.

The Venture PPL features an algorithm to incrementally update a probabilistic execution trace in response to a random choice change [5]. Implemented as part of a custom interpreter, this method walks the trace starting from the changed node, identifying nodes which must be updated or removed, and determining when re-evaluation can stop. C3 performs a similar computation but uses continuations to traverse the execution trace rather than maintaining a complete interpreter state.

The Shred system also incrementalizes MH updates for PPLs [17]. Shred traces a program to remove its control flow and then uses data-flow analysis to produce incremental update procedures for each random choice. This process produces very fast proposal code, but it requires significant implementation cost, and its re-compilation overhead grows very large for programs with high control-flow variability, such as PCFGs. C3's caching scheme is a dynamic analog to Shred's static slicing which does not have compilation overhead but may not be as fast for models with fixed control flow.

The Swift compiler for the BLOG language is another recent system supporting incrementalized MCMC updates [22]. Unlike the above systems, BLOG/Swift uses a *possible-world semantics* for probabilistic programs, representing program state as a graphical model whose structure changes over time. Swift tracks the Markov Blanket of this model, computing incremental updates to it as model structure changes, allowing it to make efficient MCMC proposals. C3 does not explicitly compute Markov blankets, but its short-circuiting facilities limit re-execution to the subset of a changed variable's Markov blanket that is affected by the change.

7 Discussion and Future Work

This paper presented C3, a lightweight, source-to-source compilation system for incrementalizing MCMC updates in probabilistic programs. We have described how C3's two main components, continuations and callsite caching, allow it both to avoid re-executing function calls and to terminate re-execution early. Our experimental results show that C3 can provide orders-of-magnitude speedups over previous lightweight inference systems on typical generative models. It even enables constant-time updates in some cases where previous systems required linear time. We also demonstrate that C3 improves performance by nearly 10x on a complex, compute-heavy inverse procedural modeling problem. Our implementation of C3 is freely available as part of the open-source WebPPL probabilistic programming language.

C3 provides the most benefit for models where the number of latent variables grows with the dataset size. For models with a small number of global latent variables, C3 will not provide any speedup, and in fact the cache overhead may result in a small constant factor slowdown (though adaptation will remove almost all cache lookups in such cases). This is less a limitation of C3 and more an intrinsic expense of such models: any MH implementation will have to completely re-execute on each proposal.

For the Bayesian data analysis models we have focused on in this paper, much of C3's performance boost comes from its ability to terminate execution early, i.e. achieving constant-time updates for the HMM program. However, programs for which early termination is not possible can still see significant performance benefits. For example, when the program involves extensive recursive branching, as in the procedural tree program of Section 5, C3 can prune large sub-trees from the overall execution trace.

[1] An incomplete, undocumented version of C3's callsite caching mechanism also appears in the original MIT-Church implementation of the Church probabilistic programming language [4].

Careful optimization of computational efficiency, such as the work presented in this paper, is necessary for PPLs to move out of the domain of research and into production machine learning and AI systems. Along these lines, there are several directions for future work. First, static analysis might allow C3 to determine at compile time dependencies between random choices and subsequent function calls, obviating the need for some input equality checks and reducing caching overhead. Second, C3's CPS transform is overcomplete: it transforms the entire program, but C3 only need continuations at random choice points. Detecting and fusing blocks of purely deterministic code before applying the CPS transform could improve performance. Finally, while the results presented in this paper focus on single-site Metropolis Hastings, C3's core incrementalization scheme also applies to other sampling algorithms, such as Gibbs samplers or particle filter rejuvenation kernels [23].

Acknowledgments

The authors would like to thank Kimball Germane for his optimized implementation of CPS with trampolining. Zach Devito, Katherine Breeden, and Manolis Savva provided useful feedback on earlier drafts of this paper, as did the anonymous reviewers. This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS 2011*.

[2] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI 2005*.

[3] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[4] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI 2008*.

[5] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, 2014.

[6] F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *AISTATS 2014*.

[7] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*, 2014.

[8] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2015-5-18.

[9] Daniel Ritchie. Quicksand: A Lightweight Embedding of Probabilistic Programming for Procedural Modeling and Design. In *The 3rd NIPS Workshop on Probabilistic Programming*, 2014.

[10] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific Independence in Bayesian Networks. In *UAI 1996*.

[11] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[12] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical report, 1974.

[13] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *POPL 1988*.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA 1993*.

[15] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis Procedural Modeling. *ACM Trans. Graph.*, 30(2), 2011.

[16] Daniel Ritchie, Ben Mildenhall, Noah D. Goodman, and Pat Hanrahan. Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. In *SIGGRAPH 2015*.

[17] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating Efficient MCMC Kernels from Probabilistic Programs. In *AISTATS 2014*.

[18] G. Ramalingam and Thomas Reps. A Categorized Bibliography on Incremental Computation. In *POPL 1993*.

[19] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-Directed Automatic Incrementalization. In *PLDI 2012*.

[20] Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. Lazy Incremental Computation for Efficient Scene Graph Rendering. In *HPG 2013*.

[21] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling Self-adjusting Programs with Continuations. In *ICFP 2008*.

[22] Lei Li, Yi Wu, and Stuart J. Russell. SWIFT: Compiled Inference for Probabilistic Programs. Technical report, EECS Department, University of California, Berkeley, 2015.

[23] Walter R. Gilks and Carlo Berzuini. Following a moving target—Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1), 2001.