

# Selected Scholarly Work

JEAN-BAPTISTE TRISTAN

March 11, 2022

## Contents

<b>Contents</b>	<b>1</b>
<b>1 Formal Verification of Translation Validators</b>	<b>2</b>
<b>2 Verified validation of lazy code motion</b>	<b>13</b>
<b>3 A simple verified validator for software pipelining</b>	<b>24</b>
<b>4 Evaluating value-graph translation validation for LLVM</b>	<b>34</b>
<b>5 RockSalt: better faster stronger SFI for the x86</b>	<b>45</b>
<b>6 Augur: Data-Parallel Probabilistic Modeling</b>	<b>55</b>
<b>7 Efficient Training of LDA on a GPU by Mean-for-Mode Estimation</b>	<b>64</b>
<b>8 Exponential Stochastic Cellular Automata for Massively Parallel Inference</b>	<b>74</b>
<b>9 Compiling Markov chain Monte Carlo algorithms for probabilistic modeling</b>	<b>84</b>
<b>10 Adding Approximate Counters</b>	<b>99</b>
<b>11 Unlocking Fairness: a Trade-off Revisited</b>	<b>144</b>
<b>12 Gradient-Based Inference for Networks with Output Constraints</b>	<b>154</b>
<b>13 Using Butterfly-patterned Partial Sums to Draw from Discrete Distributions</b>	<b>165</b>
<b>14 Sketching for Latent Dirichlet-Categorical Models</b>	<b>195</b>
<b>15 Scaling Hierarchical Coreference with Homomorphic Compression</b>	<b>215</b>
<b>16 Online Post-Processing in Rankings for Fair Utility Maximization</b>	<b>234</b>
<b>17 Conjugate Energy-Based Models</b>	<b>243</b>
<b>18 Rate-Regularization and Generalization in Variational Autoencoders</b>	<b>255</b>
<b>19 A formal proof of PAC learnability for decision stumps</b>	<b>266</b>

<b>20 mad-GP: Automatic Differentiation of Gaussian Processes for Molecules and Materials</b>	<b>279</b>
<b>21 Geometry Meta-Optimization</b>	<b>312</b>
<b>22 Computable PAC Learning of Continuous Features</b>	<b>326</b>

# Formal Verification of Translation Validators

## A Case Study on Instruction Scheduling Optimizations

Jean-Baptiste Tristan

INRIA Paris-Rocquencourt

jean-baptiste.tristan@inria.fr

Xavier Leroy

INRIA Paris-Rocquencourt

xavier.leroy@inria.fr

### Abstract

Translation validation consists of transforming a program and *a posteriori* validating it in order to detect a modification of its semantics. This approach can be used in a verified compiler, provided that validation is formally proved to be correct. We present two such validators and their Coq proofs of correctness. The validators are designed for two instruction scheduling optimizations: list scheduling and trace scheduling.

**Categories and Subject Descriptors** F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Operational semantics; D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness proofs; D.3.4 [*Programming Languages*]: Processors - Optimization

**General Terms** Languages, Verification, Algorithms

**Keywords** Translation validation, scheduling optimizations, verified compilers, the Coq proof assistant

### 1. Introduction

Compilers, and especially optimizing compilers, are complex pieces of software that perform delicate code transformations and static analyses over the programs that they compile. Despite heavy testing, bugs in compilers (either in the algorithms used or in their concrete implementation) do happen and can cause incorrect object code to be generated from correct source programs. Such bugs are particularly difficult to track down because they are often misdiagnosed as errors in the source programs. Moreover, in the case of high-assurance software, compiler bugs can potentially invalidate the guarantees established by applying formal methods to the source code.

Translation validation, as introduced by Pnueli et al. (1998b), is a way to detect such compiler bugs at compile-time, therefore preventing incorrect code from being generated by the compiler silently. In this approach, at every run of the compiler or of one of the compiler passes, the input code and the generated code are fed to a validator (a piece of software distinct from the compiler itself), which tries to establish *a posteriori* that the generated code behaves

as prescribed by the input code. The validator can use a variety of techniques to do so, ranging from dataflow analyses (Huang et al. 2006) to symbolic execution (Necula 2000; Rival 2004) to the generation of a verification condition followed by model checking or automatic theorem proving (Pnueli et al. 1998b; Zuck et al. 2003). If the validator succeeds, compilation proceeds normally. If, however, the validator detects a discrepancy, or is unable to establish the desired semantic equivalence, compilation is aborted.

Since the validator can be developed independently from the compiler, and generally uses very different algorithms than those of the compiler, translation validation significantly increases the user's confidence in the compilation process. However, as unlikely as it may sound, it is possible that a compiler bug still goes unnoticed because of a matching bug in the validator. More pragmatically, translation validators, just like type checkers and bytecode verifiers, are difficult to test: while examples of correct code that should pass abound, building a comprehensive suite of incorrect code that should be rejected is delicate (Siregar and Bershad 1999). The guarantees obtained by translation validation are therefore weaker than those obtained by formal compiler verification: the approach where program proof techniques are applied to the compiler itself in order to prove, once and for all, that the generated code is semantically equivalent to the source code. (For background on compiler verification, see the survey by Dave (2003) and the recent mechanized verifications of compilers described by Klein and Nipkow (2006), Leroy (2006), Leinenbach et al. (2005) and Strecker (2005).)

A crucial observation that drives the work presented in this paper is that translation validation can provide formal correctness guarantees as strong as those obtained by compiler verification, provided the validator itself is formally verified. In other words, it suffices to model the validator as a function  $V : \text{Source} \times \text{Target} \rightarrow \text{boolean}$  and prove that  $V(S, T) = \text{true}$  implies the desired semantic equivalence result between the source code  $S$  and the compiled code  $T$ . The compiler or compiler pass itself does not need to be proved correct and can use algorithms, heuristics and implementation techniques that do not easily lend themselves to program proof. We claim that for many optimization passes, the approach outlined above — translation validation *a posteriori* combined with formal verification of the validator — can be significantly less involved than formal verification of the compilation pass, yet provide the same level of assurance.

In this paper, we investigate the usability of the “verified validator” approach in the case of two optimizations that schedule instructions to improve instruction-level parallelism: list scheduling and trace scheduling. We develop simple validation algorithms for these optimizations, based on symbolic execution of the original and transformed codes at the level of basic blocks (for list scheduling) and extended basic blocks after tail duplication (for trace scheduling). We then prove the correctness of these validators

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'08, January 7–12, 2008, San Francisco, California, USA.  
Copyright © 2008 ACM 978-1-59593-689-9/08/0001...\$5.00

against an operational semantics. The formalizations and proofs of correctness are entirely mechanized using the Coq proof assistant (Coq development team 1989–2007; Bertot and Castéran 2004). The formally verified instruction scheduling optimizations thus obtained integrate smoothly within the CompCert verified compiler described in (Leroy 2006; Leroy et al. 2003–2007).

The remainder of this paper is organized as follows. Section 2 recalls basic notions about symbolic evaluation and its uses for translation validation. Section 3 presents the Mach intermediate language over which scheduling and validation are performed. Sections 4 and 5 present the validators for list scheduling and trace scheduling, respectively, along with their proofs of correctness. Section 6 discusses our Coq mechanization of these results. Section 7 presents some experimental data and discusses algorithmic efficiency issues. Related work is discussed in section 8, followed by concluding remarks in section 9.

## 2. Translation validation by symbolic execution

### 2.1 Translation validation and compiler verification

We model a compiler or compiler pass as a function  $L_1 \rightarrow L_2 + \text{Error}$ , where the `Error` result denotes a compile-time failure,  $L_1$  is the source language and  $L_2$  is the target language for this pass. (In the case of instruction scheduling,  $L_1$  and  $L_2$  will be the same intermediate language, Mach, described in section 3.)

Let  $\leq$  be a relation between a program  $c_1 \in L_1$  and a program  $c_2 \in L_2$  that defines the desired semantic preservation property for the compiler pass. In this paper, we say that  $c_1 \leq c_2$  if, whenever  $c_1$  has well-defined semantics and terminates with observable result  $R$ ,  $c_2$  also has well-defined semantics, also terminates, and produces the same observable result  $R$ . We say that a compiler  $C : L_1 \rightarrow L_2 + \text{Error}$  is *formally verified* if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad C(c_1) = c_2 \Rightarrow c_1 \leq c_2 \quad (1)$$

In the translation validation approach, the compiler pass is complemented by a *validator*: a function  $L_1 \times L_2 \rightarrow \text{boolean}$ . A validator  $V$  is formally verified if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad V(c_1, c_2) = \text{true} \Rightarrow c_1 \leq c_2 \quad (2)$$

Let  $C$  be a compiler and  $V$  a validator. The following function  $C_V$  defines a compiler from  $L_1$  to  $L_2$ :

$$\begin{aligned} C_V(c_1) &= c_2 \text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \text{true} \\ C_V(c_1) &= \text{Error} \text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \text{false} \\ C_V(c_1) &= \text{Error} \text{ if } C(c_1) = \text{Error} \end{aligned}$$

The line of work presented in this paper follows from the trivial theorem below.

**Theorem 1.** *If the validator  $V$  is formally verified in the sense of (2), then the compiler  $C_V$  is formally verified in the sense of (1).*

In other terms, the verification effort for the derived compiler  $C_V$  reduces to the verification of the validator  $V$ . The original compiler  $C$  itself does not need to be verified and can be treated as a black box. This fact has several practical benefits. First, programs that we need to verify formally must be written in a programming language that is conducive to program proof. In the CompCert project, we used the functional subset of the specification language of the Coq theorem prover as our programming language. This makes it very easy to reason over programs, but severely constrains our programming style: program written in Coq must be purely functional (no imperative features) and be proved to terminate. In our verified validator approach, only the validator  $V$  is written in Coq. The compiler  $C$  can be written in any programming language, using updateable data structures and other imperative features if

necessary. There is also no need to ensure that  $C$  terminates. A second benefit of translation validation is that the base compiler  $C$  can use heuristics or probabilistic algorithms that are known to generate correct code with high probability, but not always. The rare instances where  $C$  generates wrong code will be caught by the validator. Finally, the same validator  $V$  can be used for several optimizations or variants of the same optimization. The effort of formally verifying  $V$  can therefore be amortized over several optimizations.

Given two programs  $c_1$  and  $c_2$ , it is in general undecidable whether  $c_1 \leq c_2$ . Therefore, the validator is in general incomplete: the reverse implication  $\Leftarrow$  in definition (2) does not hold, potentially causing false alarms (a correct code transformation is rejected at validation time). However, we can take advantage of our knowledge of the class of transformations performed by the compiler pass  $C$  to develop a specially-adapted validator  $V$  that is complete for these transformations. For instance, the validator of Huang et al. (2006) is claimed to be complete for register allocation and spilling. Likewise, the validators we present in this paper are specialized to the code transformations performed by list scheduling and trace scheduling, namely reordering of instructions within a basic block or an extended basic block, respectively.

### 2.2 Symbolic execution

Following Necula (2000), we use *symbolic execution* as our main tool to show semantic equivalence between code fragments. Symbolic execution of a basic block represents the values of variables at the end of the block as symbolic expressions involving the values of the variables at the beginning of the block. For instance, the symbolic execution of

$$\begin{aligned} z &:= x + y; \\ t &:= z \times y \end{aligned}$$

is the following mapping of variables to expressions

$$\begin{aligned} z &\mapsto x^0 + y^0 \\ t &\mapsto (x^0 + y^0) \times y^0 \\ v &\mapsto v^0 \text{ for all other variables } v \end{aligned}$$

where  $v^0$  symbolically denotes the initial value of variable  $v$  at the beginning of the block.

Symbolic execution extends to memory operations if we consider that they operate over an implicit argument and result, `Mem`, representing the current memory state. For instance, the symbolic execution of

$$\begin{aligned} \text{store}(x, 12); \\ y := \text{load}(x) \end{aligned}$$

is

$$\begin{aligned} \text{Mem} &\mapsto \text{store}(\text{Mem}^0, x^0, 12) \\ y &\mapsto \text{load}(\text{store}(\text{Mem}^0, x^0, 12), x^0) \\ v &\mapsto v^0 \text{ for all other variables } v \end{aligned}$$

The crucial observation is that two basic blocks that have the same symbolic evaluation (identical variables are mapped to identical symbolic expressions) are semantically equivalent, in the following sense: if both blocks successfully execute from an initial state  $\Sigma$ , leading to final states  $\Sigma_1$  and  $\Sigma_2$  respectively, then  $\Sigma_1 = \Sigma_2$ .

Necula (2000) goes further and compares the symbolic evaluations of the two code fragments modulo equations such as computation of arithmetic operations (e.g.  $1 + 2 = 3$ ), algebraic properties of these operations (e.g.  $x + y = y + x$  or  $x \times 4 = x \ll 2$ ), and “good variable” properties for memory accesses (e.g.

`load(store(m, p, v), p) = v`. This is necessary to validate transformations such as constant propagation or instruction strength reduction. However, for the instruction scheduling optimizations that we consider here, equations are not necessary and it suffices to compare symbolic expressions by structure.

The semantic equivalence result that we obtain between blocks having identical symbolic evaluations is too weak for our purposes: it does not guarantee that the transformed block executes without run-time errors whenever the original block does. Consider:

$$\begin{array}{ll} \mathbf{x} := 1 & \mathbf{x} := \mathbf{x} / 0 \\ & \mathbf{x} := 1 \end{array}$$

Both blocks have the same symbolic evaluation, namely  $\mathbf{x} \mapsto 1$  and  $v \mapsto v^0$  if  $v \neq \mathbf{x}$ . However, the rightmost block crashes at run-time on a division by 0, and is therefore not a valid optimization of the leftmost block, which does not crash. To address this issue, we enrich symbolic evaluation as follows: in addition to computing a mapping from variables to expressions representing the final state, we also maintain a set of all arithmetic operations and memory accesses performed within the block, represented along with their arguments as expressions. Such expressions, meaning “this computation is well defined”, are called *constraints* by lack of a better term. In the example above, the set of constraints is empty for the leftmost code, and equal to  $\{\mathbf{x}^0/0\}$  for the rightmost code.

To validate the transformation of a block  $b_1$  into a block  $b_2$ , we now do the following: perform symbolic evaluation over  $b_1$ , obtaining a mapping  $m_1$  and a set of constraints  $s_1$ ; do the same for  $b_2$ , obtaining  $m_2, s_2$ ; check that  $m_2 = m_1$  and  $s_2 \subseteq s_1$ . This will guarantee that  $b_2$  executes successfully whenever  $b_1$  does, and moreover the final states will be identical.

### 3. The Mach intermediate language

The language that we use to implement our scheduling transformations is the Mach intermediate language outlined in (Leroy 2006). This is the lowest-level intermediate language in the CompCert compilation chain, just before generation of PowerPC assembly code. At the Mach level, registers have been allocated, stack locations reserved for spilled temporaries, and most Mach instructions correspond exactly to single PowerPC instructions. This enables the scheduler to perform precise scheduling. On the other hand, the semantics of Mach is higher-level than that of a machine language, guaranteeing in particular that terminating function calls are guaranteed to return to the instruction following the call; this keeps proofs of semantic preservation more manageable than if they were conducted over PowerPC assembly code.

#### 3.1 Syntax

A Mach program is composed of a set of functions whose bodies are lists of instructions.

Mach instructions:

$i ::= \text{setstack}(r, \tau, \delta)$	register to stack move
$  \text{getstack}(\tau, \delta, r)$	stack to register move
$  \text{getparam}(\tau, \delta, r)$	caller's stack to reg. move
$  \text{op}(op, \vec{r}, r)$	arithmetic operation
$  \text{load}(chunk, mode, \vec{r}, r)$	memory load
$  \text{store}(chunk, mode, \vec{r}, r)$	memory store
$  \text{call}(r \mid id)$	function call
$  \text{label}(l)$	branch target label
$  \text{goto}(l)$	unconditional branch
$  \text{cond}(cond, \vec{r}, l_{true})$	conditional branch
$  \text{return}$	function return

Mach functions:

$$f ::= \text{fun } id \quad \{ \text{stack } n_1; \text{frame } n_2; \text{code } \vec{i} \}$$

Mach offers an assembly-level view of control flow, using labels and branches to labels. In conditional branches  $\text{cond}$ ,  $cond$  is the condition being tested and  $\vec{r}$  its arguments. Instructions are similar to those of the processor and include arithmetic and logical operations  $op$ , as well as memory load and stores; arguments and results of these instructions are processor registers  $r$ .  $op$  ranges over the set of operations of the processor and  $mode$  over the set of addressing modes. In memory accesses,  $chunk$  indicates the kind, size and signedness of the memory datum being accessed.

To access locations in activation records where temporaries are spilled and callee-save registers are saved, Mach offers specific `getstack` and `setstack` instructions, distinct from `load` and `store`. The location in the activation record is identified by a type  $\tau$  and a byte offset  $\delta$ . `getparam` reads from the activation record of the calling function, where excess parameters to the call are stored. These special instructions enable the Mach semantics to enforce useful separation properties between activation records and the rest of memory.

#### 3.2 Dynamic semantics

The operational semantics of Mach is given in a combination of small-step and big-step styles, as three mutually inductive predicates:

$$\begin{array}{ll} \Sigma \vdash \vec{i}, R, F, M \rightarrow \vec{i}', R', F', M' & \text{one instruction} \\ \Sigma \vdash \vec{i}, R, F, M \xrightarrow{*} \vec{i}', R', F', M' & \text{several instructions} \\ G \vdash f, P, R, M \Rightarrow R', M' & \text{function call} \end{array}$$

The first predicate defines one transition within the current function corresponding to the execution of the first instruction in the list  $\vec{i}$ .  $R$  maps registers to values,  $F$  maps activation record locations to values, and  $M$  is the current memory state. The context  $\Sigma = (G, f, sp, P)$  is a quadruple of the set  $G$  of global function and variable definitions,  $f$  the function being executed,  $sp$  the stack pointer, and  $P$  the activation record of the caller. The following excerpts should give the flavor of the semantics.

$$\begin{array}{c} v = \text{eval\_op}(op, R(\vec{r})) \\ \hline \Sigma \vdash op(op, \vec{r}, r_d) :: c, R, F, M \rightarrow c, R\{r_d \leftarrow v\}, F, M \\ \begin{array}{c} \text{true} = \text{eval\_condition}(cond, R(\vec{r})) \\ c' = \text{find\_label}(l_{true}, f.\text{code}) \end{array} \\ \hline (G, f, sp, P) \vdash \text{cond}(cond, \vec{r}, l_{true}) :: c, R, F, M \rightarrow c', R, F, M \\ \hline \begin{array}{c} G(R(r_f)) = f \quad G \vdash f, F, R, M \Rightarrow R', M' \\ (G, f, sp, P) \vdash \text{call}(r_f) :: c, R, F, M \rightarrow c, R', F, M' \\ \text{alloc}(M, 0, f.\text{stack}) = (sp, M_1) \\ \text{init\_frame}(f.\text{frame}) = F_1 \\ G, f, sp, P \vdash f.\text{code}, R, F_1, M_1 \xrightarrow{*} \text{return} :: c', R', F_2, M_2 \\ M' = \text{free}(M_2, sp) \end{array} \\ \hline G \vdash f, P, R, M \Rightarrow R', M' \end{array}$$

#### 4. Validation of list scheduling

List scheduling is the simplest instruction scheduling optimization. Like all optimizations of its kind, it reorders instructions in the program to increase instruction-level parallelism, by taking advantage

of pipelining and multiple functional units. In order to preserve program semantics, reorderings of instructions must respect the following rules, where  $\rho$  is a *resource* of the processor (e.g. a register, or the memory):

- Write-After-Read: a read from  $\rho$  must not be moved after a write to  $\rho$ ;
- Read-After-Write: a write to  $\rho$  must not be moved after a read from  $\rho$ ;
- Write-After-Write: a write to  $\rho$  must not be moved after another write to  $\rho$ .

We do not detail the implementation of list scheduling, which can be found in compiler textbooks (Appel 1998; Muchnick 1997). One important feature of this transformation is that it is performed at the level of basic blocks: instructions are reordered within basic blocks, but never moved across branch instructions nor across labels. Therefore, the control-flow graph of the original and scheduled codes are isomorphic, and translation validation for list scheduling can be performed by comparing matching blocks in the original and scheduled codes.

In the remainder of the paper we will use the term “block” to denote the longest sequence of non-branching instructions between two branching instructions. The branching instructions in Mach are `label`, `goto`, `cond` and `call`. This is a change from the common view where a block includes its terminating branching instruction.

#### 4.1 Symbolic expressions

As outlined in section 2.2, we will use symbolic execution to check that the scheduling of a Mach block preserves its semantics. The syntax of symbolic expressions that we use is as follows:

Resources:

$$\rho ::= r \mid \text{Mem} \mid \text{Frame}$$

Value expressions:

$$t ::= r^0 \quad \text{initial value of register } r \\ \mid \text{Getstack}(\tau, \delta, t_f) \\ \mid \text{Getparam}(\tau, \delta) \\ \mid \text{Op}(op, \vec{t}) \\ \mid \text{Load}(chunk, mode, \vec{t}, t_m)$$

Memory expressions:

$$t_m ::= \text{Mem}^0 \quad \text{initial memory store} \\ \mid \text{Store}(chunk, mode, \vec{t}, t_m, t)$$

Frame expressions:

$$t_f ::= \text{Frame}^0 \quad \text{initial frame} \\ \mid \text{Setstack}(t, \tau, \delta, t_f)$$

Symbolic code:

$$m ::= \rho \mapsto (t \mid t_m \mid t_f)$$

Constraints:

$$s ::= \{t, t_m, t_f, \dots\}$$

The resources we track are the processor registers (tracked individually), the memory state (tracked as a whole), and the *frame* for the current function (the part of its activation record that is treated as separate from the memory by the Mach semantics). The symbolic code  $m$  obtained by symbolic evaluation is represented as a map from resources to symbolic expressions  $t$ ,  $t_f$  and  $t_m$  of the appropriate kind. Additionally, as explained in section 2.2, we also collect a set  $s$  of symbolic expressions that have well-defined semantics.

We now give a denotational semantics to symbolic codes, as transformers over concrete states  $(R, F, M)$ . We define inductively

the following four predicates:

$\Sigma \vdash \llbracket t \rrbracket(R, F, M) = v$	Value expressions
$\Sigma \vdash \llbracket t_f \rrbracket(R, F, M) = F'$	Frame expressions
$\Sigma \vdash \llbracket t_m \rrbracket(R, F, M) = M'$	Memory expressions
$\Sigma \vdash \llbracket m \rrbracket(R, F, M) = (R', F', M')$	Symbolic codes

The definition of these predicates is straightforward. We show two selected rules.

$$\frac{v = \text{eval\_op}(op, \vec{v}) \quad \Sigma \vdash \llbracket \vec{t} \rrbracket(R, F, M) = \vec{v}}{\Sigma \vdash \llbracket \text{op}(op, \vec{t}) \rrbracket(R, F, M) = v}$$

$$\frac{\Sigma \vdash \llbracket m(r) \rrbracket(R, F, M) = R'(r) \quad \Sigma \vdash \llbracket m(\text{Frame}) \rrbracket(R, F, M) = F' \quad \Sigma \vdash \llbracket m(\text{Mem}) \rrbracket(R, F, M) = M'}{\Sigma \vdash \llbracket m \rrbracket(R, F, M) = (R', F', M')}$$

For constraints, we say that a symbolic expression  $t$  viewed as the constraint “ $t$  has well-defined semantics” is satisfied in a concrete state  $(R, F, M)$ , and we write  $\Sigma, (R, F, M) \models t$ , if there exists a value  $v$  such that  $\Sigma \vdash \llbracket t \rrbracket(R, F, M) = v$ , and similarly for symbolic expressions  $t_f$  and  $t_m$  over frames and memory. For a set of constraints  $s$ , we write  $\Sigma, (R, F, M) \models s$  if every constraint in  $s$  is satisfied in state  $(R, F, M)$ .

#### 4.2 Algorithm for symbolic evaluation

We now give the algorithm that, given a list  $\vec{i}$  of non-branching Mach instructions, computes its symbolic evaluation  $\alpha(\vec{i}) = (m, s)$ .

We first define the symbolic evaluation  $\alpha(i, (m, s))$  of one instruction  $i$  as a transformer from the pair  $(m, s)$  of symbolic code and constraint “before” the execution of  $i$  to the pair  $(m', s')$  “after” the execution of  $i$ .

$$\begin{aligned} \alpha(\text{update}(\rho, t, (m, s))) &= (m\{\rho \leftarrow t\}, s \cup \{t\}) \\ \alpha(\text{setstack}(r, \tau, \delta), (m, s)) &= \text{update}(\text{Frame}, \text{Setstack}(m(r), \tau, \delta, m(\text{Frame})), (m, s)) \\ \alpha(\text{getstack}(\tau, \delta, r), (m, s)) &= \text{update}(r, \text{Getstack}(\tau, \delta, m(\text{Frame})), (m, s)) \\ \alpha(\text{getparam}(\tau, \delta, r), (m, s)) &= \text{update}(r, \text{Getparam}(\tau, \delta), (m, s)) \\ \alpha(\text{load}(chunk, mode, \vec{t}, t_m), (m, s)) &= \text{update}(r, \text{Op}(op, m(\vec{t})), (m, s)) \\ \alpha(\text{store}(chunk, mode, \vec{t}, r), (m, s)) &= \text{update}(r, \text{Load}(chunk, mode, m(\vec{t}), m(\text{Mem})), (m, s)) \\ \alpha(\text{mem}, \text{store}(chunk, mode, m(\vec{r}), m(\text{Mem}), m(r)), (m, s)) &= \text{update}(\text{Mem}, \text{Store}(chunk, mode, m(\vec{r}), m(\text{Mem}), m(r)), (m, s)) \end{aligned}$$

We then define the symbolic evaluation of the block  $b = i_1; \dots; i_n$  by iterating the one-instruction symbolic evaluation function  $\alpha$ , starting with the initial symbolic code  $\varepsilon = (\rho \mapsto \rho^0)$  and the empty set of constraints.

$$\alpha(b) = \alpha(i_n, \dots, \alpha(i_2, \alpha(i_1, (\varepsilon, \emptyset)))) \dots$$

Note that all operations performed by the block are recorded in the constraint set  $s$ . It is possible to omit operations that cannot fail at run-time (such as “load constant” operators) from  $s$ ; we elected not to do so for simplicity.

The symbolic evaluation algorithm has the following two properties that are used later in the proof of correctness for the validator. First, any concrete execution of a block  $b$  satisfies its symbolic execution  $\alpha(b)$ , in the following sense.

**Lemma 1.** Let  $b$  be a block and  $c$  an instruction list starting with a branching instruction. If  $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R', F', M'$  and  $\alpha(b) = (m, s)$ , then  $\Sigma \vdash \llbracket m \rrbracket(R, F, M) = (R', F', M')$  and  $\Sigma, (R, F, M) \models s$ .

Second, if an initial state  $R, F, M$  satisfies the constraint part of  $\alpha(b)$ , it is possible to execute  $b$  to completion from this initial state.

**Lemma 2.** Let  $b$  be a block and  $c$  an instruction list starting with a branching instruction. Let  $\alpha(b) = (m, s)$ . If  $\Sigma, (R, F, M) \models s$ , then there exists  $R', F', M'$  such that  $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R', F', M'$ .

### 4.3 Validation at the level of blocks

Based on the symbolic evaluation algorithm above, we now define a validator for transformations over blocks. This is a function  $V_b$  taking two blocks (lists of non-branching instructions)  $b_1, b_2$  and returning `true` if  $b_2$  is a correct scheduling of  $b_1$ .

$$\begin{aligned} V_b(b_1, b_2) = \\ \text{let } (m_1, s_1) = \alpha(b_1) \\ \text{let } (m_2, s_2) = \alpha(b_2) \\ \text{return } m_2 = m_1 \wedge s_2 \subseteq s_1 \end{aligned}$$

The correctness of this validator follows from the properties of symbolic evaluation.

**Lemma 3.** Let  $b_1, b_2$  be two blocks and  $c_1, c_2$  two instruction sequences starting with branching instructions. If  $V_b(b_1, b_2) = \text{true}$  and  $\Sigma \vdash (b_1; c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$ , then  $\Sigma \vdash (b_2; c_2), R, F, M \xrightarrow{*} c_2, R', F', M'$ .

*Proof.* Let  $(m_1, s_1) = \alpha(b_1)$  and  $(m_2, s_2) = \alpha(b_2)$ . By hypothesis  $V_b(b_1, b_2) = \text{true}$ , we have  $m_2 = m_1$  and  $s_2 \subseteq s_1$ . By Lemma 1, the hypothesis  $\Sigma \vdash (b_1; c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$  implies that  $\Sigma, (R, F, M) \models s_1$ . Since  $s_2 \subseteq s_1$ , it follows that  $\Sigma, (R, F, M) \models s_2$ . Therefore, by Lemma 2, there exists  $R'', F'', M''$  such that  $\Sigma \vdash (b_2; c_2), R, F, M \xrightarrow{*} c_2, R'', F'', M''$ . Applying Lemma 1 to the evaluations of  $(b_1; c_1)$  and  $(b_2; c_2)$ , we obtain that  $\Sigma \vdash \llbracket m_1 \rrbracket(R, F, M) = (R', F', M')$  and  $\Sigma \vdash \llbracket m_2 \rrbracket(R, F, M) = (R'', F'', M'')$ . Since  $m_2 = m_1$  and the denotation of a symbolic code is unique if it exists, it follows that  $(R'', F'', M'') = (R', F', M')$ . The expected result follows.  $\square$

### 4.4 Validation at the level of function bodies

Given two lists of instructions  $c_1$  and  $c_2$  corresponding to the body of a function before and after instruction scheduling, the following validator  $V$  checks that  $V_b(b_1, b_2) = \text{true}$  for each pair of matching blocks  $b_1, b_2$ , and that matching branching instructions are equal. (We require, without significant loss of generality, that the external implementation of list scheduling preserves the order of basic blocks within the function code.)

$$\begin{aligned} V(c_1, c_2) = \\ \text{if } c_1 \text{ and } c_2 \text{ are empty:} \\ \quad \text{return } \text{true} \\ \text{if } c_1 \text{ and } c_2 \text{ start with a branching instruction:} \\ \quad \text{decompose } c_1 \text{ as } i_1 :: c'_1 \\ \quad \text{decompose } c_2 \text{ as } i_2 :: c'_2 \\ \quad \text{return } i_1 = i_2 \wedge V(c'_1, c'_2) \\ \text{if } c_1 \text{ and } c_2 \text{ start with a non-branching instruction:} \\ \quad \text{decompose } c_1 \text{ as } b_1; c'_1 \\ \quad \text{decompose } c_2 \text{ as } b_2; c'_2 \\ \quad \text{(where } b_1, b_2 \text{ are maximal blocks)} \\ \quad \text{return } V_b(b_1, b_2) \wedge V(c'_1, c'_2) \\ \text{otherwise:} \\ \quad \text{return } \text{false} \end{aligned}$$

To prove that  $V(c_1, c_2) = \text{true}$  implies a semantic preservation result between  $c_1$  and  $c_2$ , the natural approach is to reason by induction on an execution derivation for  $c_1$ . However, such an induction decomposes the execution of  $c_1$  into executions of individual instructions; this is a poor match for the structure of the validation function  $V$ , which decomposes  $c_1$  into maximal blocks joined by branching instructions. To bridge this gap, we define an alternate, block-oriented operational semantics for Mach that describes executions as sequences of sub-executions of blocks and of branching instructions. Writing  $\Sigma$  for global contexts and  $S, S'$  for quadruples  $(c, R, F, M)$ , the block-oriented semantics refines the  $\Sigma \vdash S \rightarrow S'$ ,  $\Sigma \vdash S \xrightarrow{*} S'$  and  $G \vdash f, P, R, M \Rightarrow R', M'$  predicates of the original semantics into the following 5 predicates:

$$\begin{array}{ll} \Sigma \vdash S \rightarrow_{nb} S' & \text{one non-branching instruction} \\ \Sigma \vdash S \xrightarrow{*_{nb}} S' & \text{several non-branching instructions} \\ \Sigma \vdash S \rightarrow_b S' & \text{one branching instruction} \\ \Sigma \vdash S \rightsquigarrow S' & \text{block-branch-block sequences} \\ G \vdash f, P, R, M \Rightarrow_{blocks} R', M' & \end{array}$$

The fourth predicate, written  $\rightsquigarrow$ , represents sequences of  $\xrightarrow{*_{nb}}$  transitions separated by  $\rightarrow_b$  transitions:

$$\begin{array}{c} \Sigma \vdash S \xrightarrow{*_{nb}} S' \\ \hline \Sigma \vdash S \rightsquigarrow S' \\[1ex] \Sigma \vdash S \xrightarrow{*_{nb}} S_1 \quad \Sigma \vdash S_1 \rightarrow_b S_2 \quad \Sigma \vdash S_2 \rightsquigarrow S' \\ \hline \Sigma \vdash S \rightsquigarrow S' \end{array}$$

It is easy to show that the  $\rightsquigarrow$  block-oriented semantics is equivalent to the original  $\xrightarrow{*}$  semantics for executions of whole functions.

**Lemma 4.**  $G \vdash f, P, R, M \Rightarrow R', M'$  if and only if  $G \vdash f, P, R, M \Rightarrow_{blocks} R', M'$ .

We are now in a position to state and prove the correctness of the validator  $V$ . Let  $p$  be a program and  $p'$  the corresponding program after list scheduling and validation:  $p'$  is identical to  $p$  except for function bodies, and  $V(p(id).code, p'(id).code) = \text{true}$  for all function names  $id \in p$ .

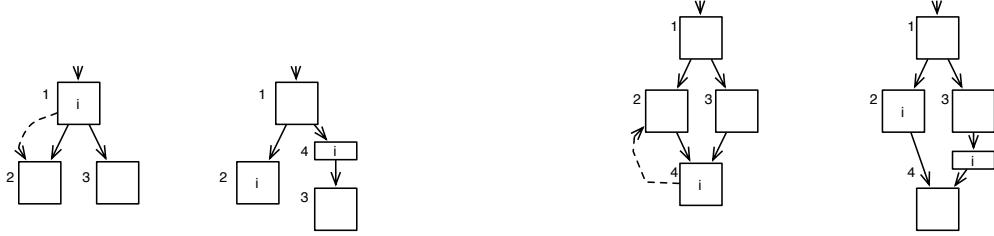
**Theorem 2.** Let  $G$  and  $G'$  be the global environments associated with  $p$  and  $p'$ , respectively. If  $G \vdash f, P, R, M \Rightarrow R', M'$  and  $V(f.code, f'.code) = \text{true}$ , then  $G' \vdash f', P, R, M \Rightarrow R', M'$ .

*Proof.* We show the analogous result using  $\Rightarrow_{blocks}$  instead of  $\Rightarrow$  in the premise and conclusion by induction over the evaluation derivation, using Lemma 3 to deal with execution of blocks. We conclude by Lemma 4.  $\square$

## 5. Validation of trace scheduling

Trace scheduling (Ellis 1986) is a generalization of list scheduling where instructions are allowed to move past a branch or before a join point, as long as this branch or joint point does not correspond to a back-edge. In this work we restrict the instructions that can be moved to non-branching instructions, thus considering a slightly weaker version of trace scheduling than the classical one.

Moving instructions to different basic blocks requires compensating code to be inserted in the control-flow graph, as depicted in figure 1. Consider an instruction  $i$  that is moved after a conditional instruction targeting a label  $l$  in case the condition is `true` (left). Then, in order to preserve the semantics, we must ensure that if the condition is true during execution the instruction  $i$  is executed. We insert a “stub”, i.e. we hijack the control by making the conditional point to a new label  $l'$  where the instruction  $i$  is executed before going back to the label  $l$ .



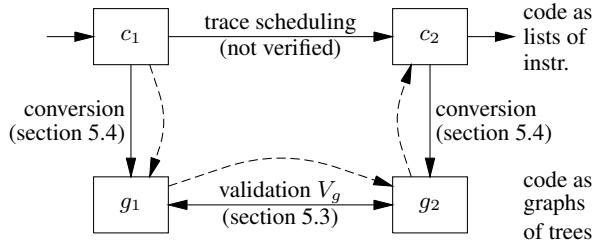
**Figure 1.** The two extra rules of trace scheduling. On the left, an example of move after a condition. On the right, an example of move before a join point. On each example the code is shown before and after hijacking.

Dually, consider an instruction  $i$  that is moved before a label  $l$  targeted by some instruction `goto` ( $l$ ) (right part of figure 1). To ensure semantics preservation, we must hijack the control of the `goto` into a new stub that contains the instruction  $i$ . This way,  $i$  is executed even if we enter the trace by following the `goto`.

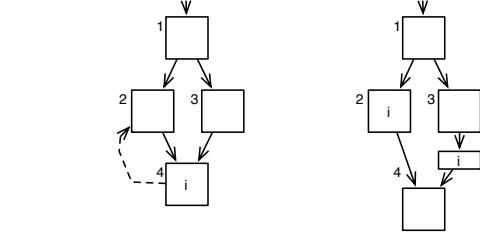
In list scheduling, the extent of code modifications was limited: an instruction can only move within the basic block that contains it. The “unit” of modification was therefore the block, i.e. the longest sequences of non-branching instructions between branches. During validation, the branches can then be used as “synchronization points” at which we check that the semantics are preserved. What are the synchronization points for trace scheduling? The only instructions that limit code movement are the return instructions and the target of back-edges, i.e. in our setting, a subset of the labels. We also fix the convention that `call` instructions cannot be crossed. Those instructions are our synchronization points. In conclusion, the unit of modification for trace scheduling is the longest sequence of instructions between these synchronization points.

As in the case of list scheduling, we would like to build the validator in two steps: first, build a function that validates pairs of traces that are expected to match; then, extend it to a validator for whole function bodies. The problem is that a trace can contain branching instructions. Our previous block validator does not handle this. Moreover, we must ensure that control flows the same way in the two programs, which was obvious for the block validator since states were equivalent before branching instructions, but is no longer true for trace scheduling because of the insertion of compensating code along some control edges.

A solution to these problems is to consider another representation of the program where traces can be manipulated as easily as blocks were in the list of instructions representation. This representation is a graph of trees, each tree being a compact representation of all the traces eligible for scheduling that begin at *cut points* in the control-flow graph. The cut points of interest, in our setting, are



**Figure 2.** Overview of trace scheduling and its validation. Solid arrows represent code transformations and validations. Dashed arrows represent proofs of semantic preservation.



function entry points, calls, returns, and the labels that are targets of back-edges. The important property of these trees is that if an instruction has been moved then it must be within the boundaries of a tree.

The validator for trace scheduling is built using this program representation. To complete the validator we must transform our program given as a list of instructions into a semantically equivalent control-flow graph of trees. The transformation to this new representation also requires some code annotation. This leads to the architecture depicted in figure 2 that we will detail in the remainder of this section. Note that the transformation from lists of instructions to graphs of trees needs to be proved semantics-preserving in both directions: if the list  $c$  is transformed to graph  $g$ , it must be the case that  $g$  executes from state  $S$  to state  $S'$  if and only if  $c$  executes from  $S$  to  $S'$ .

### 5.1 A tree-based representation of control and its semantics

Figure 3 illustrates our tree-based representation of the code of a function. In this section, we formally define its syntax and semantics.

**Syntax** The code of a function is represented as a mapping from labels to trees. Each label corresponds to a cut point in the control-flow graph of the function. A node of a tree is labeled either by a non-branching instruction, with one child representing its unique successor; or by a conditional instruction, with two children for its two successors. The leaves of instruction trees are `out`( $l$ ) nodes, carrying the label  $l$  of the tree to which control is transferred. Finally, special one-element trees are introduced to represent call and return instructions.

**Instruction trees:**

$$T ::= \text{seq}(i, T) \quad (i \text{ a non-branching instruction}) \\ | \text{ cond}(\text{cond}, r, T_1, T_2) \\ | \text{ out}(l)$$

**Call trees:**

$$T_c ::= \text{call}((r \mid id), l)$$

**Return trees:**

$$T_r ::= \text{return}$$

**Control-flow graphs:**

$$g ::= l \mapsto (T \mid T_c \mid T_r)$$

**Functions:**

$$f ::= \text{fun } id \\ \quad \{ \text{stack } n_1; \text{frame } n_2; \text{entry } l; \text{code } g; \}$$

**Semantics** The operational semantics of the tree-based representation is a combination of small-step and big-step styles. We describe executions of instruction trees using a big-step semantics

<b>label</b> $l_0$	
<b>op<sub>1</sub></b>	
<b>cond</b> $\dots, l_1$	$l_0 \mapsto \text{seq}(\text{op}_1, \text{cond}(\dots, \text{seq}(\text{op}_3, \text{out}(l_4)), \text{out}(l_2)))$
<b>label</b> $l_2$	$l_2 \mapsto \text{seq}(\text{op}_2, \text{cond}(\dots, \text{seq}(\text{op}_3, \text{out}(l_4)), \text{out}(l_2)))$
<b>op<sub>2</sub></b>	
<b>cond</b> $\dots, l_2$	$l_4 \mapsto \text{return}$
<b>label</b> $l_1$	
<b>op<sub>3</sub></b>	
<b>label</b> $l_4$	
<b>ret</b>	

**Figure 3.** A code represented as a list of instructions (left) and as a graph of instruction trees (right)

$\Sigma \vdash T, R, F, M \Rightarrow l, R', F', M'$ , meaning that the tree  $T$ , starting in state  $(R, F, M)$ , terminates on a branch to label  $l$  in state  $(R', F', M')$ . Since the execution of a tree cannot loop infinitely, this choice of semantics is adequate, and moreover is a good match for the validation algorithm operating at the level of trees that we develop next.

$$\begin{array}{c}
 \Sigma \vdash \text{out}(l), R, F, M \Rightarrow l, R, F, M \\
 \\
 \frac{\begin{array}{c} v = \text{eval\_op}(op, R(\vec{r})) \\ \Sigma \vdash T, R\{rd \leftarrow v\}, F, M \Rightarrow l, R', F', M' \end{array}}{\Sigma \vdash \text{seq}(op(op, \vec{r}, r), T), R, F, M \Rightarrow l, R', F', M'} \\
 \\
 \frac{\begin{array}{c} \text{true} = \text{eval\_condition}(cond, R(\vec{r})) \\ \Sigma \vdash T_1, R, F, M \Rightarrow l', R', F', M' \end{array}}{\Sigma \vdash \text{cond}(cond, \vec{r}, T_1, T_2), R, F, M \Rightarrow l', R', F', M'} \\
 \\
 \Sigma \vdash l, R, F, M \xrightarrow{*} l, R, F, M \\
 \\
 \frac{\begin{array}{c} \Sigma \vdash f.\text{graph}(l), R, F, M \Rightarrow l', R', F', M' \\ \Sigma \vdash l', R', F', M' \xrightarrow{*} l'', R'', F'', M'' \end{array}}{\Sigma \vdash l, R, F, M \xrightarrow{*} l'', R'', F'', M''}
 \end{array}$$

The predicate  $\Sigma \vdash l, R, F, M \xrightarrow{*} l', R', F', M'$ , defined in small-step style, expresses the chained evaluation of zero, one or several trees, starting at label  $l$  and ending at label  $l'$ .

$$\begin{array}{c}
 \Sigma \vdash l, R, F, M \xrightarrow{*} l, R, F, M \\
 \\
 \frac{\begin{array}{c} \Sigma \vdash f.\text{graph}(l), R, F, M \Rightarrow l', R', F', M' \\ \Sigma \vdash l', R', F', M' \xrightarrow{*} l'', R'', F'', M'' \end{array}}{\Sigma \vdash l, R, F, M \xrightarrow{*} l'', R'', F'', M''}
 \end{array}$$

Finally, the predicate for evaluation of function calls,  $G \vdash f, P, R, M \Rightarrow v, R', M'$ , is re-defined in terms of trees in the obvious manner.

$$\begin{array}{c}
 \text{alloc}(M, 0, f.\text{stack}) = (sp, M_1) \\
 \text{init\_frame}(f.\text{frame}) = F_1 \quad f.\text{graph} = g \\
 G, f, sp, P \vdash g(f.\text{entry}), R, M_1 \xrightarrow{*} l, R', M_2 \\
 g(l) = \text{return} \quad M' = \text{free}(M_2, sp) \\
 \hline
 G \vdash f, P, R, M \Rightarrow R', M'
 \end{array}$$

## 5.2 Validation at the level of trees

We first define a validator  $V_t$  that checks semantic preservation between two instruction trees  $T_1, T_2$ .

$$\begin{array}{c}
 V_t(T_1, T_2, (m_1, s_1), (m_2, s_2)) = \\
 \text{if } T_1 = \text{seq}(i_1, T'_1): \\
 \quad \text{return } V_t(T'_1, T_2, \alpha(i_1, (m_1, s_1)), (m_2, s_2)) \\
 \text{if } T_2 = \text{seq}(i_2, T'_2): \\
 \quad \text{return } V_t(T_1, T'_2, (m_1, s_1), \alpha(i_2, (m_2, s_2))) \\
 \text{if } T_1 = \text{cond}(cond_1, \vec{r}_1, T'_1, T''_1) \\
 \text{and } T_2 = \text{cond}(cond_2, \vec{r}_2, T'_2, T''_2):
 \end{array}$$

$$\begin{array}{c}
 \text{return } cond_1 = cond_2 \wedge m_1(\vec{r}_1) = m_2(\vec{r}_2) \\
 \wedge V_t(T'_1, T'_2, (m_1, s_1), (m_2, s_2)) \\
 \wedge V_t(T''_1, T''_2, (m_1, s_1), (m_2, s_2)) \\
 \text{if } T_1 = \text{out}(l_1) \text{ and } T_2 = \text{out}(l_2): \\
 \quad \text{return } l_2 = l_1 \wedge m_2 = m_1 \wedge s_2 \subseteq s_1 \\
 \text{in all other cases:} \\
 \quad \text{return false}
 \end{array}$$

The validator traverses the two trees in parallel, performing symbolic evaluation of the non-branching instructions. We reuse the  $\alpha(i, (m, s))$  function of section 4.2. The  $(m_1, s_1)$  and  $(m_2, s_2)$  parameters are the current states of symbolic evaluation for  $T_1$  and  $T_2$ , respectively. We process non-branching instructions repeatedly in  $T_1$  or  $T_2$  until we reach either two **cond** nodes or two **out** leaves. When we reach **cond** nodes in both trees, we check that the conditions being tested and the symbolic evaluations of their arguments are identical, so that at run-time control will flow on the same side of the conditional in both codes. We then continue validation on the **true** subtrees and on the **false** subtrees. Finally, when two **out** leaves are reached, we check that they branch to the same label and that the symbolic states agree ( $m_2 = m_1$  and  $s_2 \subseteq s_1$ ), as in the case of block verification.

As expected, a successful run of  $V_t$  entails a semantic preservation result.

**Lemma 5.** if  $V_t(T_1, T_2) = \text{true}$  and  $\Sigma \vdash T_1, R, F, M \Rightarrow l, R', F', M'$  then  $\Sigma \vdash T_2, R, F, M \Rightarrow l, R', F', M'$

## 5.3 Validation at the level of function bodies

We now extend the tree validator  $V_t$  to a validator that operates over two control-flow graphs of trees. We simply check that identically-labeled regular trees in both graphs are equivalent according to  $V_t$ , and that call trees and return trees are identical in both graphs.

$$\begin{array}{c}
 V_g(g_1, g_2) = \\
 \text{if } \text{Dom}(g_1) \neq \text{Dom}(g_2), \text{return false} \\
 \text{for each } l \in \text{Dom}(g_1): \\
 \quad \text{if } g_1(l) \text{ and } g_2(l) \text{ are regular trees:} \\
 \quad \quad \text{if } V_t(g_1(l), g_2(l), (\varepsilon, \emptyset), (\varepsilon, \emptyset)) = \text{false}, \text{return false} \\
 \quad \quad \text{otherwise:} \\
 \quad \quad \quad \text{if } g_1(l) \neq g_2(l), \text{return false} \\
 \quad \quad \text{end for each} \\
 \quad \text{return true}
 \end{array}$$

This validator is correct in the following sense. Let  $p, p'$  be two programs in the tree-based representation such that  $p'$  is identical to  $p$  except for the function bodies, and  $V_g(p(id).\text{graph}, p'(id).\text{graph}) = \text{true}$  for all function names  $id \in p$ .

**Theorem 3.** Let  $G$  and  $G'$  be the global environments associated with  $p$  and  $p'$ , respectively. If  $G \vdash f, P, R, M \Rightarrow R', M'$

and  $V_g(f.\text{graph}, f'.\text{graph}) = \text{true}$ , then  $G' \vdash f', P, R, M \Rightarrow R', M'$ .

#### 5.4 Conversion to the graph-of-trees representation

The validator developed in section 5.3 operates over functions whose code is represented as graphs of instruction trees. However, the unverified trace scheduler, as well as the surrounding compiler passes, consume and produce Mach code represented as lists of instructions. Therefore, before invoking the validator  $V_g$ , we need to convert the original and scheduled codes from the list-of-instructions representation to the graph-of-trees representation. To prove the correctness of this algorithm, we need to show that the conversion preserves semantics in both directions, or in other terms that each pair of a list of instructions and a graph of trees is semantically equivalent.

The conversion algorithm is conceptually simple, but not entirely trivial. In particular, it involves the computation of back edges in order to determine the cut points. Instead of writing the conversion algorithm in Coq and proving directly its correctness, we chose to use the translation validation approach one more time. In other terms, the conversion from lists of instructions to graphs of trees is written in unverified Caml, and complemented with a validator, written and proved in Coq, which takes a Mach function  $f$  (with its code represented as a list of instructions) and a graph of trees  $g$  and checks that  $f.\text{code}$  and  $g$  are semantically equivalent. This check is written  $f.\text{code} \sim g$ . The full validator for trace scheduling is therefore of the following form:

$$\begin{aligned} V(f_1, f_2) = \\ & \text{convert } f_1.\text{code} \text{ to a graph of trees } g_1 \\ & \text{convert } f_2.\text{code} \text{ to a graph of trees } g_2 \\ & \text{return } f_1.\text{code} \sim g_1 \wedge f_2.\text{code} \sim g_2 \wedge V_g(g_1, g_2) \end{aligned}$$

To check that an instruction sequence  $C$  and a graph  $g$  are equivalent, written  $C \sim g$ , we enumerate the cut points  $l \in \text{Dom}(g)$  and check that the list  $c$  of instructions starting at point  $l$  in the instruction sequence  $C$  corresponds to the tree  $g(l)$ . We write this check as a predicate  $C, \mathcal{B} \vdash c \sim T$ , where  $\mathcal{B} = \text{Dom}(g)$  is the set of cut points. The intuition behind this check is that every possible execution path in  $c$  should correspond to a path in  $T$  that executes the same instructions. In particular, if  $c$  starts with a non-branching instruction, we have

$$\frac{i \text{ non-branching} \quad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash i :: c \sim \text{seq}(i, T)}$$

Unconditional and conditional branches appearing in  $c$  need special handling. If the target  $l$  of the branch is a cut point ( $l \in \mathcal{B}$ ), this branch terminates the current trace and enters a new trace; it must therefore corresponds to an  $\text{out}(l)$  tree.

$$\begin{aligned} l \in \mathcal{B} \\ \hline C, \mathcal{B} \vdash \text{label}(l) :: c \sim \text{out}(l) \\ l \in \mathcal{B} \\ \hline C, \mathcal{B} \vdash \text{goto}(l) :: c \sim \text{out}(l) \\ l_{\text{true}} \in \mathcal{B} \quad C, \mathcal{B} \vdash c \sim T \\ \hline C, \mathcal{B} \vdash \text{cond}(\text{cond}, \vec{r}, l_{\text{true}}) :: c \sim \text{cond}(\text{cond}, \vec{r}, \text{out}(l_{\text{true}}), T) \end{aligned}$$

However, if  $l$  is not a cut point ( $l \notin \mathcal{B}$ ), the branch or label in  $c$  is not materialized in the tree  $T$  and is just skipped.

$$\frac{l \notin \mathcal{B} \quad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash \text{label}(l) :: c \sim T}$$

$$\frac{l \notin \mathcal{B} \quad c' = \text{find\_label}(l, C) \quad C, \mathcal{B} \vdash c' \sim T}{C, \mathcal{B} \vdash \text{goto}(l) :: c \sim T}$$

$$\frac{l_{\text{true}} \notin \mathcal{B} \quad c' = \text{find\_label}(l_{\text{true}}, C) \quad C, \mathcal{B} \vdash c \sim T \quad C, \mathcal{B} \vdash c' \sim T'}{C, \mathcal{B} \vdash \text{cond}(\text{cond}, \vec{r}, l_{\text{true}}) :: c \sim \text{cond}(\text{cond}, \vec{r}, T', T)}$$

An interesting fact is that the predicate  $C, \mathcal{B} \vdash c \sim T$  indirectly checks that  $\mathcal{B}$  contains at least all the targets of back-edges in the code  $C$ . For if this were not the case, the code  $C$  would contain a loop that does not go through any cut point, and we would have to apply one of the three “skip” rules above an infinite number of times; therefore, the inductive predicate  $C, \mathcal{B} \vdash c \sim T$  cannot hold. As discussed in section 6, the implementation of the  $\sim$  check (shown in appendix A) uses a counter of instructions traversed to abort validation instead of diverging in the case where  $\mathcal{B}$  incorrectly fails to account for all back-edges.

The equivalence check  $C \sim g$  defined above enjoys the desired semantic equivalence property:

**Lemma 6.** *Let  $p$  be a Mach program and  $p'$  a corresponding program where function bodies are represented as graphs of trees. Assume that  $p(id).\text{code} \sim p'(id).\text{code}$  for all function names  $id \in p$ . Let  $G$  and  $G'$  be the global environments associated with  $p$  and  $p'$ , respectively. If  $f.\text{code} \sim f'.\text{code}$ , then  $G \vdash f, P, R, M \Rightarrow R', M'$  in the original Mach semantics if and only if  $G' \vdash f', P, R, M \Rightarrow R', M'$  in the tree-based semantics of section 5.1.*

The combination of Theorem 3 and Lemma 6 establishes the correctness of the validator for trace scheduling.

## 6. The Coq mechanization

The algorithms presented in this paper have been formalized in their entirety and proved correct using the Coq proof assistant version 8.1. The Coq mechanization is mostly straightforward. Operational semantics are expressed as inductive predicates following closely the inference rules shown in this paper. The main difficulty was to express the algorithms as computable functions within Coq. Generally speaking, there are two ways to specify an algorithm in Coq: either as inductive predicates using inference rules, or as computable functions defined by recursion and pattern-matching over tree-shaped data structures. We chose the second presentation because it enables the automatic generation of executable Caml code from the specifications; this Caml code can then be linked with the hand-written Caml implementations of the unverified transformations.

However, Coq is a logic of total functions, so the function definitions must be written in a so-called “structurally recursive” style where termination is obvious. All our validation functions are naturally structurally recursive, except validation between trees (function  $V_t$  in section 5.2) and validation of list-to-tree conversion (the function corresponding to the  $\sim$  predicate in section 5.4).

For validation between trees, we used well-founded recursion, using the sum of the heights of the two trees as the decreasing, positive measure. Coq 8.1 provides good support for this style of recursive function definitions (the `Function` mechanism (Barthe et al. 2006)) and for the corresponding inductive proof principles (the `functional induction` tactic).

Validation of list-to-tree conversion could fail to terminate if the original, list-based code contains a loop that does not cross any cut point. This indicates a bug in the external converter, since normally cut points include all targets of back edges. To detect this situation and to make the recursive definition of the validator acceptable to

Coq, we add a counter as parameter to the validation function, initialized to the number of instructions in the original code and decremented every time we examine an instruction. If this counter drops to zero, validation stops on error. Appendix A shows the corresponding validation algorithm.

The Coq development accounts for approximatively 11000 lines of code. It took one person-year to design the validators, program them and prove their correctness. Figure 5 is a detailed line count showing, for each component of the validators, the size of the specifications (*i.e.* the algorithms and the semantics) and the size of the proofs.

	Specifications	Proofs	Total
Symbolic evaluation	736	1079	1815
Block validation	348	1053	1401
Block semantics	190	150	340
Block scheduling validation	264	590	854
Trace validation	234	1045	1279
Tree semantics	986	2418	3404
Trace scheduling validation	285	352	637
Label manipulation	306	458	764
Typing	114	149	263
Total	3463	7294	10757

**Figure 5.** Size of the development (in non-blank lines of code, without comments)

It is interesting to note that the validation of trace scheduling is no larger and no more difficult than that of list scheduling. This is largely due to the use of the graph-of-trees representation of the code. However, the part labeled “tree semantics”, which includes the definition and semantics of trees plus the validation of the conversion from list-of-instructions to graph-of-trees, is the largest and most difficult part of this development.

## 7. Preliminary experimental evaluation and algorithmic issues

Executable validators were extracted automatically from the Coq formalization of the algorithms presented in this paper and connected to two implementations of scheduling optimizations written in Caml: one for basic-block scheduling using the standard list scheduling algorithm, the other for trace scheduling. The two verified compilation passes thus obtained were integrated in the CompCert experimental compiler (Leroy 2006; Leroy et al. 2003–2007), and tested on the test suite of this compiler (a dozen Cminor programs in the 100–1000 l.o.c. range). This test suite is too small to draw any definitive conclusion. We nonetheless include the experimental results because they point out potential algorithmic inefficiencies in our approach.

All tests were successfully scheduled and validated after scheduling. Manual inspection of the scheduled code reveals that the schedulers performed a fair number of instruction reorderings and, in the case of trace scheduling, insertion of stubs. Validation was effective from a compiler engineering viewpoint: not only manual injection of errors in the schedulers were correctly caught, but the validator also found one unintentional bug in our first implementation of trace scheduling.

To assess the compile-time overheads introduced by validation, we measured the execution times of the two scheduling transformations and of the corresponding validators. Figure 4 presents the results.

The tests were conducted on a Pentium 4 3.4 GHz Linux machine with 2 GB of RAM. Each pass was repeated as many times

as needed for the measured time to be above 1 second; the times reported are averages.

On all tests except AES, the time spent in validation is comparable to that spent in the non-verified scheduling transformation. The total time (transformation + validation) of instruction scheduling is about 10% of the whole compilation time. The AES test (the optimized reference implementation of the AES encryption algorithm) demonstrates some inefficiencies in our implementation of validation, which takes about 10 times longer than the corresponding transformation, both for list scheduling and for trace scheduling.

There are two potential sources of algorithmic inefficiencies in the validation algorithms presented in this paper. The first is the comparison between the symbolic codes and constraint sets generated by symbolic execution. Viewed as a tree, the symbolic code for a block of length  $n$  can contain up to  $2^n$  nodes (consider for instance the block  $r_1 = r_0 + r_0; \dots; r_n = r_{n-1} + r_{n-1}$ ). Viewed as a DAG, however, the symbolic code has size linear in the length  $n$  of the block, and can be constructed in linear time. However, the comparison function between symbolic codes that we defined in Coq compares symbolic codes as trees, ignoring sharing, and can therefore take  $O(2^n)$  time. Using a hash-consed representation for symbolic expressions would lead to much better performance: construction of the symbolic code would take time  $O(n \log n)$  (the  $\log n$  accounts for the overhead of hash consing), comparison between symbolic codes could be done in time  $O(1)$ , and inclusion between sets of constraints in time  $O(n \log n)$ . We haven’t been able to implement this solution by lack of an existing Coq library for hash-consed data structures, so we leave it for future work.

The second source of algorithmic inefficiency is specific to trace scheduling. The tree-based representation of code that we use for validation can be exponentially larger than the original code represented as a list of instructions, because of tail duplication of basic blocks. This potential explosion caused by tail duplication can be avoided by adding more cut points: not just targets of back edges, but also some other labels chosen heuristically to limit tail duplication. For instance, we can mark as cut points all the labels that do not belong to the traces that the scheduler chose to optimize. Such heuristic choices are performed entirely in unverified code (the scheduler and the converter from list- to tree-based code representations) and have no impact on the validators and on their proofs of correctness.

## 8. Related work

The idea of translation validation appears in the work of Pnueli et al. (1998a,b). It was initially conducted in the context of the compilation of a synchronous language. The principle of the validator is to generate verification conditions that are solved by a model checker. The authors mention the possibility of generating a proof script during model checking, which generates additional confidence in the correctness of a run of validation, but is weaker than a full formal verification of the validator as in the present paper.

The case of an optimizing compiler for a conventional, imperative language has been addressed by Zuck et al. (2001, 2003) and Barret et al. (2005). They use a generalization of the Floyd method to generate verification conditions that are sent to a theorem prover. Two validators have been produced that implement this framework: `voc-64` (Zuck et al. 2003) for the SGI `pro-64` compiler and `TVOC` (Barret et al. 2005) for the `ORC` compiler. This work addresses advanced compiler transformations, including non-structure preserving transformations such as loop optimizations (Goldberg et al. 2005) and software pipelining (Leviathan and Pnueli 2006).

Test program	List scheduling			Trace scheduling		
	Transformation	Validation	Ratio $V/T$	Transformation	Validation	Ratio $V/T$
<b>fib</b>	0.29 ms	0.47 ms	1.60	0.44 ms	0.58 ms	1.32
<b>integr</b>	0.91 ms	0.87 ms	0.96	1.0 ms	1.2 ms	1.15
<b>qsort</b>	1.3 ms	1.5 ms	1.15	1.8 ms	3.3 ms	1.89
<b>fft</b>	9.1 ms	18 ms	1.98	19 ms	62 ms	3.26
<b>sha1</b>	9.4 ms	6.7 ms	0.71	12 ms	24 ms	2.00
<b>aes</b>	56 ms	550 ms	9.76	67 ms	830 ms	12.25
<b>almabench</b>	25 ms	16 ms	0.65	56 ms	200 ms	3.57
<b>stopcopy</b>	4.1 ms	4.1 ms	1.00	4.9 ms	6.1 ms	1.25
<b>marksweep</b>	5.3 ms	6.3 ms	1.18	6.8 ms	11 ms	1.69

**Figure 4.** Compilation times and verification times

While the approach of Pnueli, Zuck et al. relies on verification condition generators and theorem proving, a different approach based on abstract interpretation and static analysis was initiated by Necula (2000). He developed a validator for the GCC 2.7 compiler, able to validate most of the optimisations implemented by this compiler. His approach relies on symbolic execution of RTL intermediate code and inspired the present work. Necula’s validator addresses a wider range of optimizations than ours, requiring him to compare symbolic executions modulo arithmetic and memory-related equations.

Rival (2004) describes a translation validator for GCC 3.0 without optimizations. While Necula’s validator handles only transformations over the RTL intermediate language, Rival’s relates directly the C source code with the generated PowerPC assembly code. Rival’s validator uses Symbolic Transfer Functions to represent the behaviour of code fragments. While Necula and Rival do not discuss instruction scheduling in detail, we believe that their validators can easily handle list scheduling (reordering of instructions within basic blocks), but we do not know whether they can cope with the changes in the control-flow graph introduced by trace scheduling.

Huang et al. (2006) describe a translation validator specialized to the verification of register allocation and spilling. Their algorithm relies on data-flow analyses: computation and correlation of webs of def-use sequences. By specializing the validator to the code transformations that a register allocator typically performs, they claim to obtain a validator that is complete (no false alarms), and they can also produce detailed explanations of errors.

Compared with validators based on verification condition generators, validators based on static analysis like Necula’s, Rival’s, Huang et al.’s and ours are arguably less powerful but algorithmically more efficient, making it realistic to perform validation at every compilation run. Moreover, it seems easier to characterize the classes of transformations that can be validated.

While several of the papers mentioned above come with on-paper proofs, none has been mechanically verified. There are, however, several mechanized verifications of static analyzers, i.e. tools that establish properties of one piece of compiled code instead of relating two pieces of compiled code like translation validators do, in particular the JVM bytecode verifier (Klein and Nipkow 2003) and data flow analyzers (Cachera et al. 2005).

## 9. Conclusions and further work

We presented what we believe is the first fully mechanized verification of translation validators. The two validators presented here were developed with list scheduling and trace scheduling in mind, but they seem applicable to a wider class of code transformations: those that reorder, factor out or duplicate instructions within basic blocks or instruction trees (respectively), without taking advantage

of non-aliasing information. For instance, this includes common subexpression elimination, as well as rematerialization. We believe (without any proof) that our validators are complete, that is, raise no false alarms for this class of transformations.

It is interesting to note that the validation algorithms proceed very differently from the code transformations that they validate. The validators uses notions such as symbolic execution and block- or tree-based decompositions of program executions that have obvious semantic meanings. In contrast, the optimizations rely on notions such as RAW/WAR/WAW dependencies and back-edges whose semantic meaning is much less obvious. For this reason, we believe (without experience to substantiate this claim) that it would be significantly more difficult to prove directly the correctness of list scheduling or trace scheduling.

A direct extension of the present work is to prove semantic preservation not only for terminating executions, but also for diverging executions. The main reason why our proofs are restricted to terminating evaluations is the use of big-step operational semantics. A small-step (transition) semantics for Mach is in development, and should enable us to extend the proofs of semantic preservation to diverging executions. Another, more difficult extension is to take non-aliasing information into account in order to validate reorderings between independent loads and stores.

More generally, there are many other optimizations for which it would be interesting to formally verify the corresponding validation algorithms. Most challenging are the optimizations that move computations across loop boundaries, such as loop invariant hoisting and software pipelining.

## Acknowledgments

Julien Forest helped us use the new Coq feature `Function` and improved its implementation at our request. We thank Alain Frisch for discussions and feedback.

## References

- Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.
- Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and Logic Programming, 8th Int. Symp., FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006.
- Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

- David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. *Theoretical Computer Science*, 342(1):56–78, 2005.
- Coq development team. The Coq proof assistant. Software and documentation available at <http://coq.inria.fr/>, 1989–2007.
- Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.
- John R. Ellis. *Bulldog: a compiler for VLSI architectures*. ACM Doctoral Dissertation Awards. The MIT Press, 1986.
- Benjamin Goldberg, Lenore Zuck, and Clark Barret. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2004)*, volume 132 of *Electronic Notes in Theoretical Computer Science*, pages 53–71. Elsevier, 2005.
- Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.
- Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.
- Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
- D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a CO compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.
- Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
- Xavier Leroy et al. The CompCert certified compiler back-end. Development available at <http://gallium.inria.fr/~xleroy/compcert-backend/>, 2003–2007.
- Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Int. Conf. On Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2002)*, pages 280–287. ACM Press, 2006.
- Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
- George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.
- Amir Pnueli, Ofer Shtrichman, and Michael Siegel. The code validation tool (CVT) – automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2:192–201, 1998a.
- Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998b.
- Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
- Emin Gün Sirer and Brian N. Bershad. Testing Java virtual machines. In *Proc. Int. Conf. on Software Testing And Review*, 1999.
- Martin Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.
- L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann institute of Science, 2001.
- Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

## A. Appendix: the validation algorithm for conversion from instruction lists to instruction trees

We show here the algorithm that determines semantic equivalence between a list of instructions and an instruction tree, corresponding to the predicate  $f, \mathcal{B} \vdash c \sim T$  in section 5.4.

```

let rec skip_control B func c counter =
2   if counter = 0
      then None
4   else
      match c with
6        | Mlabel lbl :: c' →
          if lbl in B
            then Some (Mlabel lbl :: c')
            else skip_control B func c' (counter - 1)
10       | Mgoto lbl :: c' →
           match find_label lbl func with
12         | Some c'' → if lbl in B
             then Some (Mgoto lbl :: c')
             else skip_control B func c'' (counter - 1)
14         | None → None
16         | i :: c' → Some (i :: c')
18         | _ → None
19
let test_out sub lbl =
20   match sub with
21     | out lbl' → lbl = lbl'
22     | _ → false
23
let rec validTreeBase B f cur t =
24   let cur' = skip_control B (fn_code f) cur (length (fn_code f)) in
25   match cur', t with
26     | Some(getstack(i, t, m) :: 1), getstack(i', t', m', sub) →
27       i = i' ∧ t = t' ∧ m = m' ∧
28       validTreeBase B f 1 sub
29     | Some(setstack(m, i, t) :: 1), setstack(m', i', t', sub) →
30       i = i' ∧ t = t' ∧ m = m' ∧
31       validTreeBase B f 1 sub
32     | Some(getparam(i, t, m) :: 1), getparam(i', t', m', sub) →
33       i = i' ∧ t = t' ∧ m = m' ∧
34       validTreeBase B f 1 sub
35     | Some(op(op, lr, m) :: 1), op(op', lr', m', sub) →
36       op = op' ∧ lr = lr' ∧ m = m' ∧
37       validTreeBase B f 1 sub
38     | Some(load(chk, addr, lr, m) :: 1), load(chk', addr', lr', m', sub) →
39       addr = addr' ∧ chk = chk' ∧ lr = lr' ∧ m = m' ∧
40       validTreeBase B f 1 sub
41     | Some(store(chk, addr, lr, m) :: 1), store(chk', addr', lr', m', sub) →
42       addr = addr' ∧ chk = chk' ∧ lr = lr' ∧ m = m' ∧
43       validTreeBase B f 1 sub
44     | Some(cond(c, rl, lbl) :: 1), cond(c', rl', sub1, sub2) →
45       c = c' ∧ rl = rl' ∧
46       validTreeBase B f 1 sub2 ∧
47       (if lbl in B
48         then test_out sub1 lbl
49         else match find_label lbl (fn_code f) with
50           | Some l' → validTreeBase B f l' sub1
51           | None → false )
52     | Some(label(lbl) :: 1), out(lbl') → lbl = lbl'
53     | Some(goto(lbl) :: 1), out(lbl') → lbl = lbl'
54     | _, _ → false

```

# Verified Validation of Lazy Code Motion

Jean-Baptiste Tristan

INRIA Paris-Rocquencourt  
jean-baptiste.tristan@inria.fr

Xavier Leroy

INRIA Paris-Rocquencourt  
xavier.leroy@inria.fr

## Abstract

Translation validation establishes *a posteriori* the correctness of a run of a compilation pass or other program transformation. In this paper, we develop an efficient translation validation algorithm for the Lazy Code Motion (LCM) optimization. LCM is an interesting challenge for validation because it is a global optimization that moves code across loops. Consequently, care must be taken not to move computations that may fail before loops that may not terminate. Our validator includes a specific check for anticipability to rule out such incorrect moves. We present a mechanically-checked proof of correctness of the validation algorithm, using the Coq proof assistant. Combining our validator with an unverified implementation of LCM, we obtain a LCM pass that is provably semantics-preserving and was integrated in the CompCert formally verified compiler.

**Categories and Subject Descriptors** D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness proofs; D.3.4 [*Programming Languages*]: Processors - Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Operational semantics

**General Terms** Languages, Verification, Algorithms

**Keywords** Translation validation, lazy code motion, redundancy elimination, verified compilers, the Coq proof assistant

## 1. Introduction

Advanced compiler optimizations perform subtle transformations over the programs being compiled, exploiting the results of delicate static analyses. Consequently, compiler optimizations are sometimes incorrect, causing the compiler either to crash at compile-time, or to silently generate bad code from a correct source program. The latter case is especially troublesome since such compiler-introduced bugs are very difficult to track down. Incorrect optimizations often stem from bugs in the implementation of a correct optimization algorithm, but sometimes the algorithm itself is faulty, or the conditions under which it can be applied are not well understood.

The standard approach to weeding out incorrect optimizations is heavy testing of the compiler. Translation validation, as introduced

by Pnueli et al. (1998b), provides a more systematic way to detect (at compile-time) semantic discrepancies between the input and the output of an optimization. At every compilation run, the input code and the generated code are fed to a validator (a piece of software distinct from the compiler itself), which tries to establish *a posteriori* that the generated code behaves as prescribed by the input code. If, however, the validator detects a discrepancy, or is unable to establish the desired semantic equivalence, compilation is aborted; some validators also produce an explanation of the error.

Algorithms for translation validation roughly fall in two classes. (See section 9 for more discussion.) General-purpose validators such as those of Pnueli et al. (1998b), Necula (2000), Barret et al. (2005), Rinard and Marinov (1999) and Rival (2004) rely on generic techniques such as symbolic execution, model-checking and theorem proving, and can therefore be applied to a wide range of program transformations. Since checking semantic equivalence between two code fragments is undecidable in general, these validators can generate false alarms and have high complexity. If we are interested only in a particular optimization or family of related optimizations, special-purpose validators can be developed, taking advantage of our knowledge of the limited range of code transformations that these optimizations can perform. Examples of special-purpose validators include that of Huang et al. (2006) for register allocation and that of Tristan and Leroy (2008) for list and trace instruction scheduling. These validators are based on efficient static analyses and are believed to be correct and complete.

This paper presents a translation validator specialized to the Lazy Code Motion (LCM) optimization of Knoop et al. (1992, 1994). LCM is an advanced optimization that removes redundant computations; it includes common subexpression elimination and loop-invariant code motion as special cases, and can also eliminate partially redundant computations (i.e. computations that are redundant on some but not all program paths). Since LCM can move computations across basic blocks and even across loops, its validation appears more challenging than that of register allocation or trace scheduling, which preserve the structure of basic blocks and extended basic blocks, respectively. As we show in this work, the validation of LCM turns out to be relatively simple (at any rate, much simpler than the LCM algorithm itself): it exploits the results of a standard available expression analysis. A delicate issue with LCM is that it can anticipate (insert earlier computations of) instructions that can fail at run-time, such as memory loads from a potentially invalid pointer; if done carelessly, this transformation can turn code that diverges into code that crashes. To address this issue, we complement the available expression analysis with a so-called “anticipability checker”, which ensures that the transformed code is at least as defined as the original code.

Translation validation provides much additional confidence in the correctness of a program transformation, but does not completely rule out the possibility of compiler-introduced bugs: what if the validator itself is buggy? This is a concern for the development of critical software, where systematic testing does not suffice

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'09, June 15–20, 2009, Dublin, Ireland.  
Copyright © 2009 ACM 978-1-60558-392-1/09/06...\$5.00

to reach the desired level of assurance and must be complemented by formal verification of the source. Any bug in the compiler can potentially invalidate the guarantees obtained by this use of formal methods. One way to address this issue is to formally verify the compiler itself, proving that every pass preserves the semantics of the program being compiled. Several ambitious compiler verification efforts are currently under way, such as the Jinja project of Klein and Nipkow (2006), the Verisoft project of Leinenbach et al. (2005), and the CompCert project of Leroy et al. (2004–2009).

Translation validation can provide semantic preservation guarantees as strong as those obtained by formal verification of a compiler pass: it suffices to prove that the validator is correct, i.e. returns `true` only when the two programs it receives as inputs are semantically equivalent. The compiler pass itself does not need to be proved correct. As illustrated by Tristan and Leroy (2008), the proof of a validator can be significantly simpler and more reusable than that of the corresponding optimizations. The translation validator for LCM presented in this paper was mechanically verified using the Coq proof assistant (Coq development team 1989–2009; Bertot and Castéran 2004). We give a detailed overview of this proof in sections 5 to 7. Combining the verified validator with an unverified implementation of LCM written in Caml, we obtain a provably correct LCM optimization that integrates smoothly within the CompCert verified compiler (Leroy et al. 2004–2009).

The remainder of this paper is as follows. After a short presentation of Lazy Code Motion (section 3) and of the RTL intermediate language over which it is performed (section 2), section 4 develops our validation algorithm for LCM. The next three sections outline the correctness proof of this algorithm: section 5 gives the dynamic semantics of RTL, section 6 presents the general shape of the proof of semantic preservation using a simulation argument, and section 7 details the LCM-specific aspects of the proof. Section 8 discusses other aspects of the validator and its proof, including completeness, complexity, performance and reusability. Related work is discussed in section 9, followed by conclusions in section 10.

## 2. The RTL intermediate language

The LCM optimization and its validation are performed on the RTL intermediate language of the CompCert compiler. This is a standard Register Transfer Language where control is represented by a control flow graph (CFG). Nodes of the CFG carry abstract instructions, corresponding roughly to machine instructions but operating over pseudo-registers (also called temporaries). Every function has an unlimited supply of pseudo-registers, and their values are preserved across function calls.

An RTL program is a collection of functions plus some global variables. As shown in figure 1, functions come in two flavors: external functions  $ef$  are merely declared and model input-output operations and similar system calls; internal functions  $f$  are defined within the language and consist of a type signature  $sig$ , a parameter list  $\vec{r}$ , the size  $n$  of their activation record, an entry point  $l$ , and a CFG  $g$  representing the code of the function. The CFG is implemented as a finite map from node labels  $l$  (positive integers) to instructions. The set of instructions includes arithmetic operations, memory load and stores, conditional branches, and function calls, tail calls and returns. Each instruction carries the list of its successors in the CFG. When the successor  $l$  is irrelevant or clear from the context, we use the following more readable notations for register-to-register moves, arithmetic operations, and memory loads:

$$\begin{array}{ll} r := r' & \text{for } \text{op}(\text{move}, r', r, l) \\ r := \text{op}(op, \vec{r}) & \text{for } \text{op}(op, \vec{r}, r, l) \\ r := \text{load}(chunk, mode, \vec{r}) & \text{for } \text{load}(chunk, mode, \vec{r}, r, l) \end{array}$$

A more detailed description of RTL can be found in (Leroy 2008).

RTL instructions:	
$i ::= \text{nop}(l)$	no operation
$  \text{op}(op, \vec{r}, r, l)$	arithmetic operation
$  \text{load}(chunk, mode, \vec{r}, r, l)$	memory load
$  \text{store}(chunk, mode, \vec{r}, r, l)$	memory store
$  \text{call}(sig, (r \mid id), \vec{r}, r, l)$	function call
$  \text{tailcall}(sig, (r \mid id), \vec{r})$	function tail call
$  \text{cond}(cond, \vec{r}, l_{\text{true}}, l_{\text{false}})$	conditional branch
$  \text{return} \mid \text{return}(r)$	function return
Control-flow graphs:	
$g ::= l \mapsto i$	finite map
RTL functions:	
$fd ::= f \mid ef$	
$f ::= id \{ \text{sig } sig;$	internal function
$\text{params } \vec{r}; \text{ stack } n;$	
$\text{start } l; \text{ graph } g \}$	
$ef ::= id \{ \text{sig } sig \}$	external function

Figure 1. RTL syntax

## 3. Lazy Code Motion

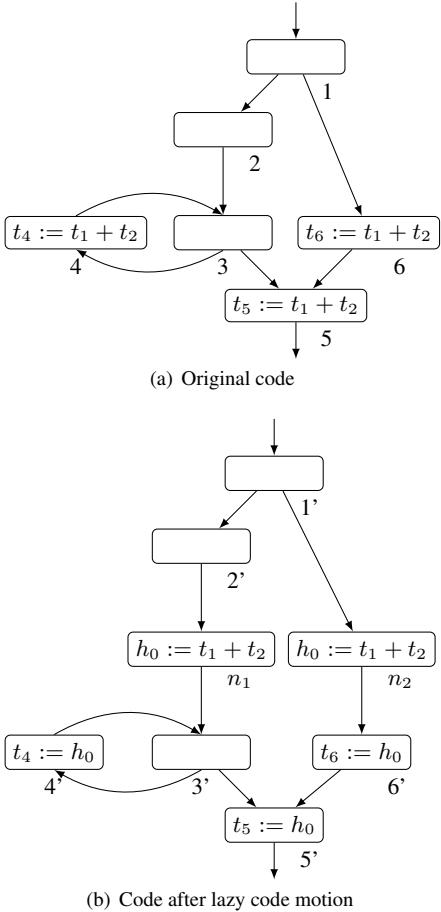
Lazy code motion (LCM) (Knoop et al. 1992, 1994) is a dataflow-based algorithm for the placement of computations within control flow graphs. It suppresses unnecessary recomputations of values by moving their first computations earlier in the execution flow (if necessary), and later reusing the results of these first computations. Thus, LCM performs elimination of common subexpressions (both within and across basic blocks), as well as loop invariant code motion. In addition, it can also factor out partially redundant computations: computations that occur multiple times on some execution paths, but once or not at all on other paths. LCM is used in production compilers, for example in GCC version 4.

Figure 2 presents an example of lazy code motion. The original program in part (a) presents several interesting cases of redundancies for the computation of  $t_1 + t_2$ : loop invariance (node 4), simple straight-line redundancy (nodes 6 and 5), and partial redundancy (node 5). In the transformed program (part (b)), these redundant computations of  $t_1 + t_2$  have all been eliminated: the expression is computed at most once on every possible execution path. Two instructions (node  $n_1$  and  $n_2$ ) have been added to the graph, both of which compute  $t_1 + t_2$  and save its result into a fresh temporary  $h_0$ . The three occurrences of  $t_1 + t_2$  in the original code have been rewritten into move instructions (nodes 4', 5' and 6'), copying the fresh  $h_0$  register to the original destinations of the instructions.

The reader might wonder why two instructions  $h_0 := t_1 + t_2$  were added in the two branches of the conditional, instead of a single instruction before node 1. The latter is what the partial redundancy elimination optimization of Morel and Renvoise (1979) would do. However, this would create a long-lived temporary  $h_0$ , therefore increasing register pressure in the transformed code. The “lazy” aspect of LCM is that computations are placed as late as possible while avoiding repeated computations.

The LCM algorithm exploits the results of 4 dataflow analyses: up-safety (also called availability), down-safety (also called anticipability), delayability and isolation. These analyses can be implemented efficiently using bit vectors. Their results are then cleverly combined to determine an optimal placement for each computation performed by the initial program.

Knoop et al. (1994) presents a correctness proof for LCM. However, mechanizing this proof appears difficult. Unlike the program transformations that have already been mechanically verified in the CompCert project, LCM is a highly non-local transformation: in-



**Figure 2.** An example of lazy code motion transformation

structions are moved across basic blocks and even across loops. Moreover, the transformation generates fresh temporaries, which adds significant bureaucratic overhead to mechanized proofs. It appears easier to follow the verified validator approach. An additional benefit of this approach is that the LCM implementation can use efficient imperative data structures, since we do not need to formally verify them. Moreover, it makes it easier to experiment with other variants of LCM.

To design and prove correct a translation validator for LCM, it is not important to know all the details of the analyses that indicate where new computations should be placed and which instructions should be rewritten. However it is important to know what kind of transformations happen. Two kinds of rewritings of the graph can occur:

- The nodes that exist in the original code (like node 4 in figure 2) still exist in the transformed code. The instruction they carry is either unchanged or can be rewritten as a move if they are arithmetic operations or loads (but not calls, tail calls, stores, returns nor conditions).
- Some fresh nodes are added (like node  $n_1$ ) to the transformed graph. Their left-hand side is a fresh register; their right-hand side is the right-hand side of some instructions in the original code.

There exists an injective function from nodes of the original code to nodes of the transformed code. We call this mapping  $\varphi$ .

It connects each node of the source code to its (possibly rewritten) counterpart in the transformed code. In the example of figure 2,  $\varphi$  maps nodes 1...6 to their primed versions 1'...6'. We assume the unverified implementation of LCM is instrumented to produce this function. (In our implementation, we arrange that  $\varphi$  is always the identity function.) Nodes that are not in the image of  $\varphi$  are the fresh nodes introduced by LCM.

## 4. A translation validator for Lazy Code Motion

In this section, we detail a translation validator for LCM.

### 4.1 General structure

Since LCM is an intraprocedural optimization, the validator proceeds function per function: each internal function  $f$  of the original program is matched against the identically-named function  $f'$  of the transformed program. Moreover, LCM does not change the type signature, parameter list and stack size of functions, and can be assumed not to change the entry point (by inserting nops at the graph entrance if needed). Checking these invariants is easy; hence, we can focus on the validation of function graphs. Therefore, the validation algorithm is of the following shape:

```
validate( $f, f', \varphi$ ) =
let  $\mathcal{AE}$  = analyze( $f'$ ) in
 $f'.\text{sig} = f.\text{sig}$  and  $f'.\text{params} = f.\text{params}$  and
 $f'.\text{stack} = f.\text{stack}$  and  $f'.\text{start} = f.\text{start}$  and
for each node  $n$  of  $f$ ,  $V(f, f', n, \varphi, \mathcal{AE}) = \text{true}$ 
```

As discussed in section 3, the  $\varphi$  parameter is the mapping from nodes of the input graph to nodes of the transformed graph provided by the implementation of LCM. The `analyze` function is a static analysis computing available expressions, described below in section 4.2.1. The `V` function validates pairs of matching nodes and is composed of two checks: `unify`, described in section 4.2.2 and `path`, described in section 4.3.2.

```
 $V(f, f', n, \varphi, \mathcal{AE}) =$ 
  unify( $\mathcal{RD}(n')$ ,  $f.\text{graph}(n)$ ,  $f'.\text{graph}(\varphi(n))$ )
  and for all successor  $s$  of  $n$  and matching successor  $s'$  of  $n'$ ,
  path( $f.\text{graph}$ ,  $f'.\text{graph}$ ,  $s', \varphi(s)$ )
```

As outlined above, our implementation of a validator for LCM is carefully structured in two parts: a generic, rather bureaucratic framework parameterized over the `analyze` and `V` functions; and the LCM-specific, more subtle functions `analyze` and `V`. As we will see in section 7, this structure facilitates the correctness proof of the validator. It also makes it possible to reuse the generic framework and its proof in other contexts, as illustrated in section 8.

We now focus on the construction of `V`, the node-level validator, and the static analysis it exploits.

### 4.2 Verification of the equivalence of single instructions

Consider an instruction  $i$  at node  $n$  in the original code and the corresponding instruction  $i'$  at node  $\varphi(n)$  in the code after LCM (for example, nodes 4 and 4' in figure 2). We wish to check that these two instructions are semantically equivalent. If the transformation was a correct LCM, two cases are possible:

- $i = i'$ : both instructions will obviously lead to equivalent runtime states, if executed in equivalent initial states.
- $i'$  is of the form  $r := h$  for some register  $r$  and fresh register  $h$ , and  $i$  is of the form  $r := rhs$  for some right-hand side  $rhs$ , which can be either an arithmetic operation  $\text{op}(\dots)$  or a memory read  $\text{load}(\dots)$ .

In the latter case, we need to verify that  $rhs$  and  $h$  produce the same value. More precisely, we need to verify that the value contained

in  $h$  in the transformed code is equal to the value produced by evaluating  $rhs$  in the original code. LCM being a purely syntactical redundancy elimination transformation, it must be the case that the instruction  $h := rhs$  exists on every path leading to  $\varphi(n)$  in the transformed code; moreover, the values of  $h$  and  $rhs$  are preserved along these paths. This property can be checked by performing an available expression analysis on the transformed code.

#### 4.2.1 Available expressions

The available expression analysis produces, for each program point of the transformed code, a set of equations  $r = rhs$  between registers and right-hand sides. (For efficiency, we encode these sets as finite maps from registers to right-hand sides, represented as Patricia trees.) Available expressions is a standard forward dataflow analysis:

$$\mathcal{AE}(s) = \bigcap \{T(f'.\text{graph}(l), \mathcal{AE}(l)) \mid s \text{ is a successor of } l\}$$

The join operation is set intersection; the top element of the lattice is the empty set, and the bottom element is a symbolic constant  $\mathcal{U}$  denoting the universe of all equations. The transfer function  $T$  is standard; full details can be found in the Coq development. For instance, if the instruction  $i$  is the operation  $r := t_1 + t_2$ , and  $R$  is the set of equations “before”  $i$ , the set  $T(i, R)$  of equations “after”  $i$  is obtained by adding the equality  $r = t_1 + t_2$  to  $R$ , then removing every equality in this set that uses register  $r$  (including the one just added if  $t_1$  or  $t_2$  equals  $r$ ). We also track equalities between register and load instructions. Those equalities are erased whenever a `store` instruction is encountered because we do not maintain aliasing information.

To solve the dataflow equations, we reuse the generic implementation of Kildall’s algorithm provided by the CompCert compiler. Leveraging the correctness proof of this solver and the definition of the transfer function, we obtain that the equations inferred by the analysis hold in any concrete execution of the transformed code. For example, if the set of equations at point  $l$  include the equality  $r = t_1 + t_2$ , it must be the case that  $R(r) = R(t_1) + R(t_2)$  for every possible execution of the program that reaches point  $l$  with a register state  $R$ .

#### 4.2.2 Instruction unification

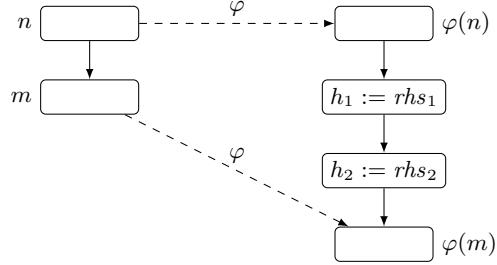
Armed with the results of the available expression analysis, the `unify` check between pairs of matching instructions can be easily expressed:

```
unify(D, i, i') =
  if i' = i then true else
  case (i, i') of
    | (r := op(op, r̄), r := h) →
      (h = op(op, r̄)) ∈ D
    | (r := load(chunk, mode, r̄), r := h) →
      (h = load(chunk, mode, r̄)) ∈ D
    | otherwise → false
```

Here,  $D = \mathcal{AE}(n')$  is the set of available expressions at the point  $n'$  where the transformed instruction  $i'$  occurs. Either the original instruction  $i$  and the transformed instruction  $i'$  are equal, or the former is  $r := rhs$  and the latter is  $r := h$ , in which case instruction unification succeeds if and only if the equation  $h = rhs$  is known to hold according to the results of the available expression analysis.

#### 4.3 Verifying the flow of control

Unifying pairs of instructions is not enough to guarantee semantic preservation: we also need to check that the control flow is preserved. For example, in the code shown in figure 2, after checking that the conditional tests at nodes 1 and  $1'$  are identical, we must



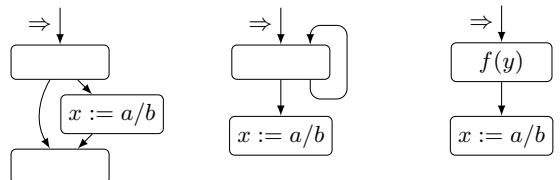
**Figure 3.** Effect of the transformation on the structure of the code

make sure that whenever the original code transitions from node 1 to node 6, the transformed code can transition from node  $1'$  to 6, executing the anticipated computation at  $n_2$  on its way.

More generally, if the  $k$ -th successor of  $n$  in the original CFG is  $m$ , there must exist a path in the transformed CFG from  $\varphi(n)$  to  $\varphi(m)$  that goes through the  $k$ -th successor of  $\varphi(n)$ . (See figure 3.) Since instructions can be added to the transformed graph during lazy code motion,  $\varphi(m)$  is not necessarily the  $k$ -th successor of  $\varphi(n)$ : one or several anticipated computations of the shape  $h := rhs$  may need to be executed. Here comes a delicate aspect of our validator: not only there must exist a path from  $\varphi(n)$  to  $\varphi(m)$ , but moreover the anticipated computations  $h := rhs$  found on this path must be semantically well-defined: they should not go wrong at run-time. This is required to ensure that whenever an execution of the original code transitions in one step from  $n$  to  $m$ , the transformed code can transition (possibly in several steps) from  $\varphi(n)$  to  $\varphi(m)$  without going wrong.

Figure 4 shows three examples of code motion where this property may not hold. In all three cases, we consider anticipating the computation  $a/b$  (an integer division that can go wrong if  $b = 0$ ) at the program points marked by a double arrow. In the leftmost example, it is obviously unsafe to compute  $a/b$  before the conditional test: quite possibly, the test in the original code checks that  $b \neq 0$  before computing  $a/b$ . The middle example is more subtle: it could be the case that the loop preceding the computation of  $a/b$  does not terminate whenever  $b = 0$ . In this case, the original code never crashes on a division by zero, but anticipating the division before the loop could cause the transformed program to do so. The rightmost example is similar to the middle one, with the loop being replaced by a function call. The situation is similar because the function call may not terminate when  $b = 0$ .

How, then, can we check that the instructions that have been added to the graph are semantically well-defined? Because we distinguish erroneous executions and diverging executions, we cannot rely on a standard anticipability analysis. Our approach is the following: whenever we encounter an instruction  $h := rhs$  that was inserted by the LCM transformation on the path from  $\varphi(n)$



**Figure 4.** Three examples of incorrect code motion. Placing a computation of  $a/b$  at the program points marked by  $\Rightarrow$  can potentially transform a well-defined execution into an erroneous one.

```

1 function ant_checker_rec (g,rhs,pc,S) =
2
3   case S(pc) of
4     | Found →(S,true)
5     | NotFound →(S,false)
6     | Visited →(S,false)
7     | Dunno →
8
9     case g(pc) of
10    | return _ →(S{pc ← NotFound},false)
11    | tailcall (...,...) →(S{pc ← NotFound},false)
12    | cond (...,ltrue,lfalse) →
13      let (S',b1) = ant_checker_rec (g,rhs,ltrue,S{pc ← Visited}) in
14      let (S'',b2) = ant_checker_rec (g,rhs,lfalse,S') in
15      if b1&& b2then (S''{pc ← Found},true) else (S''{pc ← NotFound},false)
16    | nop l →
17      let (S',b) := ant_checker_rec (g,rhs,l,S{pc ← Visited}) in
18      if b then (S'{pc ← Found},true) else (S'{pc ← NotFound},false)
19    | call (...,...,l) →(S{pc ← NotFound},false)
20    | store (...,...,l) →
21      if rhs reads memory then (S{pc ← NotFound},false) else
22      let (S',b) := ant_checker_rec (g,rhs,l,S{pc ← Visited}) in
23      if b then (S'{pc ← Found},true) else (S'{pc ← NotFound},false)
24    | op (op,args,r,l) →
25      if r is an operand of rhs then (S{pc ← NotFound},false) else
26      if rhs = (op op args) then (S{pc ← Found},true) else
27        let (S',b) = ant_checker_rec (g,rhs,l,S{pc ← Visited}) in
28        if b then (S'{pc ← Found},true) else (S'{pc ← NotFound},false)
29    | load (chk,addr,args,r,l) →
30      if r is an operand of rhs then (S{pc ← NotFound},false) else
31      if rhs = (load chk addr args) then (S{pc ← Found},true) else
32        let (S',b) = ant_checker_rec (g,rhs,l,S{pc ← Visited}) in
33        if b then (S'{pc ← Found},true) else (S'{pc ← NotFound},false)
34
35
36 function ant_checker (g,rhs,pc) = let (S,b) = ant_checker_rec(g,rhs,pc,(l ↦ Dunno)) in b

```

Figure 5. Anticipability checker

to  $\varphi(m)$ , we check that the computation of  $rhs$  is *inevitable* in the original code starting at node  $m$ . In other words, all execution paths starting from  $m$  in the original code must, in a finite number of steps, compute  $rhs$ . Since the semantic preservation result that we wish to establish takes as an assumption that the execution of the original code does not go wrong, we know that the computation of  $rhs$  cannot go wrong, and therefore it is legal to anticipate it in the transformed code. We now define precisely an algorithm, called the *anticipability checker*, that performs this check.

#### 4.3.1 Anticipability checking

Our algorithm is described in figure 5. It takes four arguments: a graph  $g$ , an instruction right-hand side  $rhs$  to search for, a program point  $l$  where the search begins and a map  $S$  that associates to every node a marker. Its goal is to verify that on every path starting at  $l$  in the graph  $g$ , execution reaches an instruction with right-hand side  $rhs$  such that none of the operands of  $rhs$  have been redefined on the path. Basically it is a depth-first search that covers all the path starting at  $l$ . Note that if there is a path starting at  $l$  that contains a loop so that  $rhs$  is neither between  $l$  and the loop nor in the loop itself, then there exists a path on which  $rhs$  is not reachable and that corresponds to an infinite execution. To obtain an efficient algorithm, we need to ensure that we do not go through loops several times. To this end, if the search reaches a join point not for the first time and where  $rhs$  was not found before, we must

stop searching immediately. This is achieved through the use of four different markers over nodes:

- **Found** means that  $rhs$  is computed on every path from the current node.
- **NotFound** means that there exists a path from the current node in which  $rhs$  is not computed.
- **Dunno** is the initial state of every node before it has been visited.
- **Visited** is the state when a state is visited and we do not know yet whether  $rhs$  is computed on all paths or not. It is used to detect loops.

Let us detail a few cases. When the search reaches a node that is marked **Visited** (line 6), it means that the search went through a loop and  $rhs$  was not found. This could lead to a semantics discrepancy (recall the middle example in figure 4) and the search fails. For similar reasons, it also fails when a call is reached (line 19). When the search reaches an operation (line 24), we first verify (line 25) that  $r$ , the destination register of the instruction does not modify the operands of  $rhs$ . Then, (line 26) if the instruction right-hand side we reached correspond to  $rhs$ , we found  $rhs$  and we mark the node accordingly. Otherwise, the search continues (line 27) and we mark the node based on whether the recursive search found  $rhs$  or not (line 28).

The `ant_checker` function, when it returns `Found`, should imply that the right-hand side expression is well defined. We prove that this is the case in section 7.3 below.

### 4.3.2 Verifying the existence of semantics paths

Once we can decide the well-definedness of instructions, checking for the existence of a path between two nodes of the transformed graph is simple. The function `path`( $g, g', n, m$ ) checks that there exists a path in CFG  $g'$  from node  $n$  to node  $m$ , composed of zero, one or several single-successor instructions of the form  $h := rhs$ . The destination register  $h$  must be fresh (unused in  $g$ ) so as to preserve the abstract semantics equivalence invariant. Moreover, the right-hand side  $rhs$  must be safely anticipable: it must be the case that  $\text{ant\_checker}(g, rhs, \varphi^{-1}(m)) = \text{Found}$ , so that  $rhs$  can be computed before reaching  $m$  without getting stuck.

## 5. Dynamic semantics of RTL

In preparation for a proof of correctness of the validator, we now outline the dynamic semantics of the RTL language. More details can be found in (Leroy 2008). The semantics manipulates values, written  $v$ , comprising 32-bit integers, 64-bit floats, and pointers. Several environments are involved in the semantics. Memory states  $M$  map pointers and memory chunks to values, in a way that accounts for byte addressing and possible overlap between chunks (Leroy and Blazy 2008). Register files  $R$  map registers to values. Global environments  $G$  associate pointers to names of global variables and functions, and function definitions to function pointers.

The semantics of RTL programs is given in small-step style, as a transition relation between execution states. Three kinds of states are used:

- Regular states:  $\mathcal{S}(\Sigma, f, \sigma, l, R, M)$ . This state corresponds to an execution point within the internal function  $f$ , at node  $l$  in the CFG of  $f$ .  $R$  and  $M$  are the current register file and memory state.  $\Sigma$  represents the call stack, and  $\sigma$  points to the activation record for the current invocation of  $f$ .
- Call states:  $\mathcal{C}(\Sigma, fd, \vec{v}, M)$ . This is an intermediate state representing an invocation of function  $F_d$  with parameters  $\vec{v}$ .
- Return states:  $\mathcal{R}(\Sigma, v, M)$ . Symmetrically, this intermediate state represents a function return, with return value  $v$  being passed back to the caller.

Call stacks  $\Sigma$  are lists of frames  $\mathcal{F}(r, f, \sigma, l, R)$ , where  $r$  is the destination register where the value computed by the callee is to be stored on return,  $f$  is the caller function, and  $\sigma, l$  and  $R$  its local state at the time of the function call.

The semantics is defined by the one-step transition relation  $G \vdash S \xrightarrow{t} S'$ , where  $G$  is the global environment (invariant during execution),  $S$  and  $S'$  the states before and after the transition, and  $t$  a trace of the external function call possibly performed during the transition. Traces record the names of external functions invoked, along with the argument values provided by the program and the return value provided by the external world.

To give a flavor of the semantics and show the level of detail of the formalization, figure 6 shows a subset of the rules defining the one-step transition relation. For example, the first rule states that if the program counter  $l$  points to an instruction that is an operation of the form  $\text{op}(op, \vec{r}, r_d, l')$ , and if evaluating the operator  $op$  on the values contained in the registers  $\vec{r}$  of the register file  $R$  returns the value  $v$ , then we transition to a new regular state where the register  $r_d$  of  $R$  is updated to hold the value  $v$ , and the program counter moves to the successor  $l'$  of the operation. The only rule that produces a non-empty trace is the one for external function invocations (last rule in figure 6); all other rules produce the empty trace  $\varepsilon$ .

$$\begin{array}{c}
f.\text{graph}(l) = \text{op}(op, \vec{r}, r_d, l') \quad v = \text{eval\_op}(G, op, R(\vec{r})) \\
G \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, f, \sigma, l', R\{r_d \leftarrow v\}, M)
\end{array}$$

$$\begin{array}{c}
f.\text{graph}(l) = \text{call}(sig, r_f, \vec{r}, r_d, l') \quad G(R(r_f)) = fd \\
fd.sig = sig \quad \Sigma' = \mathcal{F}(r_d, f, \sigma, l', R). \Sigma
\end{array}$$


---


$$\begin{array}{c}
G \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{C}(\Sigma', fd, \vec{v}, M) \\
f.\text{graph}(l) = \text{return}(r) \quad v = R(r) \\
G \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{R}(\Sigma, v, M)
\end{array}$$

$$\begin{array}{c}
\Sigma = \mathcal{F}(r_d, f, \sigma, l, R). \Sigma' \\
G \vdash \mathcal{R}(\Sigma, v, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma', f, \sigma, l, R, M)
\end{array}$$

$$\begin{array}{c}
\text{alloc}(M, 0, f.\text{stacksize}) = (\sigma, M') \\
l = f.\text{start} \quad R = [f.\text{params} \mapsto \vec{v}]
\end{array}$$

$$G \vdash \mathcal{C}(\Sigma, f, \vec{v}, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, f, \sigma, l, R, M)$$

$$\begin{array}{c}
t = (ef.\text{name}, \vec{v}, v) \\
G \vdash \mathcal{C}(\Sigma, ef, \vec{v}, M) \xrightarrow{t} \mathcal{R}(\Sigma, v, M)
\end{array}$$

**Figure 6.** Selected rules from the dynamic semantics of RTL

Sequences of transitions are captured by the following closures of the one-step transition relation:

$$\begin{array}{ll}
G \vdash S \xrightarrow{t^*} S' & \text{zero, one or several transitions} \\
G \vdash S \xrightarrow{t^+} S' & \text{one or several transitions} \\
G \vdash S \xrightarrow{T} \infty & \text{infinitely many transitions}
\end{array}$$

The finite trace  $t$  and the finite or infinite trace  $T$  record the external function invocations performed during these sequences of transitions. The observable behavior of a program  $P$ , then, is defined in terms of the traces corresponding to transition sequences from an initial state to a final state. We write  $P \Downarrow B$  to say that program  $P$  has behavior  $B$ , where  $B$  is either termination with a finite trace  $t$ , or divergence with a possibly infinite trace  $T$ . Note that computations that go wrong, such as an integer division by zero, are modeled by the absence of a transition. Therefore, if  $P$  goes wrong, then  $P \Downarrow B$  does not hold for any  $B$ .

## 6. Semantics preservation for LCM

Let  $P_i$  be an input program and  $P_o$  be the output program produced by the untrusted implementation of LCM. We wish to prove that if the validator succeeds on all pairs of matching functions from  $P_i$  and  $P_o$ , then  $P_i \Downarrow B \Rightarrow P_o \Downarrow B$ . In other words, if  $P_i$  does not go wrong and executes with observable behavior  $B$ , then so does  $P_o$ .

### 6.1 Simulating executions

The way we build a semantics preservation proof is to construct a relation between execution states of the input and output programs, written  $S_i \sim S_o$ , and show that it is a simulation:

- Initial states: if  $S_i$  and  $S_o$  are two initial states, then  $S_i \sim S_o$ .
- Final states: if  $S_i \sim S_o$  and  $S_i$  is a final state, then  $S_o$  must be a final state.
- Simulation property: if  $S_i \sim S_o$ , any transition from state  $S_i$  with trace  $t$  is simulated by one or several transitions starting in state  $S_o$ , producing the same trace  $t$ , and preserving the simulation relation  $\sim$ .

The hypothesis that the input program  $P_i$  does not go wrong plays a crucial role in our semantic preservation proof, in particular to show the correctness of the anticipability criterion. Therefore,

we reflect this hypothesis in the precise statement of the simulation property above, as follows. ( $G_i, G_o$  are the global environments corresponding to programs  $P_i$  and  $P_o$ , respectively.)

**DEFINITION 1** (Simulation property).

Let  $I_i$  be the initial state of program  $P_i$  and  $I_o$  that of program  $P_o$ . Assume that

- $S_i \sim S_o$  (*current states are related*)
  - $G_i \vdash S_i \xrightarrow{t'} S'_i$  (*the input program makes a transition*)
  - $G_i \vdash I_i \xrightarrow{t'} S_i$  and  $G_o \vdash I_o \xrightarrow{t'} S_o$  (*current states are reachable from initial states*)
  - $G_i \vdash S'_i \Downarrow B$  for some behavior  $B$  (*the input program does not go wrong after the transition*).

Then, there exists  $S'_o$  such that  $G_o \vdash S_o \xrightarrow{t}^+ S'_o$  and  $S'_i \sim S'_o$ .

The commuting diagram corresponding to this definition is depicted below. Solid lines represent hypotheses; dashed lines represent conclusions.

It is easy to show that the simulation property implies semantic preservation:

**THEOREM 1.** Under the hypotheses between initial states and final states and the simulation property,  $P_i \downarrow B$  implies  $P_o \downarrow B$ .

## 6.2 The invariant of semantic preservation

We now construct the relation  $\sim$  between execution states before and after LCM that acts as the invariant in our proof of semantic preservation. We first define a relation between register files.

**DEFINITION 2** (Equivalence of register files).

$f \vdash R \sim R'$  if and only if  $R(r) = R'(r)$  for every register  $r$  that appears in an instruction of  $f$ 's code.

This definition allows the register file  $R'$  of the transformed function to bind additional registers not present in the original function, especially the temporary registers introduced during LCM optimization. Equivalence between execution states is then defined by the three rules below.

**DEFINITION 3** (Equivalence of execution states).

$$\begin{array}{c}
\text{validate}(f, f', \varphi) = \text{true} \quad f \vdash R \sim R' \quad G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma' \\
\hline
G, G' \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \sim \mathcal{S}(\Sigma', f', \sigma, \varphi(l), R', M) \\
\frac{\mathcal{T}_{\mathcal{V}}(fd) = fd' \quad G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'}{G, G' \vdash \mathcal{C}(\Sigma, fd, \vec{v}, M) \sim \mathcal{C}(\Sigma', fd', \vec{v}, M)} \\
\frac{G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'}{G, G' \vdash \mathcal{R}(\Sigma, v, M) \sim \mathcal{R}(\Sigma', v, M)}
\end{array}$$

Generally speaking, equivalent states must have exactly the same memory states and the same value components (stack pointer  $\sigma$ , arguments and results of function calls). As mentioned before, the register files  $R, R'$  of regular states may differ on temporary registers but must be related by the  $f \vdash R \sim R'$  relation. The function parts  $f, f'$  must be related by a successful run of validation. The program points  $l, l'$  must be related by  $l' = \varphi(l)$ .

The most delicate part of the definition is the equivalence between call stacks  $G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'$ . The frames of the two stacks  $\Sigma$  and  $\Sigma'$  must be related pairwise by the following predicate.

**DEFINITION 4** (Equivalence of stack frames).

$$\begin{aligned} \text{validate}(f, f', \varphi) = \text{true} & \quad f \vdash R \sim R' \\ \forall v, M, B, \quad G \vdash S(\Sigma, f, \sigma, l, R\{r \leftarrow v\}, M) \Downarrow B \\ \implies \exists R'', \quad f \vdash R\{r \leftarrow v\} \sim R'' & \\ \wedge \quad G' \vdash S(\Sigma, f', \sigma, l', R'\{r \leftarrow v\}, M) \\ \stackrel{\varepsilon_+}{\rightarrow} \quad S(\Sigma, f', \sigma, \varphi(l), R'', M) \end{aligned}$$


---


$$G, G' \vdash \mathcal{F}(r, f, \sigma, l, R) \sim_{\mathcal{F}} \mathcal{F}(r, f', \sigma, l', R')$$

The scary-looking third premise of the definition above captures the following condition: if we suppose that the execution of the initial program is well-defined once control returns to node  $l$  of the caller, then it should be possible to perform an execution in the transformed graph from  $l'$  down to  $\varphi(l)$ . This requirement is a consequence of the anticipability problem. As explained earlier, we need to make sure that execution is well defined from  $l'$  to  $\varphi(l)$ . But when the instruction is a function call, we have to store this information in the equivalence of frames, universally quantified on the not-yet-known return value  $v$  and memory state  $M$  at return time. At the time we store the property we do not know yet if the execution will be semantically correct from  $l$ , so we suppose it until we get the information (that is, when execution reaches  $l$ ).

Having stated semantics preservation as a simulation diagram and defined the invariant of the simulation, we now turn to the proof itself.

## 7. Sketch of the formal proof

This section gives a high-level overview of the correctness proof for our validator. It can be used as an introduction to the Coq development, which gives full details. Besides giving an idea of how we prove the validation kernel (this proof differs from earlier paper proofs mainly on the handling of semantic well-definedness), we try to show that the burden of the proof can be reduced by adequate design.

## 7.1 Design: getting rid of bureaucracy

Recall that the validator is composed of two parts: first, a generic validator that requires an implementation of  $V$  and of `analyze`; second, an implementation of  $V$  and `analyze` specialized for LCM. The proof follows this structure: on one hand, we prove that if  $V$  satisfies the simulation property, then the generic validator implies semantics preservation; on the other hand, we prove that the node-level validation specialized for LCM satisfies the simulation property.

This decomposition of the proof improves re-usability and, above all, greatly improves abstraction for the proof that  $V$  satisfies the simulation property (which is the kernel of the proof on which we want to focus) and hence reduces the proof burden of the formalization. Indeed, many details of the formalization can be hidden in the proof of the framework. This includes, among other things, function invocation, function return, global variables, and stack management.

Besides, this allows us to prove that  $V$  only satisfies a weaker version of the simulation property that we call the validation property, and whose equivalence predicate is a simplification of the equivalence presented in section 6.2. In the simplified equivalence predicate, there is no mention of stack equivalence, function transformation, stack pointers or results of the validation.

**DEFINITION 5** (Abstract equivalence of states).

$$\frac{f \vdash R \sim R' \quad l' = \varphi(l)}{G, G' \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \approx_{\mathcal{S}} \mathcal{S}(\Sigma', f', \sigma, l', R', M)}$$

$$G, G' \vdash \mathcal{C}(\Sigma, fd, \vec{v}, M) \approx_{\mathcal{C}} \mathcal{C}(\Sigma', fd', \vec{v}, M)$$

$$G, G' \vdash \mathcal{R}(\Sigma, v, M) \approx_{\mathcal{R}} \mathcal{R}(\Sigma', v, M)$$

The validation property is stated in three version, one for regular states, one for calls and one for return. We present only the property for regular states. If  $S = \mathcal{S}(\Sigma, f, \sigma, l, R, M)$  is a regular state, we write  $S.f$  for the  $f$  component of the state and  $S.l$  for the  $l$  component.

#### DEFINITION 6 (Validation property).

Let  $I_i$  be the initial state of program  $P_i$  and  $I_o$  that of program  $P_o$ . Assume that

- $S_i \approx_{\mathcal{S}} S_o$
- $G_i \vdash S_i \xrightarrow{t} S'_i$
- $G_i \vdash I_i \xrightarrow{t'}^* S_i$  and  $G_o \vdash I_o \xrightarrow{t'}^* S_o$
- $S'_i \Downarrow B$  for some behavior  $B$
- $V(S_i.f, S_o.f, S_i.l, \varphi, \text{analyze}(S_o.f)) = \text{true}$

Then, there exists  $S'_o$  such that  $S_o \xrightarrow{t+} S'_o$  and  $S'_i \approx S'_o$ .

We then prove that if  $V$  satisfies the validation property, and if the two programs  $P_i, P_o$  successfully pass validation, then the simulation property (definition 1) is satisfied, and therefore (theorem 1) semantic preservation holds. This proof is not particularly interesting but represents a large part of the Coq development and requires a fair knowledge of CompCert internals.

We now outline the formal proof of the fact that  $V$  satisfies the validation property, which is the more interesting part of the proof.

#### 7.2 Verification of the equivalence of single instructions

We first need to prove the correctness of the available expression analysis. The predicate  $S \models \mathcal{E}$  states that a set of equalities  $\mathcal{E}$  inferred by the analysis are satisfied in execution state  $S$ . The predicate is always true on call states and on return states.

#### DEFINITION 7 (Correctness of a set of equalities).

$\mathcal{S}(\Sigma, f, \sigma, l, R, M) \models \mathcal{RD}(l)$  if and only if

- $(r = \text{op}(op, \vec{r})) \in \mathcal{RD}(l)$  implies  $R(r) = \text{eval\_op}(op, R(\vec{r}))$
- $(r = \text{load}(chunk, mode, \vec{r})) \in \mathcal{RD}(l)$  implies  $\text{eval\_addressing}(mode, \vec{r}) = v$  and  $R(r) = \text{load}(chunk, v)$  for some pointer value  $v$ .

The correctness of the analysis can now be stated:

LEMMA 2 (Correctness of available expression analysis). Let  $S^0$  be the initial state of the program. For all regular states  $S$  such that  $S^0 \xrightarrow{t,*} S$ , we have  $S \models \text{analyze}(S.f)$ .

Then, it is easy to prove the correctness of the unification check. The predicate  $\approx_{\mathcal{S}}^W$  is a weaker version of  $\approx_{\mathcal{S}}$ , where we remove the requirement that  $l' = \varphi(l)$ , therefore enabling the program counter of the transformed code to temporarily get out of synchronization with that of the original code.

LEMMA 3. Assume

- $S_i \approx_{\mathcal{S}} S_o$
- $S_i \xrightarrow{t} S'_i$
- $\text{unify}(\text{analyze}(S_o.f), S_i.f.\text{graph}, S_o.f.\text{graph}, S_i.l, S_o.l) = \text{true}$
- $I_o \xrightarrow{t'}^* S_o$

Then, there exists a state  $S''_o$  such that  $S_o \xrightarrow{t} S''_o$  and  $S'_i \approx_{\mathcal{S}}^W S''_o$

Indeed, from the hypothesis  $I_o \xrightarrow{t'}^* S_o$  and the correctness of the analysis, we deduce that  $S_o \models \text{analyze}(S_o.f)$ , which implies that the equality used during the unification, if any, holds at run-time. This illustrate the use of hypothesis on the past of the execu-

tion of the transformed program. By doing so, we avoid to maintain the correctness of the analysis in the predicate of equivalence.

It remains to step through the transformed CFG, as performed by path checking, in order to go from the weak abstract equivalence  $\approx_{\mathcal{S}}^W$  to the full abstract equivalence  $\approx_{\mathcal{S}}$ .

#### 7.3 Anticipability checking

Before proving the properties of path checking, we need to prove the correctness of the anticipability check: if the check succeeds and the semantics of the input program is well defined, then the right-hand side expression given to the anticipability check is well defined.

LEMMA 4. Assume  $\text{ant\_checker}(f.\text{graph}, rhs, l) = \text{true}$  and  $\mathcal{S}(\Sigma, f, \sigma, l, R, M) \Downarrow B$  for some  $B$ . Then, there exists a value  $v$  such that  $rhs$  evaluates to  $v$  (without run-time errors) in the state  $R, M$ .

Then, the semantic property guaranteed by path checking is that there exists a sequence of reductions from  $\text{successor}(\varphi(n))$  to  $\varphi(\text{successor}(n))$  such that the abstract invariant of semantic equivalence is reinstated at the end of the sequence.

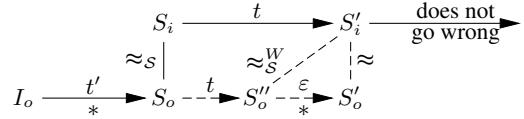
LEMMA 5. Assume

- $S'_i \approx_{\mathcal{S}}^W S''_o$
- $\text{path}(S'_i.f.\text{graph}, S''_o.f.\text{graph}, S''_o.l, \varphi(S_i.l)) = \text{true}$
- $S'_i \Downarrow B$  for some  $B$

Then, there exists a state  $S'_o$  such that  $S''_o \xrightarrow{\varepsilon,*} S'_o$  and  $S'_i \approx S'_o$

This illustrates the use of the hypothesis on the future of the execution of the initial program. All the proofs are rather straightforward once we know that we need to reason on the future of the execution of the initial program.

By combining lemmas 3 and 5 we prove the validation property for regular states, according to the following diagram.



The proofs of the validation property for call and return states are similar.

## 8. Discussion

**Implementation** The LCM validator and its proof of correctness were implemented in the Coq proof assistant. The Coq development is approximately 5000 lines long. 800 lines correspond to the specification of the LCM validator, in pure functional style, from which executable Caml code is automatically generated by Coq's extraction facility. The remaining 4200 lines correspond to the correctness proof. In addition, a lazy code motion optimization was implemented in OCaml, in roughly 800 lines of code.

The following table shows the relative sizes of the various parts of the Coq development.

Part	Size
General framework	37%
Anticipability check	16%
Path verification	7%
Reaching definition analysis	18%
Instruction unification	6%
Validation function	16%

As discussed below, large parts of this development are not specific to LCM and can be reused: the general framework of section 7.1,

anticipability checking, available expressions, etc. Assuming these parts are available as part of a toolkit, building and proving correct the LCM validator would require only 1100 lines of code and proofs.

**Completeness** We proved the correctness of the validator. This is an important property, but not sufficient in practice: a validator that rejects every possible transformation is definitely correct but also quite useless. We need evidence that the validator is relatively complete with respect to “reasonable” implementations of LCM. Formally specifying and proving such a relative completeness result is difficult, so we reverted to experimentation. We ran LCM and its validator on the CompCert benchmark suite (17 small to medium-size C programs) and on a number of examples hand-crafted to exercise the LCM optimization. No false alarms were reported by the validator.

More generally, there are two main sources of possible incompleteness in our validator. First, the external implementation of LCM could take advantage of equalities between right-hand sides of computations that our available expression analysis is unable to capture, causing instruction unification to fail. We believe this never happens as long as the available expression analysis used by the validator is identical to (or at least no coarser than) the up-safety analysis used in the implementation of LCM, which is the case in our implementation.

The second potential source of false alarms is the anticipability check. Recall that the validator prohibits anticipating a computation that can fail at run-time before a loop or function call. The CompCert semantics for the RTL language errs on the side of caution and treats all undefined behaviors as run-time failures: not just behaviors such as integer division by zero or memory loads from incorrect pointers, which can actually cause the program to crash when run on a real processor, but also behaviors such as adding two pointers or shifting an integer by more than 32 bits, which are not specified in RTL but would not crash the program during actual execution. (However, arithmetic overflows and underflows are correctly modeled as not causing run-time errors, because the RTL language uses modulo integer arithmetic and IEEE float arithmetic.) Because the RTL semantics treats all undefined behaviors as potential run-time errors, our validator restricts the points where e.g. an addition or a shift can be anticipated, while the external implementation of LCM could (rightly) consider that such a computation is safe and can be placed anywhere. This situation happened once in our tests.

One way to address this issue is to increase the number of operations that cannot fail in the RTL semantics. We could exploit the results of a simple static analysis that keeps track of the shape of values (integers, pointers or floats), such as the trivial “int or float” type system for RTL used in (Leroy 2008). Additionally, we could refine the semantics of RTL to distinguish between undefined operations that can crash the program (such as loads from invalid addresses) and undefined operations that cannot (such as adding two pointers); the latter would be modeled as succeeding, but returning an unspecified result. In both approaches, we increase the number of arithmetic instructions that can be anticipated freely.

**Complexity and performance** Let  $N$  be the number of nodes in the initial CFG  $g$ . The number of nodes in the transformed graph  $g'$  is in  $\mathcal{O}(N)$ . We first perform an available expression analysis on the transformed graph, which takes time  $\mathcal{O}(N^3)$ . Then, for each node of the initial graph we perform an unification and a path checking. Unification is done in constant time and path checking tries to find a non-cyclic path in the transformed graph, performing an anticipability checking in time  $\mathcal{O}(N)$  for instructions that may be ill-defined. Hence path checking is in  $\mathcal{O}(N^2)$  but this is a rough pessimistic approximation.

In conclusion, our validator runs in time  $\mathcal{O}(N^3)$ . Since lazy code motion itself performs four data-flow analysis that run in time  $\mathcal{O}(N^3)$ , running the validator does not change the complexity of the lazy code motion compiler pass.

In practice, on our benchmark suite, the time needed to validate a function is on average 22.5% of the time it takes to perform LCM.

**Reusing the development** One advantage of translation validation is the re-usability of the approach. It makes it easy to experiment with variants of a transformation, for example by using a different set of data-flow analyzes in lazy code motion. It also happens that, in one compiler, two different versions of a transformation co-exist. It is the case with GCC: depending on whether one optimizes for space or for time, the compiler performs partial redundancy elimination (Morel and Renvoise 1979) or lazy code motion. We believe, without any formal proof, that the validator presented here works equally well for partial redundancy elimination. In such a configuration, the formalization burden is greatly reduced by using translation validation instead of compiler proof.

Classical redundancy elimination algorithms make the safe restriction that a computation  $e$  cannot be placed on some control flow path that does not compute  $e$  in the original program. As a consequence, code motion can be blocked by *preventing regions* (Bodík et al. 1998), resulting in less redundancy elimination than expected, especially in loops. A solution to this problem is *safe speculative code motion* (Bodík et al. 1998) where we lift the restriction for some computation  $e$  as long as  $e$  cannot cause run-time errors. Our validator can easily handle this case: the anticipability check is not needed if the new instruction is safe, as can easily be checked by examination of this instruction. Another solution is to perform control flow restructuring (Steffen 1996; Bodík et al. 1998) to separate paths depending on whether they contain the computation  $e$  or not. This control flow transformation is not allowed by our validator and constitutes an interesting direction for future work.

To show that re-usability can go one step further, we have modified the unification rules of our lazy code motion validator to build a certified compiler pass of constant propagation with strength reduction. For this transformation, the available expression analysis needs to be performed not on the transformed code but on the initial one. Thankfully, the framework is designed to allow analyses on both programs. The modification mainly consists of replacing the unification rules for operation and loads, which represent about 3% of the complete development of LCM. (Note however that unification rules in the case of constant propagation are much bigger because of the multiple possible strength reductions). It took two weeks to complete this experiment. The proof of semantics preservation uses the same invariant as for lazy code motion and the proof remains unchanged apart from unification of operations and loads. Using the same invariant, although effective, is questionable: it is also possible to use a simpler invariant crafted especially for constant propagation with strength reduction.

One interesting possibility is to try to abstract the invariant in the development. Instead of posing a particular invariant and then develop the framework upon it, with maybe other transformations that will luckily fit the invariant, the framework is developed with an unknown invariant on which we suppose some properties. (See Zuck et al. (2001) for more explanations.) We may hope that the resulting tool/theory be general enough for a wider class of transformations, with the possibility that the analyses have to be adapted. For example, by replacing the available expression analysis by global value numbering of Gulwani and Necula (2004), it is possible that the resulting validator would apply to a large class of redundancy elimination transformations.

## 9. Related Work

Since its introduction by Pnueli et al. (1998a,b), translation validation has been actively researched in several directions. One direction is the construction of general frameworks for validation (Zuck et al. 2001, 2003; Barret et al. 2005; Zaks and Pnueli 2008). Another direction is the development of generic validation algorithms that can be applied to production compilers (Rinard and Marinov 1999; Necula 2000; Zuck et al. 2001, 2003; Barret et al. 2005; Rival 2004; Kanade et al. 2006). Finally, validation algorithms specialized to particular classes of transformations have also been developed, such as (Huang et al. 2006) for register allocation or (Tristan and Leroy 2008) for instruction scheduling. Our work falls in the latter approach, emphasizing algorithmic efficiency and relative completeness over generality.

A novelty of our work is its emphasis on fully mechanized proofs of correctness. While unverified validators are already very useful to increase confidence in the compilation process, a formally verified validator provides an attractive alternative to the formal verification of the corresponding compiler pass (Leinenbach et al. 2005; Klein and Nipkow 2006; Leroy 2006; Lerner et al. 2003; Blech et al. 2005). Several validation algorithms or frameworks use model checking or automatic theorem proving to check verification conditions produced by a run of validation (Zuck et al. 2001, 2003; Barret et al. 2005; Kanade et al. 2006), but the verification condition generator itself is, generally, not formally proved correct.

Many validation algorithms restrict the amount of code motion that the transformation can perform. For example, validators based on symbolic evaluation such as (Necula 2000; Tristan and Leroy 2008) easily support code motion within basic blocks or extended basic blocks, but have a hard time with global transformations that move instructions across loops, such as LCM. We are aware of only one other validator that handles LCM: that of Kanade et al. (2006). In their approach, LCM is instrumented to produce a detailed trace of the code transformations performed, each of these transformations being validated by reduction to a model-checking problem. Our approach requires less instrumentation (only the  $\varphi$  code mapping needs to be provided) and seems algorithmically more efficient.

As mentioned earlier, global code motion requires much care to avoid transforming nonterminating executions into executions that go wrong. This issue is not addressed in the work of Kanade et al. (2006), nor in the original proof of correctness of LCM by Knoop et al. (1994): both consider only terminating executions.

## 10. Conclusion

We presented a validation algorithm for Lazy Code Motion and its mechanized proof of correctness. The validation algorithm is significantly simpler than LCM itself: the latter uses four dataflow analyses, while our validator uses only one (a standard available expression analysis) complemented with an anticipability check (a simple traversal of the CFG). This relative simplicity of the algorithm, in turn, results in a mechanized proof of correctness that remains manageable after careful proof engineering. Therefore, this work gives a good example of the benefits of the verified validator approach compared with compiler verification.

We have also shown preliminary evidence that the verified validator can be re-used for other optimizations: not only other forms of redundancy elimination, but also unrelated optimizations such as constant propagation and instruction strength reduction. More work is needed to address the validation of advanced global optimizations such as global value numbering, but the decomposition of our validator and its proof into a generic framework and an LCM-specific part looks like a first step in this direction.

Even though lazy code motion moves instructions across loops, it is still a structure-preserving transformation. Future work includes extending the verified validation approach to optimizations that modify the structure of loops, such as software pipelining, loop jamming, or loop interchange.

## Acknowledgments

We would like to thank Benoît Razet, Damien Doligez, and the anonymous reviewers for their helpful comments and suggestions for improvements.

This work was supported by Agence Nationale de la Recherche, grant number ANR-05-SSIA-0019.

## References

- Clark W. Barret, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.
- Yves Bertot and Pierre Castérán. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.
- Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *SEFM ’05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 200–209. IEEE Computer Society, 2005.
- Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *PLDI ’98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 1–14. ACM, 1998.
- Coq development team. The Coq proof assistant. Software and documentation available at <http://coq.inria.fr/>, 1989–2009.
- Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis, 11th Int. Symp., SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 212–227. Springer, 2004.
- Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.
- Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *SEFM ’06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.
- Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
- Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Programming Languages Design and Implementation 1992*, pages 224–234. ACM Press, 1992.
- Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.
- Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.
- Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Programming Language Design and Implementation 2003*, pages 220–231. ACM Press, 2003.
- Xavier Leroy. A formally verified compiler back-end. arXiv:0902.2137 [cs]. Submitted, July 2008.
- Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

- Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.
- Xavier Leroy et al. The CompCert verified compiler. Development available at <http://compcert.inria.fr>, 2004–2009.
- Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communication of the ACM*, 22(2):96–103, 1979.
- George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.
- Amir Pnueli, Ofer Shtrichman, and Michael Siegel. The code validation tool (CVT) – automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2:192–201, 1998a.
- Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998b.
- Martin Rinard and Darko Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification*, 1999.
- Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
- Bernhard Steffen. Property-oriented expansion. In *Static Analysis, Third International Symposium, SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 1996.
- Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.
- Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.
- Lenore Zuck, Amir Pnueli, and Raya Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann institute of Science, 2001.
- Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

# A Simple, Verified Validator for Software Pipelining

## (verification pearl)

Jean-Baptiste Tristan

INRIA Paris-Rocquencourt  
B.P. 105, 78153 Le Chesnay, France  
jean-baptiste.tristan@inria.fr

Xavier Leroy

INRIA Paris-Rocquencourt  
B.P. 105, 78153 Le Chesnay, France  
xavier.leroy@inria.fr

## Abstract

Software pipelining is a loop optimization that overlaps the execution of several iterations of a loop to expose more instruction-level parallelism. It can result in first-class performance characteristics, but at the cost of significant obfuscation of the code, making this optimization difficult to test and debug. In this paper, we present a translation validation algorithm that uses symbolic evaluation to detect semantics discrepancies between a loop and its pipelined version. Our algorithm can be implemented simply and efficiently, is provably sound, and appears to be complete with respect to most modulo scheduling algorithms. A conclusion of this case study is that it is possible and effective to use symbolic evaluation to reason about loop transformations.

**Categories and Subject Descriptors** D.2.4 [*Software Engineering*]: Software/Program Verification - Correctness proofs; D.3.4 [*Programming Languages*]: Processors - Optimization

**General Terms** Languages, Verification, Algorithms

**Keywords** Software pipelining, translation validation, symbolic evaluation, verified compilers

## 1. Introduction

*There is one last technique in the arsenal of the software optimizer that may be used to make most machines run at tip top speed. It can also lead to severe code bloat and may make for almost unreadable code, so should be considered the last refuge of the truly desperate. However, its performance characteristics are in many cases unmatched by any other approach, so we cover it here. It is called software pipelining [...]*

Apple Developer Connection<sup>1</sup>

Software pipelining is an advanced instruction scheduling optimization that exposes considerable instruction-level parallelism by overlapping the execution of several iterations of a loop. It produces smaller code and eliminates more pipeline stalls than merely

unrolling the loop then performing acyclic scheduling. Software pipelining is implemented in many production compilers and described in several compiler textbooks [1, section 10.5] [2, chapter 20] [16, section 17.4]. In the words of the rather dramatic quote above, “truly desperate” programmers occasionally perform software pipelining by hand, especially on multimedia and signal processing kernels.

Starting in the 1980’s, many clever algorithms were designed to produce efficient software pipelines and implement them either on stock hardware or by taking advantage of special features such as those of the IA64 architecture. In this paper, we are not concerned about the performance characteristics of these algorithms, but rather about their *semantic correctness*: does the generated, software-pipelined code compute the same results as the original code? As with all advanced compiler optimizations, and perhaps even more so here, mistakes happen in the design and implementation of software pipelining algorithms, causing incorrect code to be generated from correct source programs. The extensive code rearrangement performed by software pipelining makes visual inspection of the generated code ineffective; the additional boundary conditions introduced make exhaustive testing difficult. For instance, after describing a particularly thorny issue, Rau *et al.* [23] note that

*The authors are indirectly aware of at least one computer manufacturer whose attempts to implement modulo scheduling, without having understood this issue, resulted in a compiler which generated incorrect code.*

Translation validation is a systematic technique to detect (at compile-time) semantic discrepancies introduced by buggy compiler passes or desperate programmers who optimize by hand, and to build confidence in the result of a compilation run or manual optimization session. In this approach, the programs before and after optimization are fed to a *validator* (a piece of software distinct from the optimizer), which tries to establish that the two programs are semantically equivalent; if it fails, compilation is aborted or continues with the unoptimized code, discarding the incorrect optimization. As invented by Pnueli *et al.* [20], translation validation proceeds by generation of verification conditions followed by model-checking or automated theorem proving [19, 29, 28, 3, 8]. An alternate approach, less general but less costly in validation time, relies on combinations of symbolic evaluation and static analysis [18, 24, 6, 26, 27]. The VCGen/theorem-proving approach was applied to software pipelining by Leviathan and Pnueli [14]. It was long believed that symbolic evaluation with static analyses was too weak to validate aggressive loop optimizations such as software pipelining.

In this paper, we present a novel translation validation algorithm for software pipelining, based on symbolic evaluation. This algorithm is surprisingly simple, proved to be sound (most of the

<sup>1</sup> [http://developer.apple.com/hardwaredrivers/ve/software\\_pipelining.html](http://developer.apple.com/hardwaredrivers/ve/software_pipelining.html)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL’10, January 17–23, 2010, Madrid, Spain.  
Copyright © 2010 ACM 978-1-60558-479-9/10/01...\$10.00

Original loop body $\mathcal{B}$ :	Prolog $\mathcal{P}$ :	Steady state $\mathcal{S}$ :	Epilogue $\mathcal{E}$ :
<pre>x := load(float64, p); y := x * c; store(float64, p, y); p := p + 8; i := i + 1;</pre>	<pre>p1 := p; p2 := p; x1 := x; x2 := x; x1 := load(float64, p1); p2 := p1 + 8; x2 := load(float64, p2); y := x1 * c; i := i + 2;</pre>	<pre>store(float64, p1, y); p1 := p2 + 8; y := x2 * c; x1 := load(float64, p1); store(float64, p2, y); p2 := p1 + 8; y := x1 * c; x2 := load(float64, p2); i := i + 2;</pre>	<pre>store(float64, p1, y); y := x2 * c; store(float64, p2, y); x := x2; p := p2;</pre>

Figure 1. An example of software pipelining

proof was mechanized using the Coq proof assistant), and informally argued to be complete with respect to a wide class of software pipelining optimizations.

The formal verification of a validator consists in mechanically proving that if it returns `true`, its two input programs do behave identically at run time. This is a worthwhile endeavor for two reasons. First, it brings additional confidence that the results of validation are trustworthy. Second, it provides an attractive alternative to formally verifying the soundness of the optimization algorithm itself: the validator is often smaller and easier to verify than the optimization it validates [26]. The validation algorithm presented in this paper grew out of the desire to add a software pipelining pass to the CompCert high-assurance C compiler [11]. Its formal verification in Coq is not entirely complete at the time of this writing, but appears within reach given the simplicity of the algorithm.

The remainder of this paper is organized as follows. Section 2 recalls the effect of software pipelining on the shape of loops. Section 3 outlines the basic principle of our validator. Section 4 defines the flavor of symbolic evaluation that it uses. Section 5 presents the validation algorithm. Its soundness is proved in section 6; its completeness is discussed in section 7. The experience gained on a prototype implementation is discussed in section 8. Section 9 discusses related work, and is followed by conclusions and perspectives in section 10.

## 2. Software pipelining

From the bird’s eye, software pipelining is performed in three steps.

**Step 1** Select one inner loop for pipelining. Like in most previous work, we restrict ourselves to simple counted loops of the form

```
i := 0;
while (i < N) { B }
```

We assume that the loop bound  $N$  is a loop-invariant variable, that the loop body  $\mathcal{B}$  is a basic block (no conditional branches, no function calls), and that the loop index  $i$  is incremented exactly once in  $\mathcal{B}$ . The use of structured control above is a notational convenience: in reality, software pipelining is performed on a flow graph representation of control (CFG), and step 1 actually isolates a sub-graph of the CFG comprising a proper loop of the form above, as depicted in figure 2(a).

**Step 2** Next, the *software pipeliner* is called. It takes as input the loop body  $\mathcal{B}$  and the loop index  $i$ , and produces a 5-tuple  $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$  as result.  $\mathcal{P}$ ,  $\mathcal{S}$  and  $\mathcal{E}$  are sequences of non-branching instructions;  $\mu$  and  $\delta$  are positive integers.

- $\mathcal{S}$  is the new loop body, also called the *steady state* of the pipeline.
- $\mathcal{P}$  is the loop prolog: a sequence of instructions that fills the pipeline until it reaches the steady state.

- $\mathcal{E}$  is the loop epilog: a sequence of instructions that drains the pipeline, finishing the computations that are still in progress at the end of the steady state.
- $\mu$  is the minimum number of iterations that must be performed to be able to use the pipelined loop.
- $\delta$  is the amount of unrolling that has been performed on the steady state. In other words, one iteration of  $\mathcal{S}$  corresponds to  $\delta$  iterations of the original loop  $\mathcal{B}$ .

The prolog  $\mathcal{P}$  is assumed to increment the loop index  $i$  by  $\mu$ ; the steady state  $\mathcal{S}$ , by  $\delta$ ; and the epilogue  $\mathcal{E}$ , not at all.

The software pipelining algorithms that we have in mind here are combinations of *modulo scheduling* [22, 7, 15, 23, 4] followed by *modulo variable expansion* [10]. However, the presentation above seems general enough to accommodate other scheduling algorithms.

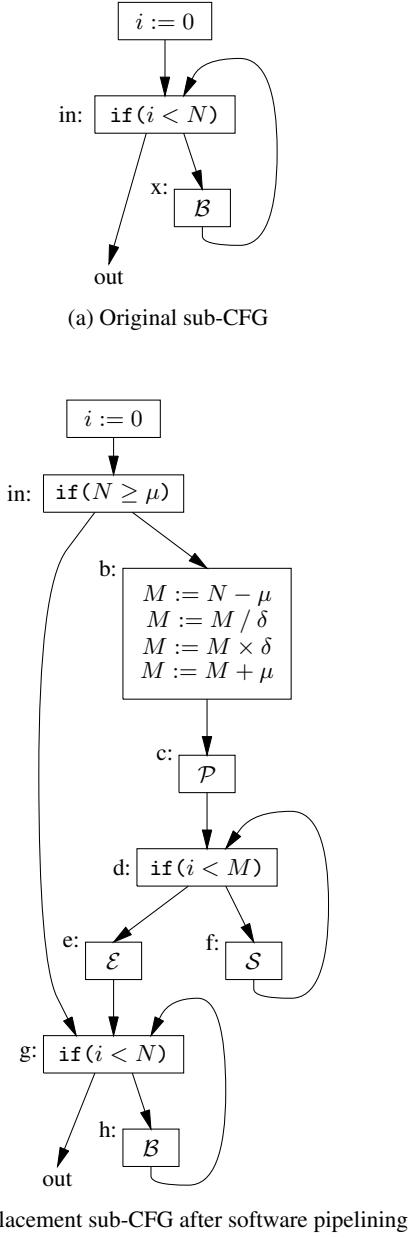
Figure 1 illustrates one run of a software pipeliner. The schedule obtained by modulo scheduling performs, at each iteration, the  $i^{\text{th}}$  *store* and increment of  $p$ , the  $(i+1)^{\text{th}}$  multiplication by  $c$ , and the  $(i+2)^{\text{th}}$  *load*. To avoid read-after-write hazards, modulo variable expansion was performed, unrolling the schedule by a factor of 2, and replacing variables  $p$  and  $x$  by two variables each,  $p1/p2$  and  $x1/x2$ , which are defined in an alternating manner. These pairs of variables are initialized from  $p$  and  $x$  in the prolog; the epilogue sets  $p$  and  $x$  back to their final values. The unrolling factor  $\delta$  is therefore 2. Likewise, the minimum number of iterations is  $\mu = 2$ .

**Step 3** Finally, the original loop is replaced by the following pseudo-code:

```
i := 0;
if (N ≥ μ) {
    M := ((N - μ)/δ) × δ + μ;
    P
    while (i < M) { S }
    E
}
while (i < N) { B }
```

In CFG terms, this amounts to replacing the sub-graph shown at the top of figure 2 by the one shown at the bottom.

The effect of the code after pipelining is as follows. If the number of iterations  $N$  is less than  $\mu$ , a copy of the original loop is executed. Otherwise, we execute the prolog  $\mathcal{P}$ , then iterate the steady state  $\mathcal{S}$ , then execute the epilogue  $\mathcal{E}$ , and finally fall through the copy of the original loop. Since  $\mathcal{P}$  and  $\mathcal{S}$  morally correspond to  $\mu$  and  $\delta$  iterations of the original loop, respectively,  $\mathcal{S}$  is iterated  $n$  times where  $n$  is the largest integer such that  $\mu + n\delta \leq N$ , namely  $n = (N - \mu)/\delta$ . In terms of the loop index  $i$ , this corresponds to the upper limit  $M$  shown in the pseudo-code above. The copy of the original loop executes the remaining  $N - M$  iterations.



**Figure 2.** The effect of software pipelining on a control-flow graph

Steps 1, 2 and 3 can of course be repeated once per inner loop eligible for pipelining. In the remainder of this paper, we focus on the transformation of a single loop.

### 3. Validation a posteriori

In a pure translation validation approach, the validator would receive as inputs the code before and after software pipelining and would be responsible for establishing the correctness of all three steps of pipelining. We found it easier to concentrate translation validation on step 2 only (the construction of the components of the pipelined loop) and revert to traditional compiler verification for steps 1 and 3 (the assembling of these components to form the transformed code). In other words, the validator we set out to build

takes as arguments the input  $(i, N, \mathcal{B})$  and output  $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$  of the software pipeliner (step 2). It is responsible for establishing conditions sufficient to prove semantic preservation between the original and transformed codes (top and bottom parts of figure 2).

What are these sufficient conditions and how can a validator establish them? To progress towards an answer, think of complete unrollings of the original and pipelined loops. We see that the run-time behavior of the original loop is equivalent to that of  $\mathcal{B}^N$ , that is,  $N$  copies of  $\mathcal{B}$  executed in sequence. Likewise, the generated code behaves either like  $\mathcal{B}^N$  if  $N < \mu$ , or like

$$\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}$$

if  $N \geq \mu$ , where

$$\kappa(N) \stackrel{\text{def}}{=} (N - \mu)/\delta$$

$$\rho(N) \stackrel{\text{def}}{=} N - \mu - \delta \times \kappa(N)$$

The verification problem therefore reduces to establishing that, for all  $N$ , the two basic blocks

$$X_N \stackrel{\text{def}}{=} \mathcal{B}^N \quad \text{and} \quad Y_N \stackrel{\text{def}}{=} \mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}$$

are semantically equivalent: if both blocks are executed from the same initial state (same memory state, same values for variables), they terminate with the same final state (up to the values of temporary variables introduced during scheduling and unused later).

For a given value of  $N$ , *symbolic evaluation* provides a simple, effective way to ensure this semantic equivalence between  $X_N$  and  $Y_N$ . In a nutshell, symbolic evaluation is a form of denotational semantics for basic blocks where the values of variables  $x, y, \dots$  at the end of the block are determined as symbolic expressions of their values  $x_0, y_0, \dots$  at the beginning of the block. For example, the symbolic evaluation of the block  $B \stackrel{\text{def}}{=} y := x + 1; x := y \times 2$  is

$$\alpha(B) = \begin{cases} x &\mapsto (x_0 + 1) \times 2 \\ y &\mapsto x_0 + 1 \\ v &\mapsto v_0 \text{ for all } v \notin \{x, y\} \end{cases}$$

(This is a simplified presentation of symbolic evaluation; section 4 explains how to extend it to handle memory accesses, potential run-time errors, and the fresh variables introduced by modulo variable expansion during pipelining.)

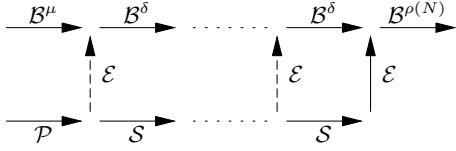
The fundamental property of symbolic evaluation is that if two blocks have identical symbolic evaluations, whenever they are executed in the same but arbitrary initial state, they terminate in the same final state. Moreover, symbolic evaluation is insensitive to reordering of independent instructions, making it highly suitable to reason over scheduling optimizations [26]. Therefore, for any given  $N$ , our validator can check that  $\alpha(X_N) = \alpha(Y_N)$ ; this suffices to guarantee that  $X_N$  and  $Y_N$  are semantically equivalent.

If  $N$  were a compile-time constant, this would provide us with a validation algorithm. However,  $N$  is in general a run-time quantity, and statically checking  $\alpha(X_N) = \alpha(Y_N)$  for all  $N$  is not decidable. The key discovery of this paper is that this undecidable property  $\forall N, \alpha(X_N) = \alpha(Y_N)$  is implied by (and, in practice, equivalent to) the following two decidable conditions:

$$\alpha(\mathcal{E}; \mathcal{B}^\delta) = \alpha(\mathcal{S}; \mathcal{E}) \tag{1}$$

$$\alpha(\mathcal{B}^\mu) = \alpha(\mathcal{P}; \mathcal{E}) \tag{2}$$

The programmer's intuition behind condition 1 is the following: in a correct software pipeline, if enough iterations remain to be performed, it should always be possible to choose between 1) execute the steady state  $\mathcal{S}$  one more time, then leave the pipelined loop and execute  $\mathcal{E}$ ; or 2) leave the pipelined loop immediately by executing  $\mathcal{E}$ , then run  $\delta$  iterations of the original loop body  $\mathcal{B}$ . (See figure 3.) Condition 2 is even easier to justify: if the number



**Figure 3.** A view of an execution of the loop, before (top) and after (bottom) pipelining. The dashed vertical arrows provide intuitions for equations 1 and 2.

of executions is exactly  $\mu$ , the original code performs  $B^\mu$  and the pipelined code performs  $P; \mathcal{E}$ .

The translation validation algorithm that we define in section 5 is based on checking minor variations of conditions 1 and 2, along with some additional conditions on the evolutions of variable  $i$  (also checked using symbolic evaluation). Before presenting this algorithm, proving that it is sound, and arguing that it is complete in practice, we first need to define symbolic evaluation more precisely and study its algebraic structure.

#### 4. Symbolic evaluation with observables

In this section, we formally define the form of symbolic evaluation used in our validator. Similar forms of symbolic evaluation are at the core of several other translation validators, such as those of Necula [18], Rival [24], and Tristan and Leroy [26].

##### 4.1 The intermediate language

We perform symbolic evaluation over the following simple language of basic blocks:

Non-branching instructions:

$$\begin{aligned} I ::= & r := r' && \text{variable-variable move} \\ | & r := \text{op}(op, \vec{r}) && \text{arithmetic operation} \\ | & r := \text{load}(\kappa, r_{addr}) && \text{memory load} \\ | & \text{store}(\kappa, r_{addr}, r_{val}) && \text{memory store} \end{aligned}$$

Basic blocks:

$$B ::= I_1; \dots; I_n \quad \text{instruction sequences}$$

Here,  $r$  ranges over variable names (a.k.a. pseudo-registers or temporaries);  $op$  ranges over arithmetic operators (such as “load integer constant  $n$ ” or “add integer immediate  $n$ ” or “float multiply”); and  $\kappa$  stands for a memory quantity (such as “8-bit signed integer” or “64-bit float”).

##### 4.2 Symbolic states

The symbolic evaluation of a block produces a pair  $(\varphi, \sigma)$  of a mapping  $\varphi$  from resources to terms, capturing the relationship between the initial and final values of resources, and a set of terms  $\sigma$ , recording the computations performed by the block.

Resources:

$$\rho ::= r \mid \text{Mem}$$

Terms:

$$\begin{aligned} t ::= & \rho_0 && \text{initial value of } \rho \\ | & \text{Op}(op, \vec{t}) && \text{result of an operation} \\ | & \text{Load}(\kappa, t_{addr}, t_m) && \text{result of a load} \\ | & \text{Store}(\kappa, t_{addr}, t_{val}, t_m) && \text{effect of a store} \end{aligned}$$

Resource maps:

$$\varphi ::= \rho \mapsto t \quad \text{finite map}$$

Computations performed:

$$\sigma ::= \{t_1; \dots; t_n\} \quad \text{finite set}$$

Symbolic states:  
 $A ::= (\varphi, \sigma)$

Resource maps symbolically track the values of variables and the memory state, represented like a ghost variable  $\text{Mem}$ . A resource  $\rho$  that is not in the domain of a map  $\varphi$  is considered mapped to  $\rho_0$ . Memory states are represented by terms  $t_m$  built out of  $\text{Mem}_0$  and  $\text{Store}(\dots)$  constructors. For example, the following block

$$\text{store}(\kappa, x, y); z := \text{load}(\kappa, w)$$

symbolically evaluates to the following resource map:

$$\begin{aligned} \text{Mem} &\mapsto \text{Store}(\kappa, x_0, y_0, \text{Mem}_0) \\ z &\mapsto \text{Load}(\kappa, w_0, \text{Store}(\kappa, x_0, y_0, \text{Mem}_0)) \end{aligned}$$

Resource maps describe the final state after execution of a basic block, but do not capture all intermediate operations performed. This is insufficient if some intermediate operations can fail at run-time (e.g. integer divisions by zero or loads from a null pointer). Consider for example the two blocks

$$\begin{aligned} y &:= x \quad \text{and} \quad y := \text{load(int32}, x); \\ y &:= x \end{aligned}$$

They have the same resource map, namely  $y \mapsto x_0$ , yet the right-most block right can cause a run-time error if the pointer  $x$  is null, while the leftmost block never fails. Therefore, resource maps do not suffice to check that two blocks have the same semantics; they must be complemented by sets  $\sigma$  of symbolic terms, recording all the computations that are performed by the blocks, even if these computations do not contribute directly to the final state. Continuing the example above, the leftmost block has  $\sigma = \emptyset$  (a move is not considered as a computation, since it cannot fail) while the right-most block has  $\sigma' = \{\text{Load(int32}, x_0, \text{Mem}_0)\}$ . Since these two sets differ, symbolic evaluation reveals that the two blocks above are not semantically equivalent with respect to run-time errors.

##### 4.3 Composition

A resource map  $\varphi$  can be viewed as a parallel substitution of symbolic terms for resources: the action of a map on a term  $t$  is the term  $\varphi(t)$  obtained by replacing all occurrences of  $\rho_0$  in  $t$  by the term associated with  $\rho$  in  $\varphi$ . Therefore, the reverse composition  $\varphi_1; \varphi_2$  of two maps is, classically,

$$\varphi_1; \varphi_2 \stackrel{\text{def}}{=} \{\rho \mapsto \varphi_1(\varphi_2(\rho)) \mid \rho \in \text{Dom}(\varphi_1) \cup \text{Dom}(\varphi_2)\} \quad (3)$$

The reverse composition operator extends naturally to abstract states:

$$(\varphi_1, \sigma_1); (\varphi_2, \sigma_2) \stackrel{\text{def}}{=} (\varphi_1; \varphi_2, \sigma_1 \cup \{\varphi_1(t) \mid t \in \sigma_2\}) \quad (4)$$

Composition is associative and admits the identity abstract state  $\varepsilon \stackrel{\text{def}}{=} (\emptyset, \emptyset)$  as neutral element.

##### 4.4 Performing symbolic evaluation

The symbolic evaluation  $\alpha(I)$  of a single instruction  $I$  is the abstract state defined by

$$\begin{aligned} \alpha(r := r') &= (r \mapsto r'_0, \emptyset) \\ \alpha(r := \text{op}(op, \vec{r})) &= (r \mapsto t, \{t\}) \\ &\quad \text{where } t = \text{Op}(op, \vec{r}_0) \\ \alpha(r := \text{load}(\kappa, r')) &= (r \mapsto t, \{t\}) \\ &\quad \text{where } t = \text{Load}(\kappa, r'_0, \text{Mem}_0) \\ \alpha(\text{store}(\kappa, r, r')) &= (\text{Mem} \mapsto t_m, \{t_m\}) \\ &\quad \text{where } t_m = \text{Store}(\kappa, r_0, r'_0, \text{Mem}_0) \end{aligned}$$

Symbolic evaluation then extends to basic blocks, by composing the evaluations of individual instructions:

$$\alpha(I_1; \dots; I_n) \stackrel{\text{def}}{=} \alpha(I_1); \dots; \alpha(I_n) \quad (5)$$

By associativity, it follows that symbolic evaluation distributes over concatenation of basic blocks:

$$\alpha(B_1; B_2) = \alpha(B_1); \alpha(B_2) \quad (6)$$

#### 4.5 Comparing symbolic states

We use two notions of equivalence between symbolic states. The first one, written  $\sim$ , is defined as strict syntactic equality:<sup>2</sup>

$$(\varphi, \sigma) \sim (\varphi', \sigma') \text{ if and only if } \forall \rho, \varphi(\rho) = \varphi'(\rho) \text{ and } \sigma = \sigma' \quad (7)$$

The relation  $\sim$  is therefore an equivalence relation, and is compatible with composition:

$$A_1 \sim A_2 \implies A_1; A \sim A_2; A \quad (8)$$

$$A_1 \sim A_2 \implies A; A_1 \sim A; A_2 \quad (9)$$

We need a coarser equivalence between symbolic states to account for the differences in variable usage between the code before and after software pipelining. Recall that pipelining may perform modulo variable expansion to avoid read-after-write hazards. This introduces fresh temporary variables in the pipelined code. These fresh variables show up in symbolic evaluations and prevent strict equivalence  $\sim$  from holding. Consider:

$$\begin{aligned} x := \text{op}(op, x) \quad &\text{and} \quad x' := \text{op}(op, x) \\ &x := x' \end{aligned}$$

These two blocks have different symbolic evaluations (in the sense of the  $\sim$  equivalence), yet should be considered as semantically equivalent if the temporary  $x'$  is unused later.

Let  $\theta$  be a finite set of variables: the *observable* variables. (In our validation algorithm, we take  $\theta$  to be the set of all variable mentioned in the original code before pipelining; the fresh temporaries introduced by modulo variable expansion are therefore not in  $\theta$ .) Equivalence between symbolic states up to the observables  $\theta$  is written  $\approx_\theta$  and defined as

$$(\varphi, \sigma) \approx_\theta (\varphi', \sigma') \text{ if and only if}$$

$$\forall \rho \in \theta \cup \{\text{Mem}\}, \varphi(\rho) = \varphi'(\rho) \text{ and } \sigma = \sigma' \quad (10)$$

The relation  $\approx_\theta$  is an equivalence relation, coarser than  $\sim$ , and compatible with composition on the right:

$$A_1 \sim A_2 \implies A_1 \approx_\theta A_2 \quad (11)$$

$$A_1 \approx_\theta A_2 \implies A; A_1 \approx_\theta A; A_2 \quad (12)$$

However, it is not, in general, compatible with composition on the left. Consider an abstract state  $A$  that maps a variable  $x \in \theta$  to a term  $t$  containing an occurrence of  $y \notin \theta$ . Further assume  $A_1 \approx_\theta A_2$ . The composition  $A_1; A$  maps  $x$  to  $A_1(t)$ ; likewise,  $A_2; A$  maps  $x$  to  $A_2(t)$ . Since  $y \notin \theta$ , we can have  $A_1(y) \neq A_2(y)$  and therefore  $A_1(t) \neq A_2(t)$  without violating the hypothesis  $A_1 \approx_\theta A_2$ . Hence,  $A_1; A \approx_\theta A_2; A$  does not hold in general.

To address this issue, we restrict ourselves to symbolic states  $A$  that are *contained* in the set  $\theta$  of observables. We say that a

<sup>2</sup> We could relax this definition in two ways. One is to compare symbolic terms up to a decidable equational theory capturing algebraic identities such as  $x \times 2 = x + x$ . This enables the validation of e.g. instruction strength reduction [18, 24]. Another relaxation is to require  $s' \subseteq s$  instead of  $s = s'$ , enabling the transformed basic block to perform fewer computations than the original block (e.g. dead code elimination). Software pipelining being a purely lexical transformation that changes only the placement of computations but not their nature nor their number, these two relaxations are not essential and we stick with strict syntactic equality for the time being.

Semantic interpretation of arithmetic and memory operations:

$$\begin{aligned} \overline{\text{op}} &: \text{list val} \rightarrow \text{option val} \\ \overline{\text{load}} &: \text{quantity} \times \text{val} \times \text{mem} \rightarrow \text{option val} \\ \overline{\text{store}} &: \text{quantity} \times \text{val} \times \text{val} \times \text{mem} \rightarrow \text{option mem} \end{aligned}$$

Transition semantics for instructions:

$$\begin{array}{c} r := r' : (R, M) \rightarrow (R[r \leftarrow R(r')], M) \\ \overline{\text{op}}(R(\vec{r})) = [v] \\ \hline r := \text{op}(op, \vec{r}) : (R, M) \rightarrow (R[r \leftarrow v], M) \end{array}$$

$$\begin{array}{c} \overline{\text{load}}(\kappa, R(r_{addr}), M) = [v] \\ \hline r := \text{load}(\kappa, r_{addr}) : (R, M) \rightarrow (R[r \leftarrow v], M) \end{array}$$

$$\begin{array}{c} \overline{\text{store}}(\kappa, R(r_{addr}), R(r_{val}), M) = [M'] \\ \hline \text{store}(\kappa, r_{addr}, r_{val}) : (R, M) \rightarrow (R, M') \end{array}$$

Transition semantics for basic blocks:

$$\begin{array}{c} e : S \xrightarrow{*} S \\ \hline \begin{array}{c} I : S \rightarrow S' \quad B : S' \xrightarrow{*} S'' \\ I.B : S \xrightarrow{*} S'' \end{array} \end{array}$$

Figure 4. Operational semantics for basic blocks

symbolic term  $t$  is contained in  $\theta$ , and write  $t \sqsubseteq \theta$ , if all variables  $x$  appearing in  $t$  belong to  $\theta$ . We extend this notion to symbolic states as follows:

$$(\varphi, \sigma) \sqsubseteq \theta \text{ if and only if}$$

$$\forall \rho \in \theta \cup \{\text{Mem}\}, \varphi(\rho) \sqsubseteq \theta \text{ and } \forall t \in \sigma, t \sqsubseteq \theta \quad (13)$$

It is easy to see that  $\approx_\theta$  is compatible with composition on the left if the right symbolic state is contained in  $\theta$ :

$$A_1 \approx_\theta A_2 \wedge A \sqsubseteq \theta \implies A_1; A \approx_\theta A_2; A \quad (14)$$

The containment relation enjoys additional useful properties:

$$A_1 \sqsubseteq \theta \wedge A_1 \approx_\theta A_2 \implies A_2 \sqsubseteq \theta \quad (15)$$

$$A_1 \sqsubseteq \theta \wedge A_2 \sqsubseteq \theta \implies (A_1; A_2) \sqsubseteq \theta \quad (16)$$

#### 4.6 Semantic soundness

The fundamental property of symbolic evaluation is that if  $\alpha(B_1) \approx_\theta \alpha(B_2)$ , the two blocks  $B_1$  and  $B_2$  have the same run-time behavior, in a sense that we now make precise.

We equip our language of basic blocks with the straightforward operational semantics shown in figure 4. The behavior of arithmetic and memory operations is axiomatized as functions  $\overline{\text{op}}$ ,  $\overline{\text{load}}$  and  $\overline{\text{store}}$  that return “option” types to model run-time failures: the result  $\emptyset$  (pronounced “none”) denotes failure; the result  $[x]$  (pronounced “some  $x$ ”) denotes success. In our intended application, these interpretation functions are those of the RTL intermediate language of the CompCert compiler [12, section 6.1] and of its memory model [13].

The semantics for a basic block  $B$  is, then, given as the relation  $B : S \xrightarrow{*} S'$ , sometimes also written  $S \xrightarrow{B} S'$ , where  $S$  is the initial state and  $S'$  the final state. States are pairs  $(R, M)$  of a variable state  $R$ , mapping variables to values, and a memory state  $M$ .

We say that two execution states  $S = (R, M)$  and  $S' = (R', M')$  are equivalent up to the observables  $\theta$ , and write  $S \cong_\theta S'$ , if  $M = M'$  and  $R(r) = R'(r)$  for all  $r \in \theta$ .

**THEOREM 1** (Soundness of symbolic evaluation). *Let  $B_1$  and  $B_2$  be two basic blocks. Assume that  $\alpha(B_1) \approx_\theta \alpha(B_2)$  and  $\alpha(B_2) \sqsubseteq$*

$\theta$ . If  $B_1 : S \xrightarrow{*} S'$  and  $S \cong_{\theta} T$ , there exists  $T'$  such that  $B_2 : T \xrightarrow{*} T'$  and  $S' \cong_{\theta} T'$ .

We omit the proof, which is similar to that of Lemma 3 in [26].

## 5. The validation algorithm

We can now piece together the intuitions from section 3 and the definitions from section 4 to obtain the following validation algorithm:

$$\begin{aligned} \text{validate } (i, N, \mathcal{B}) & (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta) \quad \theta = \\ & \alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{E}) \quad (\text{A}) \\ & \wedge \alpha(\mathcal{E}; \mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{S}; \mathcal{E}) \quad (\text{B}) \\ & \wedge \alpha(\mathcal{B}) \sqsubseteq \theta \quad (\text{C}) \\ & \wedge \alpha(\mathcal{B})(i) = \text{Op}(\text{addi}(1), i_0) \quad (\text{D}) \\ & \wedge \alpha(\mathcal{P})(i) = \alpha(\mathcal{B}^{\mu})(i) \quad (\text{E}) \\ & \wedge \alpha(\mathcal{S})(i) = \alpha(\mathcal{B}^{\delta})(i) \quad (\text{F}) \\ & \wedge \alpha(\mathcal{E})(i) = i_0 \quad (\text{G}) \\ & \wedge \alpha(\mathcal{B})(N) = \alpha(\mathcal{P})(N) = \alpha(\mathcal{S})(N) = \alpha(\mathcal{E})(N) = N_0 \quad (\text{H}) \end{aligned}$$

The inputs of the validator are the components of the loops before  $(i, N, \mathcal{B})$  and after  $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$  pipelining, as well as the set  $\theta$  of observables. Checks A and B correspond to equations (1) and (2) of section 3, properly reformulated in terms of equivalence up to observables. Check C makes sure that the set  $\theta$  of observables includes at least the variables appearing in the symbolic evaluation of the original loop body  $\mathcal{B}$ . Checks D to G ensure that the loop index  $i$  is incremented the way we expect it to be: by one in  $\mathcal{B}$ , by  $\mu$  in  $\mathcal{P}$ , by  $\delta$  in  $\mathcal{S}$ , and not at all in  $\mathcal{E}$ . (We write  $\alpha(\mathcal{B})(i)$  to refer to the symbolic term associated to  $i$  by the resource map part of the symbolic state  $\alpha(\mathcal{B})$ .) Finally, check H makes sure that the loop limit  $N$  is invariant in both loops.

The algorithm above has low computational complexity. Using a hash-consed representation of symbolic terms, the symbolic evaluation  $\alpha(\mathcal{B})$  of a block  $\mathcal{B}$  containing  $n$  instructions can be computed in time  $O(n \log n)$ : for each of the  $n$  instructions, the algorithm performs one update on the resource map, one insertion in the constraint set, and one hash-consing operation, all of which can be done in logarithmic time. (The resource map and the constraint set have sizes at most  $n$ .) Likewise, the equivalence and containment relations ( $\approx_{\theta}$  and  $\sqsubseteq$ ) can be decided in time  $O(n \log n)$ . Finally, note that this  $O(n \log n)$  complexity holds even if the algorithm is implemented within the Coq specification language, using persistent, purely functional data structures.

The running time of the validation algorithm is therefore  $O(n \log n)$  where  $n = \max(\mu|\mathcal{B}|, \delta|\mathcal{B}|, |\mathcal{P}|, |\mathcal{S}|, |\mathcal{E}|)$  is proportional to the size of the loop after pipelining. Theoretically, software pipelining can increase the size of the loop quadratically. In practice, we expect the external implementation of software pipelining to avoid this blowup and produce code whose size is linear in that of the original loop. In this case, the validator runs in time  $O(n \log n)$  where  $n$  is the size of the original loop.

## 6. Soundness proof

In this section, we prove the soundness of the validator: if it returns “true”, the software pipelined loop executes exactly like the original loop. The proof proceeds in two steps: first, we show that the basic blocks  $X_N$  and  $Y_N$  obtained by unrolling the two loops have the same symbolic evaluation and therefore execute identically; then, we prove that the concrete execution of the two loops in the CFG match.

### 6.1 Soundness with respect to the unrolled loops

We first show that the finitary conditions checked by the validator imply the infinitary equivalences between the symbolic evaluations of  $X_N$  and  $Y_N$  sketched in section 3.

LEMMA 2. If  $\text{validate } (i, N, \mathcal{B}) (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta) \theta$  returns `true`, then, for all  $n$ ,

$$\alpha(\mathcal{B}^{\mu+n\delta}) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}) \quad (17)$$

$$\alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}) \sqsubseteq \theta \quad (18)$$

$$\alpha(\mathcal{B}^{\mu+n\delta})(i) = \alpha(\mathcal{P}; \mathcal{S}^n)(i) = \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E})(i) \quad (19)$$

**Proof:** By hypothesis, we know that the properties A to H checked by the validator hold. First note that

$$\forall n, \quad \alpha(\mathcal{B}^n) \sqsubseteq \theta \quad (20)$$

as a consequence of check C and property (16).

Conclusion (17) is proved by induction on  $n$ . The base case  $n = 0$  reduces to  $\alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{E})$ , which is ensured by check A. For the inductive case, assume  $\alpha(\mathcal{B}^{\mu+n\delta}) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E})$ .

$$\begin{aligned} \alpha(\mathcal{B}^{\mu+(n+1)\delta}) & \approx_{\theta} \alpha(\mathcal{B}^{\mu+n\delta}); \alpha(\mathcal{B}^{\delta}) \\ & \quad (\text{by distributivity (6)}) \\ & \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}); \alpha(\mathcal{B}^{\delta}) \\ & \quad (\text{by left compatibility (14) and ind. hyp.}) \\ & \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{E}; \mathcal{B}^{\delta}) \\ & \quad (\text{by distributivity (6)}) \\ & \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{S}; \mathcal{E}) \\ & \quad (\text{by check B, right compatibility (12), and (20)}) \\ & \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^{n+1}; \mathcal{E}) \\ & \quad (\text{by distributivity (6)}) \end{aligned}$$

Conclusion (18) follows from (17), (20), and property (15). Conclusion (19) follows from checks E, F and G by induction on  $n$ .  $\square$

THEOREM 3. Assume that  $\text{validate } (i, N, \mathcal{B}) (\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta) \theta$  returns `true`. Consider an execution  $\mathcal{B}^N : S \xrightarrow{*} S'$  of the unrolled initial loop, with  $N \geq \mu$ . Assume  $S \cong_{\theta} T$ . Then, there exists a concrete state  $T'$  such that

$$\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)} : T \xrightarrow{*} T' \quad (21)$$

$$S' \cong_{\theta} T' \quad (22)$$

Moreover, execution (21) decomposes as follows:

$$T \xrightarrow{\mathcal{P}} T_0 \xrightarrow[\kappa(N) \text{ times}]{\mathcal{S}} \cdots \xrightarrow[\kappa(N) \text{ times}]{\mathcal{S}} T_{\kappa(N)} \xrightarrow{\mathcal{E}} T'_0 \xrightarrow[\rho(N) \text{ times}]{\mathcal{B}} \cdots \xrightarrow[\rho(N) \text{ times}]{\mathcal{B}} T'_{\rho(N)} = T' \quad (23)$$

and the values of the loop index  $i$  in the various intermediate states satisfy

$$T_j(i) = S(i) + \mu + \delta j \pmod{2^{32}} \quad (24)$$

$$T'_j(i) = S(i) + \mu + \delta \times \kappa(N) + j \pmod{2^{32}} \quad (25)$$

**Proof:** As a corollary of Lemma 2, properties (17) and (18), we obtain

$$\alpha(\mathcal{B}^N) \approx_{\theta} \alpha(\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)})$$

$$\alpha(\mathcal{P}; \mathcal{S}^{\kappa(N)}; \mathcal{E}; \mathcal{B}^{\rho(N)}) \sqsubseteq \theta$$

since  $N = \mu + \delta \times \kappa(N) + \rho(N)$ . The existence of  $T'$  then follows from Theorem 1.

It is obvious that the concrete execution of the pipelined code can be decomposed as shown in (23). Check D in the validator

CFG instructions:

$$I_g ::= (I, \ell) \quad \text{non-branching instr.} \\ | (\text{if}(r < r'), \ell_t, \ell_f) \quad \text{conditional branch}$$

CFG:

$$g ::= \ell \mapsto I_g \quad \text{finite map}$$

Transition semantics:

$$\begin{array}{c} g(\ell) = (I, \ell') \quad I : (R, M) \rightarrow (R', M') \text{ (as in figure 4)} \\ \hline g : (\ell, R, M) \rightarrow (\ell', R', M') \\ \\ g(\ell) = (\text{if}(r < r'), \ell_t, \ell_f) \quad \ell' = \begin{cases} \ell_t & \text{if } R(r) < R(r') \\ \ell_f & \text{if } R(r) \geq R(r') \end{cases} \\ \hline g : (\ell, R, M) \rightarrow (\ell', R, M) \end{array}$$

**Figure 5.** A simple language of control-flow graphs

guarantees that  $S_{j+1}(i) = S_j(i) + 1 \pmod{2^{32}}$ , which implies  $S_j(i) = S(i) + j \pmod{2^{32}}$ . Combined with conclusion (19) of Lemma 2, this implies equation (24):

$$T_j(i) = S_{\mu+\delta j}(i) = S(i) + \mu + \delta j \pmod{2^{32}}$$

as well as

$$T'_0(i) = S_{\mu+\delta \times \kappa(N)}(i) = S(i) + \mu + \delta \times \kappa(N) \pmod{2^{32}}$$

Check D entails that  $T'_j(i) = T'_0(i) + j \pmod{2^{32}}$ . Equation (25) therefore follows.  $\square$

## 6.2 Soundness with respect to CFG loops

The second and last part of the soundness proof extends the semantic preservation results of theorem 3 from the unrolled loops in a basic-block representation to the actual loops in a CFG (control-flow graph) representation, namely the loops before and after pipelining depicted in figure 2. This extension is intuitively obvious but surprisingly tedious to formalize in full details: indeed, this is the only result in this paper that we have not yet mechanized in Coq.

Our final objective is to perform instruction scheduling on the RTL intermediate language from the CompCert compiler [12, section 6.1]. To facilitate exposition, we consider instead the simpler language of control-flow graphs defined in figure 5. The CFG  $g$  is represented as partial map from labels  $\ell$  to instructions  $I_g$ . An instruction can be either a non-branching instruction  $I$  or a conditional branch  $\text{if}(r < r')$ , complemented by one or two labels denoting the successors of the instruction. The operational semantics of this language is given by a transition relation  $g : (\ell, R, M) \rightarrow (\ell', R', M')$  between states comprising a program point  $\ell$ , a register state  $R$  and a memory state  $M$ .

The following trivial lemma connects the semantics of basic blocks (figure 4) with the semantics of CFGs (figure 5). We say that a CFG  $g$  contains the basic block  $B$  between points  $\ell$  and  $\ell'$ , and write  $g, B : \ell \rightsquigarrow \ell'$ , if there exists a path in  $g$  from  $\ell$  to  $\ell'$  that “spells out” the instructions  $I_1, \dots, I_n$  of  $B$ :

$$g(\ell) = (I_1, \ell_1) \wedge g(\ell_1) = (I_2, \ell_2) \wedge \dots \wedge g(\ell_n) = (I_n, \ell')$$

**LEMMA 4.** Assume  $g, B : \ell \rightsquigarrow \ell'$ . Then,  $g : (\ell, S) \xrightarrow{*} (\ell', S')$  if and only if  $B : S \xrightarrow{*} S'$ .

In the remainder of this section, we consider two control-flow graphs:  $g$  for the original loop before pipelining, depicted in part (a) of figure 2, and  $g'$  for the loop after pipelining, depicted in part (b) of figure 2. (Note that figure 2 is not a specific example: it describes

the general shape of CFGs accepted and produced by all software pipelining optimizations we consider in this paper.) We now use lemma 4 to relate a CFG execution of the original loop  $g$  with the basic-block execution of its unrolling.

**LEMMA 5.** Assume  $\alpha(\mathcal{B})(i) = \text{Op}(\text{addi}(1), i_0)$  and  $\alpha(\mathcal{B})(N) = N_0$ . Consider an execution  $g : (\text{in}, S) \xrightarrow{*} (\text{out}, S')$  from point  $\text{in}$  to point  $\text{out}$  in the original CFG, with  $S(i) = 0$ . Then,  $\mathcal{B}^N : S \xrightarrow{*} S'$  where  $N$  is the value of variable  $N$  in state  $S$ .

**Proof:** The CFG execution can be decomposed as follows:

$(\text{in}, S_0) \rightarrow (\text{x}, S_0) \xrightarrow{*} (\text{in}, S_1) \rightarrow \dots \xrightarrow{*} (\text{in}, S_n) \rightarrow (\text{out}, S_n)$  with  $S = S_0$  and  $S' = S_n$  and  $S_n(i) \geq S_n(N)$  and  $S_j(i) < S_j(N)$  for all  $j < n$ . By construction of  $\mathcal{B}$ , we have  $g, \mathcal{B} : \text{x} \rightsquigarrow \text{in}$ . Therefore, by Lemma 4,  $\mathcal{B} : S_j \xrightarrow{*} S_{j+1}$  for  $j = 0, \dots, n-1$ . It follows that  $\mathcal{B}^n : S \xrightarrow{*} S'$ . Moreover, at each loop iteration, variable  $N$  is unchanged and variable  $i$  is incremented by 1. It follows that the number  $n$  of iterations is equal to the value  $N$  of variable  $N$  in initial state  $S$ . This is the expected result.  $\square$

Symmetrically, we can reconstruct a CFG execution of the pipelined loop  $g'$  from the basic-block execution of its unrolling.

**LEMMA 6.** Let  $T_{\text{init}}$  be an initial state where variable  $i$  has value 0 and variable  $N$  has value  $N$ . Let  $T$  be  $T_{\text{init}}$  where variable  $M$  is set to  $\mu + \delta \times \kappa(N)$ . Assume that properties (23), (24) and (25) hold, as well as check H. We can, then, construct an execution  $g' : (\text{in}, T_{\text{init}}) \xrightarrow{*} (\text{out}, T')$  from point  $\text{in}$  to point  $\text{out}$  in the CFG  $g'$  after pipelining.

**Proof:** By construction of  $g'$ , we have  $g' : \mathcal{P} : \text{c} \rightsquigarrow \text{d}$  and  $g' : \mathcal{S} : \text{f} \rightsquigarrow \text{d}$  and  $g' : \mathcal{E} : \text{e} \rightsquigarrow \text{g}$  and  $g' : \mathcal{B} : \text{h} \rightsquigarrow \text{g}$ . The result is obvious if  $N < \mu$ . Otherwise, applying Lemma 4 to the basic-block executions given by property (23), we obtain the following CFG executions:

$$\begin{aligned} g' : (\text{in}, T_{\text{init}}) &\xrightarrow{*} (\text{c}, T) \\ g' : (\text{c}, T) &\xrightarrow{*} (\text{d}, T_0) \\ g' : (\text{f}, T_j) &\xrightarrow{*} (\text{d}, T_{j+1}) \text{ for } j = 0, \dots, \kappa(N)-1 \\ g' : (\text{e}, T_{\kappa(N)}) &\xrightarrow{*} (\text{g}, T'_0) \\ g' : (\text{h}, T'_j) &\xrightarrow{*} (\text{g}, T'_{j+1}) \text{ for } j = 0, \dots, \rho(N)-1 \end{aligned}$$

The variables  $N$  and  $M$  are invariant between points  $\text{c}$  and  $\text{out}$ :  $N$  because of check H,  $M$  because it is a fresh variable. Using properties (24) and (25), we see that the condition at point  $\text{d}$  is true in states  $T_0, \dots, T_{\kappa(N)-1}$  and false in state  $T_{\kappa(N)}$ . Likewise, the condition at point  $\text{g}$  is true in states  $T'_0, \dots, T'_{\rho(N)-1}$  and false in state  $T'_{\rho(N)}$ . The expected result follows.  $\square$

Piecing everything together, we obtain the desired semantic preservation result.

**THEOREM 7.** Assume that validate  $(i, N, \mathcal{B})$   $(\mathcal{P}, \mathcal{S}, \mathcal{E}, \mu, \delta)$  0 returns true. Let  $g$  and  $g'$  be the CFG before and after pipelining, respectively. Consider an execution  $g : (\text{in}, S) \xrightarrow{*} (\text{out}, S')$  with  $S(i) = 0$ . Then, for all states  $T$  such that  $S \cong_{\theta} T$ , there exists a state  $T'$  such that  $g' : (\text{in}, T) \xrightarrow{*} (\text{out}, T')$  and  $S' \cong_{\theta} T'$ .

**Proof:** Follows from Theorem 3 and Lemmas 5 and 6.  $\square$

## 7. Discussion of completeness

Soundness (rejecting all incorrect code transformations) is the fundamental property of a validator; but relative completeness (not rejecting valid code transformations) is important in practice, since

a validator that reports many false positives is useless. Semantic equivalence between two arbitrary pieces of code being undecidable, a general-purpose validation algorithm cannot be sound and complete at the same time. However, a specialized validator can still be sound and complete for a specific class of program transformations—in our case, software pipelining optimizations based on modulo scheduling and modulo variable expansion. In this section, we informally discuss the sources of potential incompleteness for our validator.

**Mismatch between infinitary and finitary symbolic evaluation** A first source of potential incompleteness is the reduction of the infinitary condition  $\forall N, \alpha(X_N) = \alpha(Y_N)$  to the two finitary checks A and B. More precisely, the soundness proof of section 6 hinges on the following implication being true:

$$(A) \wedge (B) \implies (17)$$

or in other words,

$$\begin{aligned} \alpha(\mathcal{B}^\mu) &\approx_\theta \alpha(\mathcal{P}; \mathcal{E}) \wedge \alpha(\mathcal{E}; \mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{S}; \mathcal{E}) \\ &\implies \forall n, \alpha(\mathcal{B}^{\mu+n\delta}) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n; \mathcal{E}) \end{aligned}$$

However, the reverse implication does not hold in general. Assume that (17) holds. Taking  $n = 0$ , we obtain equation (A). However, we cannot prove (B). Exploiting (17) for  $n$  and  $n + 1$  iterations, we obtain:

$$\begin{aligned} \alpha(\mathcal{B}^{\mu+n\delta}) &\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{E}) \\ \alpha(\mathcal{B}^{\mu+n\delta}); \alpha(\mathcal{B}^\delta) &\approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{S}; \mathcal{E}) \end{aligned}$$

Combining these two equivalences, it follows that

$$\alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{E}; \mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{P}; \mathcal{S}^n); \alpha(\mathcal{S}; \mathcal{E}) \quad (26)$$

However, we cannot just “simplify on the left” and conclude  $\alpha(\mathcal{E}; \mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{S}; \mathcal{E})$ . To see why such simplifications are incorrect, consider the two blocks

$$\begin{aligned} x := 1; \quad \text{and} \quad x := 1; \\ y := 1 \quad y := x \end{aligned}$$

We have  $\alpha(x := 1); \alpha(y := 1) \approx_\theta \alpha(x := 1); \alpha(y := x)$  with  $\theta = \{x, y\}$ . However, the equivalence  $\alpha(y := 1) \approx_\theta \alpha(y := x)$  does not hold.

Coming back to equation (26), the blocks  $\mathcal{E}; \mathcal{B}^\delta$  and  $\mathcal{S}; \mathcal{E}$  could make use of some property of the state set up by  $\mathcal{P}; \mathcal{S}^n$  (such as the fact that  $x = 1$  in the simplified example above). However, (26) holds for any number  $n$  of iteration. Therefore, this hypothetical property of the state that invalidates (B) must be a loop invariant. We are not aware of any software pipelining algorithm that takes advantage of loop invariants such as  $x = 1$  in the simplified example above.

In conclusion, the reverse implication  $(17) \implies (A) \wedge (B)$ , which is necessary for completeness, is not true in general, but we strongly believe it holds in practice as long as the software pipelining algorithm used is purely syntactic and does not exploit loop invariants.

**Mismatch between symbolic evaluation and concrete executions** A second, more serious source of incompleteness is the following: equivalence between the symbolic evaluations of two basic blocks is a sufficient, but not necessary condition for those two blocks to execute identically. Here are some counter-examples.

1. The two blocks could execute identically because of some property of the initial state, as in

$$y := 1 \quad \text{vs.} \quad y := x$$

assuming  $x = 1$  initially.

2. Symbolic evaluation fails to account for some algebraic properties of arithmetic operators:

$$\begin{aligned} i := i + 1; \quad \text{vs.} \quad i := i + 2 \\ i := i + 1 \end{aligned}$$

3. Symbolic evaluation fails to account for memory separation properties within the same memory block:

$$\begin{aligned} \text{store(int32, } p + 4, y); \quad \text{vs.} \quad x := \text{load(int32, } p); \\ x := \text{load(int32, } p) \quad \text{store(int32, } p + 4, y) \end{aligned}$$

4. Symbolic evaluation fails to account for memory separation properties within different memory blocks:

$$\begin{aligned} \text{store(int32, } a, y); \quad \text{vs.} \quad x := \text{load(int32, } b); \\ x := \text{load(int32, } b) \quad \text{store(int32, } a, y) \end{aligned}$$

assuming that  $a$  and  $b$  point to different arrays.

Mismatches caused by specific properties of the initial state, as in example 1 above, are not a concern for us: as previously argued, it is unlikely that the pipelining algorithm would take advantage of these properties. The other three examples are more problematic: software pipeliners do perform some algebraic simplifications (mostly, on additions occurring within address computations and loop index updates) and do take advantage of nonaliasing properties to hoist loads above stores.

Symbolic evaluation can be enhanced in two ways to address these issues. The first way, pioneered by Necula [18], consists in comparing symbolic terms and states modulo an *equational theory* capturing the algebraic identities and “good variable” properties of interest. For example, the following equational theory addresses the issues illustrated in examples 2 and 3:

$$\begin{aligned} \text{Op(addi}(n), \text{Op(addi}(m), t)) &\equiv \text{Op(addi}(n + m), t) \\ \text{Load}(\kappa', t_1, \text{Store}(\kappa, t_2, t_v, t_m)) &\equiv \text{Load}(\kappa', t_2, t_m) \\ &\text{if } |t_2 - t_1| \geq \text{sizeof}(\kappa) \end{aligned}$$

The separation condition between the symbolic addresses  $t_1$  and  $t_2$  can be statically approximated using a decision procedure such as Pugh’s Omega test [21]. In practice, a coarse approximation, restricted to the case where  $t_1$  and  $t_2$  differ by a compile-time constant, should work well for software pipelining.

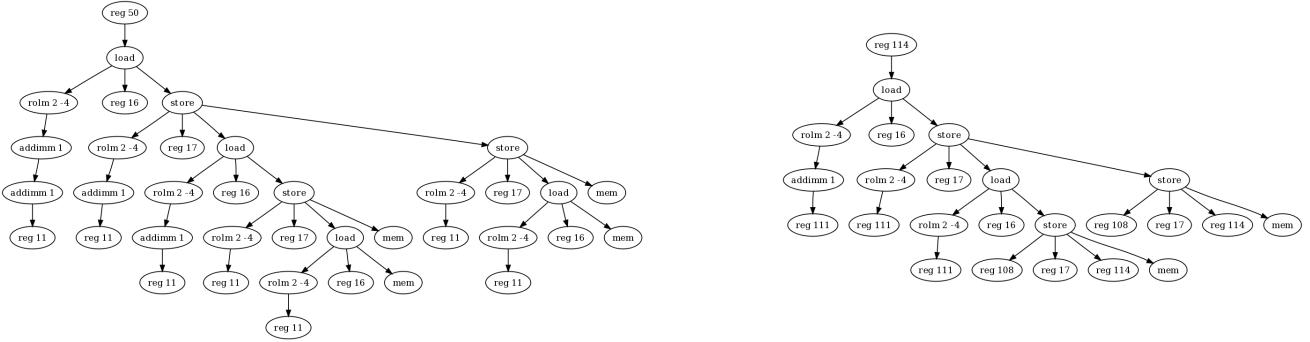
A different extension of symbolic evaluation is needed to account for nonaliasing properties between separate arrays or memory blocks, as in example 4. Assume that these nonaliasing properties are represented by region annotations on load and store instructions. We can, then, replace the single resource  $\text{Mem}$  that symbolically represent the memory state by a family of resources  $\text{Mem}^x$  indexed by region identifiers  $x$ . A load or store in region  $x$  is, then, symbolically evaluated in terms of the resource  $\text{Mem}^x$ . Continuing example 4 above and assuming that the store takes place in region  $a$  and the load in region  $b$ , both blocks symbolically evaluate to

$$\begin{aligned} \text{Mem}^a &\mapsto \text{Store(int32, } a_0, y_0, \text{Mem}_0^a) \\ x &\mapsto \text{Load(int32, } b_0, \text{Mem}_0^b) \end{aligned}$$

## 8. Implementation and preliminary experiments

The first author implemented the validation algorithm presented here as well as a reference software pipeliner. Both components are implemented in OCaml and were connected with the CompCert C compiler for testing.

The software pipeliner accounts for approximately 2000 lines of OCaml code. It uses a backtracking iterative modulo scheduler [2, chapter 20] to produce a steady state. Modulo variable expansion is then performed following Lam’s approach [10]. The modulo scheduler uses a heuristic function to decide the order in which



**Figure 6.** The symbolic values assigned to register 50 (left) and to its pipelined counterpart, register 114 (right). The pipelined value lacks one iteration due to a corner case error in the implementation of the modulo variable expansion. Symbolic evaluation was performed by omitting the prolog and epilog moves, thus showing the difference in register use.

to consider the instructions for scheduling. We tested our software pipeliner with two such heuristics. The first one randomly picks the nodes. We use it principally to stress the pipeliner and the validator. The second one picks the node using classical dependencies constraints. We also made a few schedules by hand.

We experimented our pipeliner on the CompCert benchmark suite, which contains, among others, some numerical kernels such as a fast Fourier transform. On these examples, the pipeliner performs significant code rearrangements, although the observed speedups are modest. There are several reasons for this: we do not exploit the results of an alias analysis; our heuristics are not state of the art; and we ran our programs on an out-of-order PowerPC G5 processor, while software pipelining is most effective on in-order processors.

The implementation of the validator follows very closely the algorithm presented in this paper. It accounts for approximately 300 lines of OCaml code. The validator is instrumented to produce a lot of debug information, including in the form of drawings of symbolic states. The validator found many bugs during the development of the software pipeliner, especially in the implementation of modulo variable expansion. The debugging information produced by the validator was an invaluable tool to understand and correct these mistakes. Figure 6 shows the diagrams produced by our validator for one of these bugs. We would have been unable to get the pipeliner right without the help of the validator.

On our experiments, the validator reported no false alarms. The running time of the validator is negligible compared with that of the software pipeliner itself (less than 5%). However, our software pipeliner is not state-of-the-art concerning the determination of the minimal initiation interval, and therefore is relatively costly in terms of compilation time.

Concerning formal verification, the underlying theory of symbolic evaluation (section 4) and the soundness proof with respect to unrolled loops (section 6.1) were mechanized using the Coq proof assistant. The Coq proof is not small (approximately 3500 lines) but is relatively straightforward: the mechanization raised no unexpected difficulties. The only part of the soundness proof that we have not mechanized yet is the equivalence between unrolled loops and their CFG representations (section 6.2): for such an intuitively obvious result, a mechanized proof is infuriatingly difficult. Simply proving that the situation depicted in figure 2(a) holds (we have a proper, counted loop whose body is the basic block  $\mathcal{B}$ ) takes a great deal of bureaucratic formalization. We suspect that the CFG representation is not appropriate for this kind of proofs; see section 10 for a discussion.

## 9. Related work

This work is not the first attempt to validate software pipelining *a posteriori*. Leviathan and Pnueli [14] present a validator for a software pipeliner for the IA-64 architecture. The pipeliner makes use of a rotating register file and predicate registers, but from a validation point of view it is almost the same problem as the one studied in this paper. Using symbolic evaluation, the validator generates a set of verification conditions that are discharged by a theorem prover. We chose to go further with symbolic evaluation: instead of using it to generate verification conditions, we use it to directly establish semantic preservation. We believe that the resulting validator is simpler and algorithmically more efficient. However, exploiting nonaliasing information during validation could possibly be easier in Leviathan and Pnueli’s approach than in our approach.

Another attempt at validating software pipelining is the one of Kundu *et al.* [9]. It proceeds by parametrized translation validation. The software pipelining algorithm they consider is based on code motion, a rarely-used approach very different from the modulo scheduling algorithms we consider. This change of algorithms makes the validation problem rather different. In the case of Kundu *et al.*, the software pipeliner proceeds by repeated rewriting of the original loop, with each rewriting step being validated. In contrast, modulo scheduling builds a pipelined loop in one shot (usually using a backtracking algorithm). It could possibly be viewed as a sequence of rewrites but this would be less efficient than our solution.

Another approach to establishing trust in software pipelining is run-time validation, as proposed by Goldberg *et al.* [5]. Their approach takes advantages of the IA-64 architecture to instrument the pipelined loop with run-time check that verify properties of the pipelined loop and recover from unexpected problems. This technique was used to verify that aliasing is not violated by instruction switching, but they did not verify full semantic preservation.

## 10. Conclusions

Software pipelining is one of the pinnacles of compiler optimizations; yet, its semantic correctness boils down to two simple semantic equivalence properties between basic blocks. Building on this novel insight, we presented a validation algorithm that is simple enough to be used as a debugging tool when implementing software pipelining, efficient enough to be integrated in production compilers and executed at every compilation run, and mathematically elegant enough to lend itself to formal verification.

Despite its respectable age, symbolic evaluation—the key technique behind our validator—finds here an unexpected application

to the verification of a loop transformation. Several known extensions of symbolic evaluation can be integrated to enhance the precision of the validator. One, discussed in section 7, is the addition of an equational theory and of region annotations to take nonaliasing information into account. Another extension would be to perform symbolic evaluation over extended basic blocks (acyclic regions of the CFG) instead of basic blocks, like Rival [24] and Tristan and Leroy [26] do. This extension could make it possible to validate pipelined loops that contain predicated instructions or even conditional branches.

From a formal verification standpoint, symbolic evaluation lends itself well to mechanization and gives a pleasant denotational flavor to proofs of semantic equivalence. This stands in sharp contrast with the last step of our proof (section 6.2), where the low-level operational nature of CFG semantics comes back to haunt us. We are led to suspect that control-flow graphs are a poor representation to reason over loop transformations, and to wish for an alternate representation that would abstract the syntactic details of loops just as symbolic evaluation abstracts the syntactic details of (extended) basic blocks. Two promising candidates are the PEG representation of Tate *et al.* [25] and the representation as sets of mutually recursive HOL functions of Myreen and Gordon [17]. It would be interesting to reformulate software pipelining validation in terms of these representations.

## Acknowledgments

This work was supported by Agence Nationale de la Recherche, project U3CAT, program ANR-09-ARPEGE. We thank François Pottier, Alexandre Pilkiwicz, and the anonymous reviewers for their careful reading and suggestions for improvements.

## References

- [1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, second edition, 2006.
- [2] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- [3] C. W. Barret, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *LNCS*, pages 291–295. Springer, 2005.
- [4] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *Proc. of the 16th international conference on Supercomputing*, pages 97–106. ACM, 2002.
- [5] B. Goldberg, E. Chapman, C. Huneycutt, and K. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *2002 International Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, pages 211–221. IEEE Computer Society Press, 2002.
- [6] Y. Huang, B. R. Childers, and M. L. Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *LNCS*, pages 281–300. Springer, 2006.
- [7] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267. ACM, 1993.
- [8] A. Kanade, A. Sanyal, and U. Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.
- [9] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 327–337. ACM Press, 2009.
- [10] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328. ACM, 1988.
- [11] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
- [12] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 2009. To appear.
- [13] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.
- [14] R. Leviathan and A. Pnueli. Validating software pipelining optimizations. In *Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002)*, pages 280–287. ACM Press, 2002.
- [15] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.
- [16] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.
- [17] M. O. Myreen and M. J. C. Gordon. Transforming programs into recursive functions. In *Brazilian Symposium on Formal Methods (SBMF 2008)*, volume 240 of *ENTCS*, pages 185–200. Elsevier, 2009.
- [18] G. C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.
- [19] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (CVT) – automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.
- [20] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.
- [21] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991.
- [22] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Processing*, 24(1):1–102, 1996.
- [23] B. R. Rau, M. S. Schlansker, and P. P. Timmalai. Code generation schema for modulo scheduled loops. Technical Report HPL-92-47, Hewlett-Packard, 1992.
- [24] X. Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
- [25] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *36th symposium Principles of Programming Languages*, pages 264–276. ACM Press, 2009.
- [26] J.-B. Tristan and X. Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.
- [27] J.-B. Tristan and X. Leroy. Verified validation of Lazy Code Motion. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 316–326. ACM Press, 2009.
- [28] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.
- [29] L. Zuck, A. Pnueli, and R. Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann institute of Science, 2001.

# Evaluating Value-Graph Translation Validation for LLVM

Jean-Baptiste Tristan    Paul Govereau    Greg Morrisett

Harvard University

{tristan,govereau,greg}@seas.harvard.edu

## Abstract

Translation validators are static analyzers that attempt to verify that program transformations preserve semantics. Normalizing translation validators do so by trying to match the value-graphs of an original function and its transformed counterpart. In this paper, we present the design of such a validator for LLVM's intra-procedural optimizations, a design that does not require any instrumentation of the optimizer, nor any rewriting of the source code to compile, and needs to run only once to validate a pipeline of optimizations. We present the results of our preliminary experiments on a set of benchmarks that include GCC, a perl interpreter, SQLite3, and other C programs.

**Categories and Subject Descriptors** F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Algebraic approaches to semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical Verification; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs - Program and recursion schemes

**General Terms** Algorithms, languages, reliability, theory, verification

**Keywords** Translation validation, symbolic evaluation, LLVM, optimization

## 1. Introduction

Translation validation is a static analysis that, given two programs, tries to verify that they have the same semantics [13]. It can be used to ensure that program transformations do not introduce semantic discrepancies, or to improve testing and debugging. Previous experiments have successfully applied a range of translation validation techniques to a variety of real-world compilers. Necula [11] experimented on GCC 2.7; Zuck *et al.* [4] experimented on the Tru64 compiler; Rival [14] experimented on unoptimized GCC 3.0; and Kanade *et al.* [8] experimented on GCC 4.

Given all these results, can we effectively validate a production optimizer? For a production optimizer, we chose LLVM. To be effective, we believe our validator must satisfy the following criteria. First, we do not want to instrument the optimizer. LLVM has a large collection of program transformations that are updated and improved at a frantic pace. To be effective, we want to treat the

optimizer as a “black box”. Second, we do not want to modify the source code of the input programs. This means that we handle the output of the optimizer when run on the input C programs “as is.” Third, we want to run only one pass of validation for the whole optimization pipeline. This is important for efficiency, and also because the boundaries between different optimizations are not always firm. Finally, it is crucial that the validator can scale to large functions. The experiments of Kanade *et al.* are the most impressive of all, with an exceptionally low rate of false alarms (note the validator requires heavy instrumentation of the compiler). However the authors admit that their approach does not scale beyond a few hundred instructions. In our experiments, we routinely have to deal with functions having several thousand instructions.

The most important issue is the instrumentation. To our knowledge, two previous works have proposed solutions that do not require instrumentation of the optimizer. Necula evaluated the effectiveness of a translation validator for GCC 2.7 with common-subexpression elimination, register allocation, scheduling, and loop inversion. The validator is simulation-based: it verifies that a simulation-relation holds between the two programs. Since the compiler is not instrumented, this simulation relation is inferred by collecting constraints. The experimental validation shows that this approach scales, as the validator handles the compilation of programs such as GCC or Linux. However, adding other optimizations such as loop-unswitch or loop-deletion is likely to break the collection of constraints.

On the other hand, Tate *et al.* [16] recently proposed a system for translation validation. They compute a value-graph for an input function and its optimized counterpart. They then augment the terms of the graphs by adding equivalent terms through a process known as *equality saturation*, resulting in a data structure similar to the E-graphs of congruence closure. If, after saturation, the two graphs are the same, they can safely conclude that the two programs they represent are equivalent. However, equality saturation was originally designed for other purposes, namely the search for better optimizations. For translation validation, it is unnecessary to saturate the value-graph, and generally more efficient and scalable to simply normalize the graph by picking an orientation to the equations that agrees with what a typical compiler will do (e.g.  $1 + 1$  is replaced by  $2$ ). Their preliminary experimental evaluation for the Soot optimizer on the JVM shows that the approach is effective and can lead to an acceptable rate of false alarms. However, it is unclear how well this approach would work for the more challenging optimizations available in LLVM, such as global-value-numbering with alias analysis or sparse-conditional constant propagation.

We believe that a *normalizing* value-graph translation validator would have both the simplicity of the *saturation* validator proposed by Tate *et al.*, and the scalability of Necula's constraint-based approach. In this paper we set out to evaluate how well such a design works. We therefore present the design and implementation of such a validator along with experimental results for a number of benchmarks including SQLite [2] and programs drawn from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'11, June 4–8, 2011, San Jose, California, USA.  
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...\$10.00

spec marks [5], including GCC and a perl interpreter. The optimizations we consider include global-value numbering with alias analysis, sparse-conditional constant propagation, aggressive dead-code elimination, loop invariant code motion, loop unswitching, loop deletion, instruction combining, and dead-store elimination. We have also experimented, on a smaller suite of hand-written programs and optimizations, that our tool could handle without further modification various flavors of scheduling (list, trace, etc.) and loop fusion and fission.

While Tate *et al.* tried a saturation approach on JVM code and found the approach to be effective, we consider here a normalization approach in the context of C code. C code is challenging to validate because of the lack of strong typing and the relatively low-level nature of the code when compared to the JVM. However, our results show that a normalizing value-graph translation validator can effectively validate the most challenging intra-procedural optimizations of the LLVM compiler. Another significant contribution of our work is that we provide a detailed analysis of the effectiveness of the validator with respect to the different optimizations. Finally, we discuss a number of more complicated approaches that we tried but ultimately found less successful than our relatively simple architecture.

The remainder of the paper is organized as follows. Section 2 presents the design of our tool. Section 3 details the construction of the value-graph using examples. Section 4 reviews the normalization rules that we use and addresses their effectiveness through examples. Section 5 presents the results of our experiments. We discuss related work in Section 6 and conclude in Section 7.

## 2. Normalizing translation validation

Our validation tool is called *LLVM-MD*. At a high-level, LLVM-MD is an optimizer: it takes as input an LLVM assembly file and outputs an LLVM assembly file. The difference between our tool and the usual LLVM optimizer is that our tool certifies that the semantics of the program is preserved. LLVM-MD has two components, the usual off-the-shelf LLVM optimizer, and a translation validator. The validator takes two inputs: the assembly code of a function before and after it has been transformed by the optimizer. The validator outputs a boolean: *true* if it can prove the assembly codes have the same semantics, and *false* otherwise. Assuming the correctness of our validator, a semantics-preserving LLVM optimizer can be constructed as follows (opt is the command-line LLVM optimizer, validate is our translation validator):

```
function llvm-md(var input) {
    output = opt -options input
    for each function f in input {
        extract f from input as fi and output as fo
        if (!validate fi fo) {
            replace fo by fi in output
        }
    }
    return output
}
```

For now, our validator works on each function independently, hence the limitation to intra-procedural optimizations. We believe that standard techniques can be used to validate programs in the presence of function inlining, but have not yet implemented these. Rather, we concentrate on the workhorse intra-procedural optimizations of LLVM.

At a high level, our tool works as follows. First, it “compiles” each of the two functions into a value-graph that represents the data dependencies of the functions. Such a value-graph can be thought of as a dataflow program, or as a generalization of the result of symbolic evaluation. Then, each graph is normalized by rewriting using

rules that mirror the rewritings that may be applied by the off-the-shelf optimizer. For instance, it will rewrite the 3-node sub-graph representing the expression  $2+3$  into a single node representing the value 5, as this corresponds to constant folding. Finally, we compare the resulting value-graphs. If they are *syntactically* equivalent, the validator returns *true*. To make comparison efficient, the value-graphs are hash-consed (from now on, we will say “reduced”). In addition, we construct a single graph for both functions to allow sharing between the two (conceptually distinct) graphs. Therefore, in the best case—when semantics has been preserved—the comparison of the two functions has complexity  $\mathcal{O}(1)$ . The best-case complexity is important because we expect most optimizations to be semantics-preserving.

The LLVM-MD validation process is depicted in figure 1. First, each function is converted into Monadic Gated SSA form [6, 10, 18]. The goal of this representation is to make the assembly instructions *referentially transparent*: all of the information required to compute the value of an instruction is contained within the instruction. More importantly, referential transparency allows us to substitute sub-graphs with equivalent sub-graphs without worrying about computational effects. Computing Monadic Gated SSA form is done in two steps:

1. Make side-effects explicit in the syntax by interpreting assembly instructions as monadic commands. For example, a *load* instruction will have a extra parameter representing the memory state.
2. Compute Gated SSA form, which extends  $\phi$ -nodes with conditions, insert  $\mu$ -nodes at loop headers, and  $\eta$ -nodes at loop exits[18].

Once we have the Monadic Gated SSA form, we compute a shared value-graph by replacing each variable with its definition, being careful to maximize sharing within the graph. Finally, we apply normalization rules and maximize sharing until the value of the two functions merge into a single node, or we cannot perform any more normalization.

It is important to note that the precision of the semantics-preservation property depends on the precision of the monadic form. If the monadic representation does not model arithmetic overflow or exceptions, then a successful validation does not guarantee anything about those effects. At present, we model memory state, including the local stack frame and the heap. We do not model runtime errors or non-termination, although our approach can be extended to include them. Hence, a successful run of our validator implies that if the input function terminates and does not produce a runtime error, then the output function has the same semantics. Our tool does not yet offer *formal* guarantees for non-terminating or semantically undefined programs.

## 3. Validation by Example

### 3.1 Basic Blocks

We begin by explaining how the validation process works for basic blocks. Considering basic blocks is interesting because it allows us to focus on the monadic representation and the construction of the value-graph, leaving for later the tricky problem of placing gates in  $\phi$ - and  $\eta$ -nodes.

Our validator uses LLVM assembly language as input. LLVM is a portable SSA-form assembly language with an infinite number of registers. Because we start with SSA-form, producing the value-graph consists of replacing variables by their definitions while maximizing sharing among graph nodes. For example, consider the

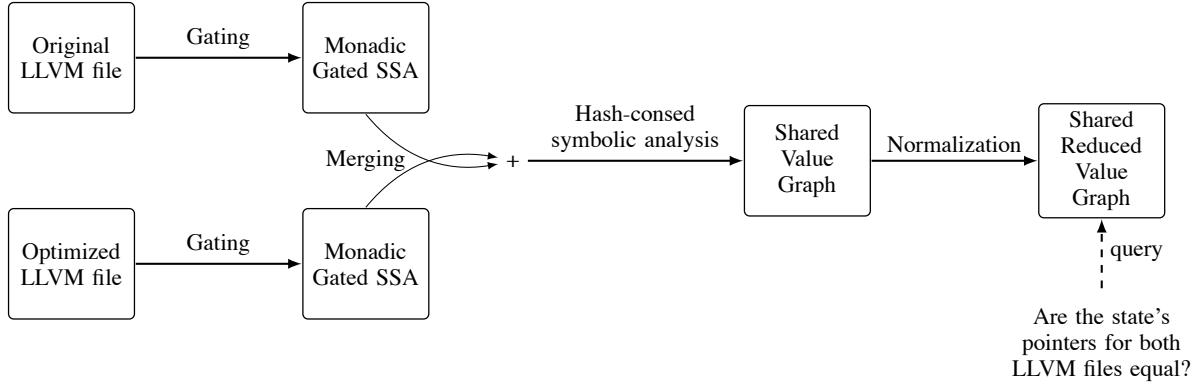


Figure 1: LLVM M.D. from a bird’s eye view

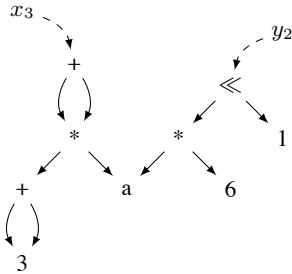
following basic block, B1:

$$\begin{aligned} \text{B1: } &x_1 = 3 + 3 \\ &x_2 = a * x_1 \\ &x_3 = x_2 + x_2 \end{aligned}$$

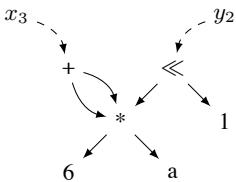
and its optimized counterpart, B2:

$$\begin{aligned} \text{B2: } &y_1 = a * 6 \\ &y_2 = y_1 \ll 1 \end{aligned}$$

Replacing variables  $x_1$ ,  $x_2$ , and  $y_1$  by their definition, we obtain the value-graph presented below. The dashed arrows are not part of the value graph, they are only meant to point out which parts of the graph correspond to which program variables. Note that both blocks have been represented within one value graph, and the node for the variable  $a$  has been shared.



Suppose that we want to show that the variables  $x_3$  and  $y_2$  will hold the same value. Once we have the shared value graph in hand, we simply need to check if  $x_3$  and  $y_2$  are represented by subgraphs rooted at the same graph node. In the value graph above,  $x_3$  and  $y_2$  are not represented by the same subgraph, so we cannot conclude they are equivalent. However, we can now apply normalization rules to the graph. First, we can apply a constant folding rule to reduce the subgraph  $3 + 3$  to a single node 6. The resulting graph is shown below (we have maximized sharing in the graph).



We have managed to make the value graph smaller, but we still cannot conclude that the two variables are equivalent. So, we continue to normalize the graph. A second rewrite rule allows us to replace  $x + x$  with  $x \ll 1$  for any  $x$ . In our setting, this rule is only appropriate if the optimizer would prefer the shift instruction to addition (which LLVM does). After replacing addition with left shift, and maximizing sharing,  $x_3$  and  $y_2$  point to the same node and we can conclude that the two blocks are equivalent.

**Side Effects.** The translation we have described up to this point would not be correct in the presence of side effects. Consider the following basic block.

```

p1 = alloc 1
p2 = alloc 1
store x, p1
store y, p2
z = load p1

```

If we naively apply our translation, then the graph corresponding to  $z$  would be:  $z \mapsto \text{load } (\text{alloc } 1)$ , which does not capture the complete computation for register  $z$ . In order to make sure that we do not lose track of side effects, we use abstract state variables to capture the dependencies between instructions. A simple translation gives the following sequence of instructions for this block:

```

p1, m1 = alloc 1, m0
p2, m2 = alloc 1, m1
m3 = store x, p1, m2
m4 = store y, p2, m3
z, m5 = load p1, m4

```

Here, the current memory state is represented by the  $m$  registers. Each instruction requires and produces a memory register in addition to its usual parameters. This extra register enforces a dependency between, for instance, the `load` instruction and the preceding `store` instructions. This translation is the same as we would get if we interpreted the assembly instructions as a sequence of monadic commands in a simple state monad[10]. Using these “monadic” instructions, we can apply our transformation and produce a value graph that captures all of the relevant information for each register.

The rewriting rules in our system are able to take into account aliasing information to relax the strict ordering of instructions imposed by the monadic transformation. In our setting (LLVM), we know that pointers returned by `alloc` never alias with each other.

Using this information, we are able to replace  $m_4$  with  $m_3$  in the `load` instruction for  $z$ . Then, because we have a `load` from a memory consisting of a `store` to the same pointer, we can simplify the `load` to  $x$ .

Using the state variables, we can validate that a function not only computes the same value as another function, but also affects the heap in the same way. The same technique can be applied to different kinds of side effects, such as arithmetic overflow, division by zero, and non-termination. Thus far we have only modeled memory side-effects in our implementation. Hence, we only prove semantics preservation for terminating programs that do not raise runtime errors. However, our structure allows us to easily extend our implementation to a more accurate model, though doing so may make it harder to validate optimizations.

### 3.2 Extended Basic Blocks

These ideas can be generalized to extended basic blocks as long as  $\phi$ -nodes are referentially transparent. We ensure this through the use of *gated*  $\phi$ -nodes. Consider the following program, which uses a normal  $\phi$ -node as you would find in an SSA-form assembly program.

```

entry : c = a < b
        cbr c, true, false

true : x1 = x0 + x0      (True branch)
       br join

false : x2 = x0 * x0     (False branch)
        br join

join : x3 = φ(x1, x2)   (Join point)

```

Rewriting these instructions as is would not be correct. For instance, replacing the condition  $a < b$  by  $a \geq b$  would result in the same value-graph, and we could not distinguish these two different programs. However, if the  $\phi$ -node is extended to include the conditions for taking each branch, then we can distinguish the two programs. In our example, the last instruction would become  $x_3 = \phi(b, x_1, x_2)$ , which means  $x_3$  is  $x_1$  if  $b$  is *true*, and  $x_2$  otherwise.

In order to handle real C programs, the actual syntax of  $\phi$ -nodes has to be a bit more complex. In general, a  $\phi$ -node is composed of a set of possible branches, one for each control-flow edge that enters the  $\phi$ -node. Each branch has a set of conditions, all of which must be true for the branch to be taken.<sup>1</sup>

$$\phi \left\{ \begin{array}{l} c_{11} \dots c_{1n} \rightarrow v_1 \\ \dots \\ c_{k1} \dots c_{km} \rightarrow v_k \end{array} \right\}$$

Given this more general syntax, the notation  $\phi(c, x, y)$  is simply shorthand for  $\phi(c \rightarrow x, !c \rightarrow y)$ .

Gated  $\phi$  nodes come along with a set of normalization rules that we present in the next section. When generating gated  $\phi$ -nodes, it is important that the conditions for each branch are mutually exclusive with the other branches. This way, we are free to apply normalization rules to  $\phi$ -nodes (such as reordering conditions and branches) without worrying about changing the semantics.

It is also worth noting that if the values coming from various paths are equivalent, then they will be shared in the value graph. This makes it possible to validate optimizations based on global-

<sup>1</sup>  $\phi$ -nodes with several branches and many conditions are very common in C programs. For example, an if-statement with a condition that uses short-cut boolean operators, can produce complex  $\phi$ -nodes.

value numbering that is aware of equivalences between definitions from distinct paths.

### 3.3 Loops

In order to generalize our technique to loops, we must come up with a way to place gates within looping control flow (including breaks, continues, and returns from within a loop). Also, we need a way to represent values constructed within loops in a referentially transparent way. Happily, Gated SSA form is ideally suited to our purpose.

Gated SSA uses two constructs to represent loops in a referentially transparent fashion. The first,  $\mu$ , is used to define variables that are modified within a loop. Each  $\mu$  is placed at a loop header and holds the initial value of the variable on entry to the loop and a value for successive iterations. The  $\mu$ -node is equivalent to a non-gated  $\phi$  node from classical SSA. The second,  $\eta$ , is used to refer to loop-defined variables from outside their defining loops. The  $\eta$ -node carries the variable being referred to along with the condition required to reach the  $\eta$  from the variable definition.

Figure 2a shows a simple while loop in SSA form. The Gated SSA form of the same loop is shown in figure 2b. The  $\phi$ -nodes in the loop header have been replaced with  $\mu$ -nodes, and the access to the  $x_p$  register from outside to loop is transformed into an  $\eta$ -node that carries the condition required to exit the loop and arrive at this definition.

With Gated SSA form, recursively defined variables must contain a  $\mu$ -node. The recursion can be thought of as a cycle in the value graph, and all cycles are dominated by a  $\mu$ -node. The value graph corresponding to our previous example is presented in figure 2c. Intuitively, the cycle in the value-graph can be thought of as generating a stream of values. The  $\mu$ -nodes start by selecting the initial value from the arrow marked with an “i”. Successive values are generated from the cyclic graph structure attached to the other arrow. This  $\mu$ -node “produces” the values  $c, c + 1, c + 2, \dots$ . The  $\eta$  receives a stream of values and a stream of conditions. When the stream of conditions goes from *true* to *false*, the  $\eta$  selects the corresponding value in the value stream.

Generally, we can think of  $\mu$  and  $\eta$  behaving according to the following formulas:

$$\begin{aligned} \mu(a, n) &= a : \mu(n[a/x], n) \\ \eta(0 : \bar{b}, x : \bar{v}) &= \eta(\bar{b}, \bar{v}) \\ \eta(1 : \bar{b}, x : \bar{v}) &= x \end{aligned}$$

Of course, for our purposes, we do not need to evaluate these formulas, we simply need an adequate, symbolic representation for the registers  $x$ ,  $x_p$  and  $b_p$ . A more formal semantics should probably borrow ideas from dataflow programming languages such as those studied by Tate *et. al.*[16].

## 4. Normalization

Once a graph is constructed for two functions, if the functions’ values are not already equivalent, we begin to normalize the graph. We normalize value graphs using a set of rewrite rules. We apply the rules to each graph node individually. When no more rules can be applied, we maximize sharing within the graph and then reapply our rules. When no more sharing or rules can be applied, the process terminates.

Our rewrite rules come in two basic types: general simplification rules and optimization-specific rules. The general simplification rules reduce the number of graph nodes by removing unnecessary structure. We say *general* because these rules only depend on the graph representation, replacing graph structures with smaller, simpler graph structures. The optimization-specific rules rewrite graphs in a way that mirrors the effects of specific optimizations.

```

 $x_0 = c$ 
loop :
 $x_p = \phi(x_0, x_k)$ 
 $b = x_p < n$ 
cbr b, loop1, exit

loop1 :
 $x_k = x_p + 1$ 
br loop

exit :
 $x = x_p$ 

```

(a) SSA while loop

```

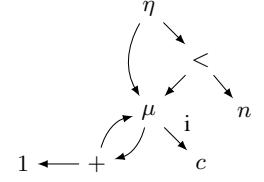
 $x_0 = c$ 
loop :
 $x_p = \mu(x_0, x_k)$ 
 $b = x_p < n$ 
cbr b, loop1, exit

loop1 :
 $x_k = x_p + 1$ 
br loop

exit :
 $x = \eta(b, x_p)$ 

```

(b) GSSA while loop



(c) Value graph representation

Figure 2: Representation of while loops

These rules do not always make the graph smaller or simpler, and one often needs to have specific optimizations in mind when adding them to the system.

**General Simplification Rules.** The notation  $a \downarrow b$  means that we match graphs with structure  $a$  and replace them with  $b$ . The first four general rules simplify boolean expressions:

$$a = a \downarrow \text{true} \quad (1)$$

$$a \neq a \downarrow \text{false} \quad (2)$$

$$a = \text{true} \downarrow a \quad (3)$$

$$a \neq \text{false} \downarrow a \quad (4)$$

These last two rules only apply if the comparison is performed at the boolean type. In fact, all LLVM operations, and hence our graph nodes, are typed. The types of operators are important and uninteresting: we do not discuss types in this paper.

There are two general rules for removing unnecessary  $\phi$ -nodes.

$$\phi \{ \dots, \overline{\text{true}}_i \rightarrow t, \dots \} \downarrow t \quad (5)$$

$$\phi \{ \overline{c_i} \rightarrow t \} \downarrow t \quad (6)$$

The first rule replaces a  $\phi$ -node with one of its branches if all of its conditions are satisfied for that branch. We write  $\overline{x}_i$  for a set of terms indexed by  $i$ . In the first rule, we have a set of true values. Note that the conditions for each branch are mutually exclusive with the other branches, so only one branch can have conditions which are all true. The second rule removes the  $\phi$ -node if all of the branches contain the same value. A special case of this rule is a  $\phi$ -node with only one branch indicating that there is only one possible path to the  $\phi$ -node, as happens with branch elimination.

The  $\phi$  rules are required to validate sparse conditional constant propagation (SCCP) and global value numbering (GVN). The following example can be optimized by both:

```

if (c) {a = 1; b = 1; d = a;}
else {a = 2; b = 2; d = 1;}
if (a == b) {x = d;} else {x = 0;}
return x;

```

Applying global-value numbering followed by sparse conditional constant propagation transforms this program to `return 1`. Indeed, in each of the branches of the first if-statement,  $a$  is equal to  $b$ . Since  $a == b$  is always true, the condition of the second if-statement is constant, and sparse conditional constant propagation can propagate the left definition of  $x$ . The above program and `return 1` have the same normalized value graph, computed as fol-

lows:

$$\begin{aligned}
&x \mapsto \phi(\phi(c, 1, 2) == \phi(c, 1, 2), \phi(c, 1, 1), 0) \\
&\downarrow \phi(\text{true}, \phi(c, 1, 1), 0) \quad \text{by (1)} \\
&\downarrow \phi(c, 1, 1) \quad \text{by (5)} \\
&\downarrow 1 \quad \text{by (6)}
\end{aligned}$$

There are also general rules for simplifying  $\eta$ - and  $\mu$ -nodes. The first rule allows us to remove loops that never execute.

$$\eta(\text{false}, \mu(x, y)) \downarrow x \quad (7)$$

This rule rewrites to the initial value of the  $\mu$ -node before the loop is entered, namely  $x$ . This rule is needed to validate loop-deletion, a form of dead code elimination. In addition, there are two rules for loop invariants. The first says that if we have a constant  $\mu$ -node, then the corresponding  $\eta$ -node can be removed:

$$\eta(c, \mu(x, x)) \downarrow x \quad (8)$$

$$\eta(c, y \mapsto \mu(x, y)) \downarrow x \quad (9)$$

In rule (8), the  $\mu$ -node has an initial value of  $x$ , which must be defined outside of the loop, and therefore cannot vary within the loop. Since,  $x$  does not vary within the loop the  $\mu$ -node does not vary, and the loop structure can be removed. Rule (9), expresses the same condition, but the second term in the  $\mu$ -node is again the same  $\mu$ -node (we use the notation  $y \mapsto \mu(x, y)$  to represent this self-reference).

These rules are necessary to validate loop invariant code motion. As an example, consider the following program:

```

x = a + 3; c = 3;
for (i = 0; i < n; i++) {x = a + c;}
return x;

```

In this program, variable  $x$  is defined within a loop, but it is invariant. Moreover, variable  $c$  is a constant. Applying global constant propagation, followed by loop-invariant code motion and loop deletion transforms the program to `return (a + 3)`. The value graph for  $x$  is computed as follows:

$$\begin{aligned}
&i_n \mapsto \mu(0, i_n + 1) \\
&x \mapsto \eta(i_n < n, \mu(a + 3, a + 3)) \\
&\downarrow a + c \quad \text{(by 8)}
\end{aligned}$$

Note that the global copy propagation is taken care of “automatically” by our representation, and we can apply our rule (8) immediately. The other nodes of the graph ( $i_n$ ) are eliminated since they are no longer needed.

**Optimization-specific Rules.** In addition to the general rules, we also have a number of rewrite rules that are derived from the semantics of LLVM. For example, we have a family of laws for simplifying constant expressions, such as:

```
add 3 2 ↓ 5
mul 3 2 ↓ 6
sub 3 2 ↓ 1
```

Currently we have rules for simplifying constant expressions over integers, but not floating point or vector types. There are also rules for rewriting instructions such as:

```
add a a ↓ shl a 1
mul a 4 ↓ shl a 2
```

These last two rules are included in our validator because we know the LLVM’s optimizer prefers the shift left instruction. While preferring shift left may be obvious, there are some less obvious rules such as:

```
add x (-k) ↓ sub x k
gt 10 a ↓ 1t a 10
1t a b ↓ 1e a (sub b 1)
```

While these transformations may not be optimizations, we believe LLVM makes them to give the instructions a more regular structure.

Finally, we also have rules that make use of aliasing information to simplify memory accesses. For example, we have the following two rules for simplifying loads from memory:

$$\text{load}(p, \text{store}(x, q, m)) \downarrow \text{load}(p, m) \quad (10)$$

$$\text{load}(p, \text{store}(x, p, m)) \downarrow x \quad (11)$$

when  $p$  and  $q$  do not alias. Our validator can use the result of a may alias analysis. For now, we only use simple non-aliasing rules: two pointers that originate from two distinct stack allocations may not alias; two pointers forged using `getelementptr` with different parameters may not alias, etc.

#### 4.1 Efficiency

At this point is natural to wonder why we did not simply define a normal form for expressions and rewrite our graphs to this normal form. This is a good option, and we have experimented with this strategy using external SMT provers to find equivalences between expressions. However, one of our goals is to build a practical tool which optimizes the best case performance of the validator (we expect most optimizations to be correct). Using our strategy of performing rewrites motivated by the optimizer, we are often able to validate functions with tens of thousands of instructions (resulting in value graphs with hundreds of thousands of nodes) with only a few dozen rewritings. That is, we strive to make the amount of work done by the validator proportional to the number of transformations performed by the optimizer.

To this end, the rewrite rules derived from LLVM semantics are designed to mirror the kinds of rewritings that are done by the LLVM optimization pipeline. In practice, it is *much* more efficient to transform the value graphs in the same way the optimizer transforms the assembly code: if we know that LLVM will prefer `shl a 1` to  $a + a$ , then we will rewrite  $a + a$  but not the other way around.

As another, more extreme, example, consider the following C code, and two possible optimizations: SCCP and GVN.

```
a = x < y;
b = x < y;
if (a) {
    if (a == b) {c = 1;} else {c = 2;}
```

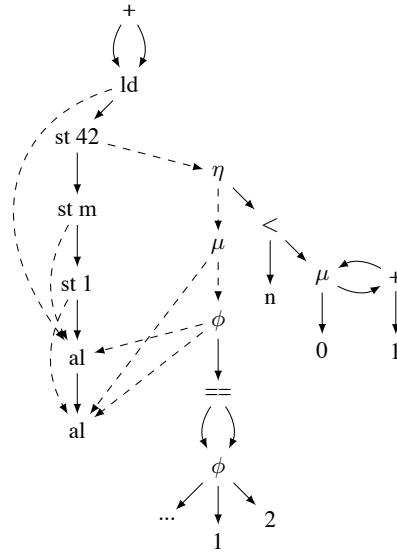


Figure 3: A shared value-graph

```
} else {c = 1;}
return c;
```

If the optimization pipeline is setup to apply SCCP first, then  $a$  may be replaced by `true`. In this case, GVN can not recognize that  $a$  and  $b$  are equal, and the inner condition will not be simplified. However, if GVN is applied first, then the inner condition can be simplified, and SCCP will propagate the value of  $c$ , leading to the program that simply returns 1. The problem of how to order optimizations is well-known, and an optimization pipeline may be reordered to achieve better results. If the optimization pipeline is configured to use GVN before SCCP, then, for efficiency, our simplification should be setup to simplify at join points before we substitute the value of  $a$ .

#### 4.2 Extended Example

We now present an example where all these laws interplay to produce the normalized value-graph. Consider the C code below:

```
int f(int n, int m) {
    int * t = NULL;
    int * t1 = alloca(sizeof(int));
    int * t2 = alloca(sizeof(int));
    int x, y, z = 0;
    *t1 = 1; *t2 = m;
    t = t1;
    for (int i = 0; i < n; ++i) {
        if (i % 3) {
            x = 1; z = x << y; y = x;
        } else {
            x = 2; y = 2;
        }
        if (x == y) t = t1;
        else t = t2;
    }
    *t = 42;
    return *t2 + *t2;
}
```

First, note that this function returns  $m + m$ . Indeed,  $x$  is always equal to  $y$  after the execution of the first conditional statement in

the for loop. Therefore, the second conditional statement always executes the left branch and assigns  $t_1$  to  $t$ . This is actually a loop invariant, and, since  $t_1$  is assigned to  $t$  before the loop,  $t_1$  is always equal to  $t$ .  $t_1$  and  $t_2$  are pointers to two distinct regions of memory and cannot alias. Writing through  $t_1$  does not affect what is pointed to by  $t_2$ , namely  $m$ . The function therefore returns  $m + m$ . Since the loop terminates, an optimizer may replace the body of this function with  $m \ll 1$ , using a blend of global-value numbering with alias analysis, sparse-conditional constant propagation and loop deletion.

Our value-graph construction and normalization produces the value-graph corresponding to  $m \ll 1$  for this example. The initial value-graph is presented in figure 3. Some details of the graph have been elided for clarity. We represent `load`, `store`, and `alloca` nodes with `ld`, `st`, and `al` respectively. To make the graph easier to read, we also used dashed lines for edges that go to pointer values. Below is a potential normalization scenario.

- The arguments of the `==` node are shared; it is rewritten to `true`.
- The gate of the  $\phi$  node is `true`; its predecessor,  $\mu$ , is modified to point to the target of the true branch of the  $\phi$  node rather than to  $\phi$  itself.
- The arguments of this  $\mu$  node are shared; its predecessor,  $\eta$ , is modified to point to the target of  $\mu$  instead of  $\mu$  itself.
- The condition of the  $\eta$  node is terminating, and its value acyclic; Its predecessor, `st42`, is modified to point to the target of  $\eta$  instead of  $\eta$  itself.
- It is now obvious that the load and its following store use distinct memory regions; the load can “jump over the store.”
- The load and its new store use the same memory region; the load is therefore replaced by the stored value,  $m$ .
- The arguments of the `+` node are shared; The `+` node is rewritten into a left shift.

## 5. Experimental evaluation

In our experimental evaluation, we aim to answer three different questions.

1. How effective is the tool for a decent pipeline of optimization?
2. How effective is the tool optimization-by-optimization?
3. What is the effect of normalization?

Our measure of effectiveness is simple: the fewer false alarms our tool produces, the more optimized the code will be. Put another way, assuming the optimizer is always correct, what is the cost of validation? In our experiments, we consider any alarm to be a false alarm.

### 5.1 The big picture

We have tested our prototype validator on the pure C programs of the SPECCPU 2006[5] benchmark (The `xalancbmk` benchmark is missing because the LLVM bitcode linker fails on this large program). In addition, we also tested our validator on the SQLite embedded database[2]. Each program was first compiled with clang version 2.8[1], and then processed with the `mem2reg` pass of the LLVM compiler to place  $\phi$ -nodes. These assembly files make up the unoptimized inputs to our validation tool. Each unoptimized input was optimized with LLVM, and the result compared to the unoptimized input by our validation tool.

**Test suite information.** Table 1 lists the benchmark programs with the size of the LLVM-assembly code file, number of lines of assembly, and the number of functions in the program. Our research

	size	LOC	functions
SQLite	5.6M	136K	1363
bzip2	904K	23K	104
gcc	63M	1.48M	5745
h264ref	7.3M	190K	610
hmmer	3.3M	90K	644
lmb	161K	5K	19
libquantum	337K	9K	115
mcf	149K	3K	24
milc	1.2M	32K	237
perlbench	15M	399K	1998
sjeng	1.5M	39K	166
sphinx	1.7M	44K	391

Table 1: Test suite information

prototype does not analyze functions with irreducible control flow graphs. This restriction comes from the front-end’s computation of Gated SSA form. It is well known how to compute Gated SSA form for any control-flow graph[18]. However, we have not extended our front-end to handle irreducible control flow graphs. We do not believe the few irreducible functions in our benchmarks will pose any new problems, since neither the Gated SSA form nor our graph representation would need to be modified.

**Pipeline information.** For our experiment, we used a pipeline consisting of:

- ADCE (advanced dead code elimination), followed by
- GVN (global value numbering),
- SCCP (sparse-condition constant propagation),
- LICM (loop invariant code motion),
- LD (loop deletion),
- LU (loop unswitching),
- DSE (dead store elimination).

The optimizations we chose are meant to cover, as much as possible, the intra-procedural optimizations available in LLVM. We do not include constant propagation and constant folding because both of these are subsumed by sparse-condition constant propagation (SCCP). Similarly, dead-code and dead-instruction elimination are subsumed by aggressive dead-code elimination (ADCE). We did not include reassociate and instcombine because we haven’t addressed those yet. One reason these haven’t been our priority is that they are conceptually simple to validate but require many rules.

For each benchmark, we ran all of the optimizations, and then attempted to validate the final result. Since we used the SQLite benchmark to engineer our rules, it is not surprising that, overall, that benchmark is very close to 90%. The rules chosen by studying SQLite are also very effective across the other benchmarks. We do not do quite as well for the perlbench and gcc benchmarks. Currently, our tool only uses the basic rewrite rules we have described here. Also we do not handle global constants or floating point expressions.

**Pipeline Results.** Overall, with the small number of rules we have described in this paper, we can validate 80% of the per-function optimizations. The results per-benchmark are shown in figure 4. For our measurements, we counted the number of functions for which we could validate all of the optimizations performed on the function: even though we may validate many optimizations, if even one optimization fails to validate we count the entire function as failed. We found that this conservative approach, while rejecting

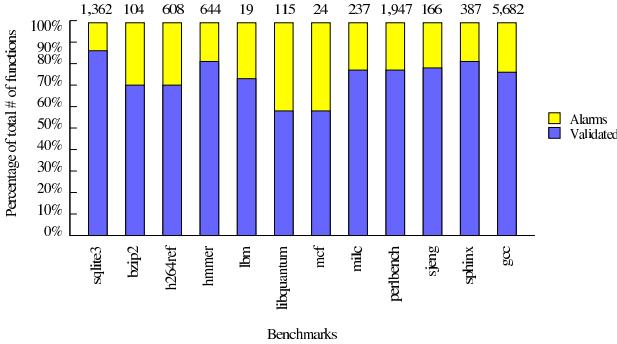


Figure 4: Validation results for optimization pipeline

more optimizations, leads to a simpler design for our validated optimizer which rejects or accepts whole functions at a time. The validation time for GCC is 19m19s, perl 2m56s, and SQLite 55s.

## 5.2 Testing Individual Optimizations

The charts in Figure 5 summarize the results of validating functions for different, single optimizations. The height of each bar indicates the total number of functions transformed for a given benchmark and optimization. The bar is split showing the number of validated (below) and unvalidated (above) functions. The total number of functions is lower in these charts because we do not count functions that are not transformed by the optimization.

It is clear from the charts that GVN with alias analysis is the most challenging optimization for our tool. It is also the most important as it performs many more transformations than the other optimizations. In the next section we will study the effectiveness of rewrite rules on our system.

## 5.3 Rewrite Rules

Figure 6 shows the effect of different rewrite rules for the GVN optimization with our benchmarks. The total height of each bar shows the percentage of functions we validated for each benchmark after the GVN optimization. The bars are divided to show how the results improve as we add rewrite rules to the system. We start with no rewrite rules, then we add rules and measure the improvement. The bars correspond to adding rules as follows:

1. no rules
2.  $\phi$  simplification
3. constant folding
4. load/store simplification
5.  $\eta$  simplification
6. commuting rules

We have already described the first five rules. The last set of rules tries to rearrange the graph nodes to enable the former rules. For example, we have a rule that tries to “push down”  $\eta$ -nodes to get them close to the matching  $\mu$ -nodes.

We can see from this chart that different benchmarks are effected differently by the different rules. For example, SQLite is not improved by adding rules for constant folding or  $\phi$  simplification. However, load/store simplification has an effect. This is probably because SQLite has been carefully tuned by hand and does not have many opportunities for constant folding or branch elimination. The lbm benchmark, on the other hand, benefits quite a lot from  $\phi$  simplification.

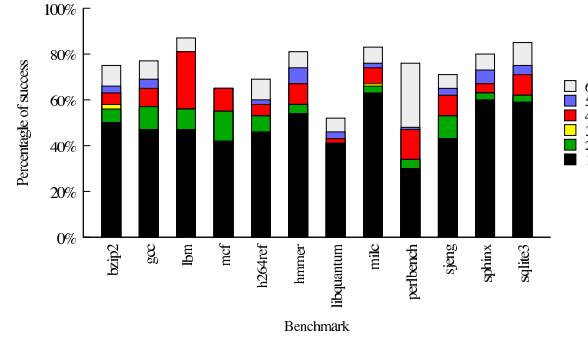


Figure 6: GVN

From the data we have, it seems that our technique is able to successfully validate approximately 50% of GVN optimizations with no rewrite rules at all. This makes intuitive sense because our symbolic evaluation hides many of the syntactic details of the programs, and the transformations performed by many optimizations are, in the end, minor syntactic changes. By adding rewrite rules we can dramatically improve our results.

Up to this point, we have avoided adding “special purpose” rules. For instance, we could improve our results by adding rules that allow us to reason about specific C library functions. For example, the rule:

$$\begin{array}{lcl} x = \text{atoi}(p); & \implies & y = \text{atoi}(q); \\ y = \text{atoi}(q); & \implies & x = \text{atoi}(p); \end{array}$$

can be added because `atoi` does not modify memory. Another example is:

$$memset(p, x, l_1); \quad \xrightarrow{l_2 < l_1} \quad y = x \\ y = \text{load}(\text{getelemptr}(p, l_2))$$

which enables more aggressive constant propagation. Both of these rules seem to be used by the LLVM optimizer, but we have not added them to our validator at this time. However, adding these sorts of rules is fairly easy, and in a realistic setting many such rules would likely be desired.

Figure 7 shows similar results for loop-invariant code motion (LICM). The baseline LICM, with no rewrite rules, is approximately 75-80%. If we add in all of our rewrite rules, we only improve very slightly. In theory, we should be able to completely validate LICM with no rules. However, again, LLVM uses specific knowledge of certain C library functions. For example, in the following loop:

```
for (int i = 0; i < strlen(p); i++) f(p[i]);
```

the call to `strlen` is known (by LLVM) to be constant. Therefore, LLVM will lift the call to `strlen` out of the loop:

```
int tmp = strlen(p);  
for (int i = 0; i < tmp; i++) f(p[i]);
```

Our tool does not have any rules for specific functions, and therefore we do not validate this transformation. The reason why we sometimes get a small improvement in LICM is because very occasionally a rewriting like the one above corresponds to one of our general rules.

Finally, figure 8 shows the effect of rewrite rules on sparse-conditional constant propagation. For this optimization, we used four configurations:

1. no rules

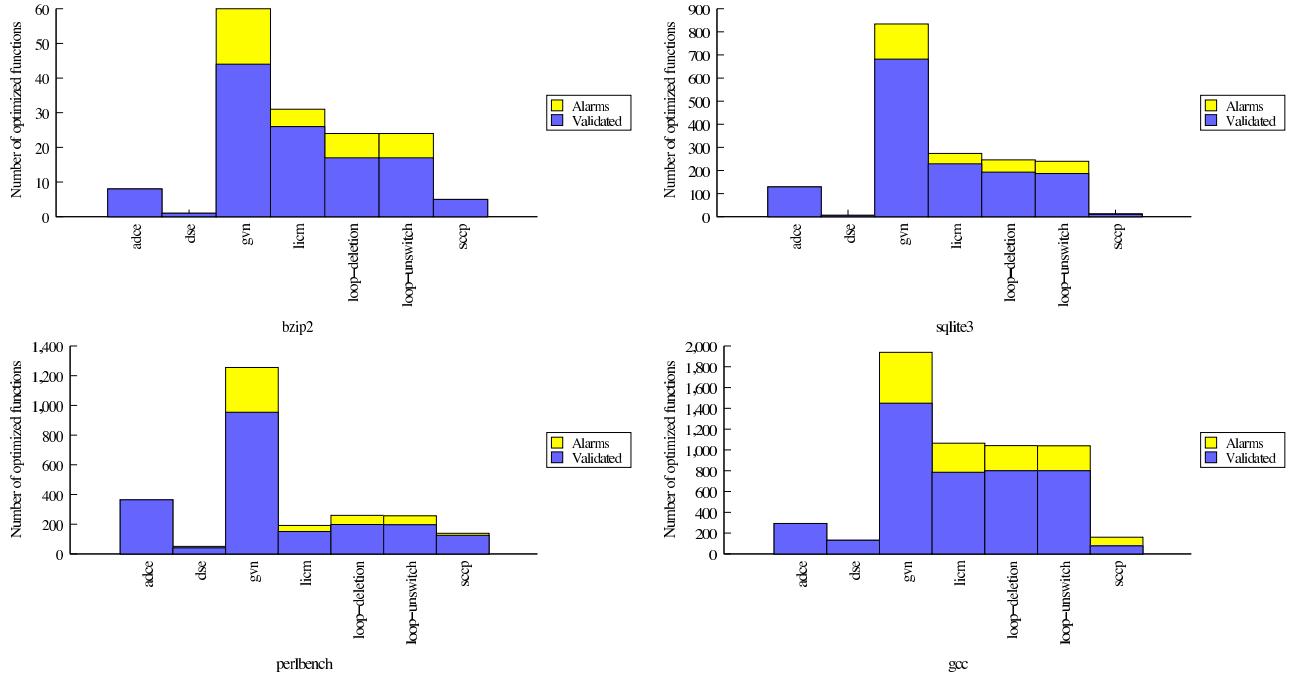


Figure 5: Validator results for individual optimizations

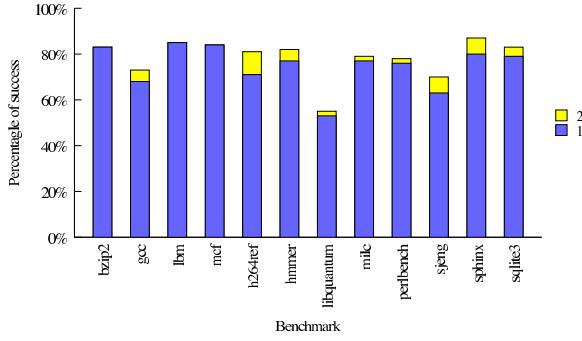


Figure 7: LICM

2. constant folding
3.  $\phi$  simplification
4. all rules

As expected, with no rules the results are very poor. However, if we add rules for constant folding, we see an immediate improvement, as expected. If we also add rules for reducing  $\phi$ -nodes, `bzip2` immediately goes to 100%, even though these rules ave no effect on `SQLite`. However, additional rules do improve `SQLite`, but not the other benchmarks.

#### 5.4 Discussion

While implementing our prototype, we were surprised to find that essentially all of the technical difficulties lie in the complex  $\phi$ -nodes. In an earlier version of this work we focused on structured code, and the (binary)  $\phi$ -nodes did not present any real difficulties. However, once we moved away from the structured-code restriction

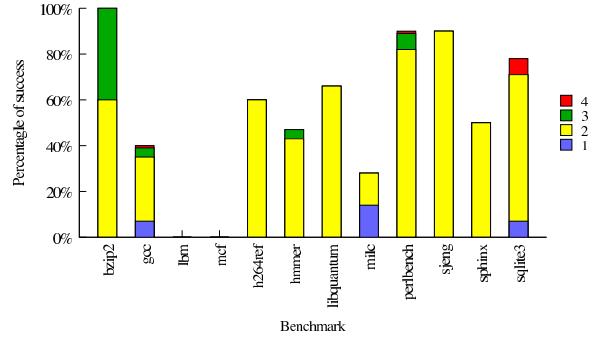


Figure 8: SCCP

we encountered more problems. First, although the algorithms are known, computing the gates for arbitrary control flow is a fairly involved task. Also, since the gates are dependent on the paths in the CFG, and general C code does not have a simple structured control flow, optimizations will often change the gating conditions even if the control flow is not changed.

Another important aspect of the implementation is the technique for maximizing sharing within the graph. The rewrite rules do a good job of exposing equivalent leaf nodes in the graphs. However, in order to achieve good results, it is important to find equivalent cycles in the graphs and merge them. Again, matching complex  $\phi$ -nodes seems to be the difficult part. To match cycles, we find pairs of  $\mu$ -nodes in the graph, and trace along their paths in parallel trying to build up a unifying substitution for the graph nodes involved. For  $\phi$ -nodes we sort the branches and conditions and perform a syntactic equality check. This technique is very simple, and

efficient because it only needs to query and update a small portion of the graph.

We also experimented with a Hopcroft partitioning algorithm[7]. Rather than a simple syntactic matching, our partitioning algorithm uses a prolog-style backtracking unification algorithm to find congruences between  $\phi$ -nodes. Surprisingly, the partitioning algorithm with backtracking does not perform better than the simple unification algorithm. Both algorithms give us roughly the same percentage of validation. Our implementation uses the simple algorithm by default, and when this fails it falls back to the slower, partitioning algorithm. Interestingly, this strategy performs slightly better than either technique alone, but not significantly better.

Matching expressions with complex  $\phi$ -nodes seems well within the reach of any SMT prover. Our preliminary experiments with Z3 suggest that it can easily handle the sort of equivalences we need to show. However, this seems like a very heavy-weight tool. One question in our minds is whether or not there is an effective technique somewhere in the middle: more sophisticated than syntactic matching, but short of a full SMT prover.

## 6. Related work

How do those results compare with the work of Necula and Tate *et al.*? The validator of Necula validates part of GCC 2.7 and the experiments show the results of compiling, and validating, GCC 2.91. It is important to note that the experiments were run on a Pentium Pro machine running at 400 MHz. Four optimizations were considered. Common-subexpression elimination, with a rate of false alarms of roughly 5% and roughly 7 minutes running time. Loop unrolling with a rate of false alarms of 6.3% and roughly 17 minutes running time. Register allocation with a rate of false alarms of 0.1% and around 10 minutes running time. Finally, instruction scheduling with a rate of false alarms of 0.01% and around 9 minutes running time. Unfortunately, the only optimization that we can compare to is CSE as we do not handle loop unrolling, and register allocation is part of the LLVM backend. In theory, we could handle scheduling (with as good results) but LLVM does not have this optimization. For CSE, our results are not as good as Necula's. However, we are dealing with a more complex optimization: global value numbering with partial redundancy elimination and alias information, libc knowledge, and some constant folding. The running times are also similar, but on different hardware. It is therefore unclear whether our validator does better or worse.

The validator of Tate *et al.* validates the Soot research compiler which compiles Java code to the JVM. On SpecJVM they report an impressive rate of alarms of only 2%. However, the version of the Soot optimizer they validate uses more basic optimizations than LLVM, and does not include, for instance, GVN. Given that our results are mostly directed by GVN with alias analysis, it makes comparisons difficult. Moreover, they do not explain whether the number they report takes into account all the functions or only the ones that were actually optimized.

The validator of Kanade *et al.*, even though heavily instrumented, is also interesting. They validate GCC 4.1.0 and report no false alarms for CSE, LICM, and copy propagation. To our knowledge, this experiment has the best results. However, it is unclear whether their approach can scale. The authors say that their approach is limited to functions with several hundred RTL instructions and a few hundred transformations. As explained in the introduction, functions with more than a thousand instructions are common in our setting.

There is a wide array of choices for value-graph representations of programs. Weise *et al.* [19] have a nice summary of the various value-graphs. Ours is close to the representation that results from the hash-consed symbolic analysis of a gated SSA graph [6, 18].

## 7. Conclusion

In conclusion, we believe that normalizing value-graph translation validation of industrial-strength compilers without instrumentation is feasible. The design relies on well established algorithms and is simple enough to implement. We have been able, in roughly 3 man-months, to build a tool that can validate the optimizations of a decent LLVM pipeline on challenging benchmarks, with a reasonable rate of false alarms. Better yet, we know that many of the false alarms that we witness now require the addition of normalization rules but no significant changes in the design. For instance, insider knowledge of libc functions, floating-points constant folding and folding of global variables are sources of false alarms that can be dealt with by adding normalization rules. There is also room for improvement of the runtime performance of the tool.

There are still a few difficult challenges ahead of us, the most important of which is inter-procedural optimizations. With LLVM, even -O1 makes use of such optimizations and, even though it is clear that simulation-based translation validation can handle inter-procedural optimizations [12], we do not yet know how to precisely generalize normalizing translation. We remark that, in the case of safety-critical code that respects standard code practices [3], as can be produced by tools like Simulink [15], the absence of recursive functions allows us to inline every function (which is reasonable with hash-consing). Preliminary experiments indicate that we are able to validate very effectively inter-procedural optimizations in such a restricted case. Advanced loop transformations are also important, and we believe that this problem may not be as hard as it may seem at first. Previous work [9, 17] has shown that it can be surprisingly easy to validate advanced loop optimizations such as software pipelining with modulo variable expansion if we reason at the value-graph level.

## Acknowledgments

We would like to thank Vikram Adve for his early interest and enthusiasm, and Sorin Lerner for discussing this project and exchanging ideas.

## References

- [1] LLVM 2.8. <http://llvm.org>.
- [2] SQLite 3. <http://www.sqlite.org>.
- [3] The Motor Industry Software Reliability Association. Guidelines for the use of the c language in critical systems. <http://www.misra.org.uk>, 2004.
- [4] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.
- [5] SPEC CPU2006. <http://www.spec.org/cpu2006/>.
- [6] Paul Havlak. Construction of thinned gated single-assignment form. In *Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.
- [7] John Hopcroft. An  $n \log n$  algorithm for minimizing states of a finite automaton. In *The Theory of Machines and Computations*, 1971.
- [8] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *4th Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.
- [9] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
- [10] E. Moggi. Computational lambda-calculus and monads. In *4th Logic in computer science*, pages 14–23. IEEE, 1989.

- [11] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.
- [12] Amir Pnueli and Anna Zaks. Validation of interprocedural optimization. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.
- [13] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.
- [14] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
- [15] Simulink. <http://mathworks.com>.
- [16] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *36th Principles of Programming Languages*, pages 264–276. ACM, 2009.
- [17] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *37th Principles of Programming Languages*, pages 83–92. ACM Press, 2010.
- [18] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Programming Language Design and Implementation*, pages 47–55, 1995.
- [19] Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *21st Principles of Programming Languages*, pages 297–310, 1994.

# RockSalt: Better, Faster, Stronger SFI for the x86

Greg Morrisett \*

greg@eecs.harvard.edu

Gang Tan

gtan@cse.lehigh.edu

Joseph Tassarotti

tassarotti@college.harvard.edu

Jean-Baptiste Tristan

tristan@seas.harvard.edu

Edward Gan

egan@college.harvard.edu

## Abstract

Software-based fault isolation (SFI), as used in Google’s Native Client (NaCl), relies upon a conceptually simple machine-code analysis to enforce a security policy. But for complicated architectures such as the x86, it is all too easy to get the details of the analysis wrong. We have built a new checker that is smaller, faster, and has a much reduced trusted computing base when compared to Google’s original analysis. The key to our approach is automatically generating the bulk of the analysis from a declarative description which we relate to a formal model of a subset of the x86 instruction set architecture. The x86 model, developed in Coq, is of independent interest and should be usable for a wide range of machine-level verification tasks.

**Categories and Subject Descriptors** D.2.4 [Software Engineering]: Software/Program Verification

**General Terms** security, verification

**Keywords** software fault isolation, domain-specific languages

## 1. Introduction

Native Client (NaCl) is a new service provided by Google’s Chrome browser that allows native executable code to be run directly in the context of the browser [37]. To prevent buggy or malicious code from corrupting the browser’s state, leaking information, or directly accessing system resources, the NaCl loader checks that the binary code respects a *sandbox* security policy. The sandbox policy is meant to ensure that, when loaded and executed, the untrusted code (a) will only read or write data in specified segments of memory, (b) will only execute code from a specified segment of memory, disjoint from the data segments, (c) will not execute a specific class of instructions (*e.g.*, system calls), and (d) will only communicate with the browser through a well-defined set of entry points.

Ensuring the correctness of the NaCl checker is crucial for preventing vulnerabilities, yet early versions had bugs that attackers could exploit, as demonstrated by a contest that Google ran [25]. A high-level goal of this work is to produce a high-assurance checker for the NaCl sandbox policy. Thus far, we have managed to construct a new NaCl checker for the 32-bit x86 (IA-32) processor (minus floating-point) which we call RockSalt. The RockSalt checker is smaller, marginally faster, and easier to modify than Google’s

\* This research was sponsored in part by NSF grants CCF-0915030, CCF-0915157, CNS-0910660, CCF-1149211, AFOSR MURI grant FA9550-09-1-0539, and a gift from Google.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI’12, June 11–16, 2012, Beijing, China.  
Copyright © 2012 ACM 978-1-4503-1205-9/12/06...\$10.00

original code. Furthermore, the core of RockSalt is automatically generated from a higher-level specification, and this generator has been proven correct with respect to a model of the x86 using the Coq proof assistant [9].

We are not the first to address assurance for SFI using formal methods. In particular, Zhao *et al.* [38] built a provably correct verifier for a sandbox policy similar to NaCl’s. Specifically, building upon a model of the ARM processor in HOL [13], they constructed a program logic and a provably correct verification condition generator, which when coupled with an abstract interpretation, generates proofs that assembly code respects the policy.

Our work has two key differences: First, there is no formal model for the subset of x86 that NaCl supports. Consequently, we have constructed a new model for the x86 in Coq. We believe that this model is an important contribution of our work, as it can be used to validate reasoning about the behavior of x86 machine code in other contexts (*e.g.*, for verified compilers).

Second, Zhao *et al.*’s approach takes about 2.5 hours to check a 300 instruction program, whereas RockSalt checks roughly 1M instructions per second. Instead of a general-purpose theorem prover, RockSalt only relies upon a set of tables that encode a deterministic finite-state automaton (DFA) and a few tens of lines of (trusted) C code. Consequently, the checker is extremely fast, has a much smaller run-time trusted computing base, and can be easily integrated into the NaCl runtime.

### 1.1 Overview

This paper has two major parts: the first part describes our model of the x86 in Coq and the second describes the RockSalt NaCl checker and its proof of correctness with respect to the model.

The x86 architecture is notoriously complicated, and our fragment includes a parser for over 130 different instructions with semantic definitions for over 70 instructions<sup>1</sup>. This includes support for operands that include byte and word immediates, registers, and complicated addressing modes (*e.g.*, scaled index plus offset). Furthermore, the x86 allows prefix bytes, such as operand size override, locking, and string repeat, that can be combined in many different ways to change the behavior of an instruction. Finally, the instruction set architecture is so complex, that it is unlikely that we can produce a faithful model from documentation, so we must be able to validate our model against implementations.

To address these issues, we have constructed a pair of domain-specific languages (DSLs), inspired by the work on SLED [30] and  $\lambda$ -RTL [29] (as well as more recent work [11, 19]), for specifying

<sup>1</sup> Some instructions have numerous encodings. For example, there are fourteen different opcode forms for the ADC instruction, but we count this as a single instruction.

```

Register      reg   ::= EAX | ECX | EDX | ...
Segment Reg. sreg ::= ES | CS | SS | ...
Scale        scale ::= 1 | 2 | 4 | 8
Operand
op  ::= int32 | reg
      int32 × option reg × option(scale × reg)
Instruction
i   ::= AAA | AAD | AAM | AAS | ADC(bool × op1 × op2)
      | ADD(bool × op1 × op2)
      | AND(bool × op1 × op2) | ...

```

**Figure 1.** Some Definitions for the x86 Abstract Syntax

the semantics of machine architectures, and have embedded those languages within Coq. Our DSLs are declarative and reasonably high-level, yet we can use them to generate OCaml code that can be run as a simulator. Furthermore, the tools are architecture independent and can thus be re-used to specify the semantics of other machine architectures. For example, one of the undergraduate co-authors constructed a model of the MIPS architecture using our DSLs in just a few days.

The Decoder DSL provides support for specifying the translation from bits to abstract syntax in a declarative fashion. We were able to take the tables from Intel’s manual [14] and use them to directly construct patterns for our decoder. Our embedding of the Decoder DSL includes both a denotational and operational semantics, and a proof of adequacy for the two interpretations. We use the denotational semantics for proving important properties about the decode stage of execution, and the operational semantics for execution validation.

The RTL (register transfer list) DSL is a small RISC-like core language parameterized by a notion of machine state. The RTL library includes an executable, small-step operational semantics. Each step in the semantics is specified as a (pure) function from machine states to machine states. We give meaning to x86 instructions by translating their abstract syntax to appropriate sequences of RTL instructions, similar to the way that a modern processor works. Reasoning about RTL is much easier than x86 code, as the number of instructions is smaller and orthogonal.

In what follows, we describe our DSLs and how they were used to construct the x86 model. We also describe our framework for validating the model against existing x86 implementations. We then describe the NaCl sandbox policy in detail, and the new RockSalt checker we have built to enforce it. Next, we describe the actual verification code and the proof of correctness. Finally, we close with a discussion of related work, future directions, and lessons learned.

## 2. A Coq Model of the x86

Our Coq model of the x86 instruction set architecture has three major stages: (1) a decoder that translates bytes into abstract syntax for instructions, (2) a compiler that translates abstract syntax into sequences of RTL instructions, (3) an interpreter for RTL instructions. The interface between the first two components is the definition of the abstract syntax, which is specified using a set of inductive datatype definitions that are informally sketched in Figure 1.

### 2.1 The Decoder Specification

The job of the x86 model’s decoder is to translate bytes into abstract syntax. We specify the translation using generic grammars constructed in a domain-specific language, which is embedded into Coq. The language lets users specify a pattern and associated se-

```

Definition CALL_p : grammar instr :=
  "1110" $$ "1000" $$ word @
  (fun w => CALL true false (Imm_op w) None)
|| "1111" $$ "1111" $$ ext_op_modrm2 "010" @
  (fun op => CALL true true op None)
|| "1001" $$ "1010" $$ halfword $ word @
  (fun p => CALL false false (Imm_op (snd p))
    (Some (fst p)))
|| "1111" $$ "1111" $$ ext_op_modrm2 "011" @
  (fun op => CALL false true op None).

```

**Figure 2.** Parsing Specification for the CALL instruction

mantic actions for transforming input strings to outputs such as abstract syntax. Our pattern language is limited to regular expressions, but the semantic actions are arbitrary Coq functions.

Figure 2 gives an example parsing specification we use for the CALL instruction. At a high-level, this grammar specifies four alternatives that can build a CALL instruction. Each case includes a pattern specifying literal sequences of bits (e.g., “1110”), followed by other components like `word` or `modrm2` that are themselves grammars that compute values of an appropriate type. The “@” separates the pattern from a Coq function that can be used to transform the values returned from the pattern match. For example, in the first case, we take the `word` value and use it to build the abstract syntax for a version of the CALL instruction with an immediate operand.

We chose to specify patterns at the bit-level, instead of the byte-level, because this avoids the need to introduce or reason about shifts and masks in the semantic actions. Furthermore, we were able to take the tables from the Intel IA-32 instruction manual and translate them directly into appropriate patterns.

Our decoding specifications take advantage of Coq’s notation mechanism, as well as some derived forms to make the grammar readable, but these are defined in terms of a small set of constructors given by the following type-indexed datatype:

```

Inductive grammar : Type → Type
| Char: char → grammar char
| Any: grammar char
| Eps: grammar unit
| Cat: ∀T1 T2, grammar T1 → grammar T2 → grammar (T1*T2)
| Void: ∀T, grammar T
| Alt: ∀T, grammar T → grammar T → grammar T
| Star: ∀T, grammar T → grammar (list T)
| Map: ∀T1 T2, (T1 → T2) → grammar T1 → grammar T2

```

A value of type `grammar T` represents a relation between lists of `char`<sup>2</sup> and semantic values of type `T`. Alternatively, we can think of the grammar as matching an input string and returning a set of associated semantic values. Formally, the denotation of a grammar is the least relation over strings and values satisfying the following equations:

$$\begin{aligned}
\llbracket \text{Char } c \rrbracket &= \{(c :: \text{nil}, c)\} \\
\llbracket \text{Any} \rrbracket &= \bigcup_c \{(c :: \text{nil}, c)\} \\
\llbracket \text{Eps} \rrbracket &= \{(\text{nil}, \text{tt})\} \\
\llbracket \text{Void} \rrbracket &= \emptyset \\
\llbracket \text{Alt } g_1 g_2 \rrbracket &= \llbracket g_1 \rrbracket \cup \llbracket g_2 \rrbracket \\
\llbracket \text{Cat } g_1 g_2 \rrbracket &= \{((s_1 s_2), (v_1, v_2)) \mid (s_i, v_i) \in \llbracket g_i \rrbracket\} \\
\llbracket \text{Map } f g \rrbracket &= \{(s, f(v)) \mid (s, v) \in \llbracket g \rrbracket\} \\
\llbracket \text{Star } g \rrbracket &= \llbracket \text{Map } (\lambda \_. \text{nil}) \text{Eps} \rrbracket \cup \\
&\quad \llbracket \text{Map } (:) (\text{Cat } g (\text{Star } g)) \rrbracket
\end{aligned}$$

Thus, `Char c` matches strings containing only the character `c`, and returns that character as the semantic value. Similarly, `Any` matches a string containing any single character `c`, and returns `c`. `Eps` matches only the empty string and returns `tt` (Coq’s unit).

<sup>2</sup>Grammars are parameterized by the type `char`.

The grammar `Void` matches no strings and thus returns no values. When  $g_1$  and  $g_2$  are grammars that each return values of type  $T$ , then the grammar `Alt g1 g2` matches a string  $s$  if either  $g_1$  matches  $s$  or  $g_2$  matches  $s$ . It returns the union of the values that  $g_1$  and  $g_2$  associate with the string. `Cat g1 g2` matches a string if it can be broken into two pieces that match the sub-grammars. It returns a pair of the values computed by the grammars. `Star` matches zero or more occurrences of a pattern, returning the results as a list.

The last constructor, `Map`, is our constructor for semantic actions. When  $g$  is a grammar that returns  $T_1$  values, and  $f$  is a function of type  $T_1 \rightarrow T_2$ , then  $\text{Map } f \ g$  is the grammar that matches the same set of strings as  $g$ , but transforms the outputs from  $T_1$  values to  $T_2$  values using  $f$ . If a grammar forgoes the use of `Map`, then the semantic values represent a parse tree for the input. `Map` makes it possible to incrementally transform the parse tree into alternate semantic values, such as abstract syntax.

As noted above, we use Coq's notation mechanism to make the grammars more readable. In particular, the following table gives some definitions for the notation used here:

$$\begin{array}{lcl} g_1 \sqcup\!\sqcup g_2 & := & \text{Alt } g_1 \ g_2 \\ & & | \\ & & g_1 \$ g_2 & := & \text{Cat } g_1 \ g_2 \\ g @ f & := & \text{Map } f \ g & | & g_1 \$\$ g_2 & := & (g_1 \$ g_2) @ \text{snd} \end{array}$$

We encode the denotational semantics in Coq using an inductively defined predicate, which makes it easy to symbolically reason about grammars. For example, one of our key theorems shows that our top-level grammar, which includes all possible prefixes and all possible integer instructions, is deterministic:

$$(s, v_1) \in [\![\text{x86grammar}]\!] \wedge (s, v_2) \in [\![\text{x86grammar}]\!] \implies v_1 = v_2$$

This helps provide some assurance that in transcribing the grammar from Intel's manual, we have not made a mistake. In fact, when we first tried to prove determinism, we failed because we had flipped a bit in an infrequently used encoding of the `MOV` instruction, causing it to overlap with another instruction.

## 2.2 The Decoder Implementation

While the denotational specification makes it easy to reason about grammars, it cannot be directly executed. Consequently, we define a parsing function which, when given a record representing a machine state, fetches bytes from the address specified by the program counter and attempts to match them against the grammar and build the appropriate instruction abstract syntax.

Our parsing function is defined by taking the *derivative* of the `x86grammar` with respect to the sequence of bits in each byte, and then checking to see if the resulting grammar accepts the empty string. The notion of derivatives is based on the ideas of Brzozowski [5] and more recently, Owens *et al.* [26] and Might *et al.* [24]. Reasoning about derivatives is much easier in Coq than attempting to transform grammars into the usual graph-based formalisms, as we need not worry about issues such as naming nodes, equivalence on graphs, or induction principles for graphs. Rather, all of our computation and reasoning can be done directly on the algebraic datatype of grammars.

Semantically, the derivative of a grammar  $g$  with respect to a character  $c$  is the relation:

$$\text{deriv}_c \ g = \{(s, v) \mid (c :: s, v) \in [\![g]\!]\}$$

That is,  $\text{deriv}_c \ g$  matches the tail of any string that starts with  $c$  and matches  $g$ .

Fortunately, calculating the derivative, including the appropriate transformation on the semantic actions, can be written as a straight-

forward function:

$$\begin{array}{ll} \text{deriv}_c \ \text{Any} & = \text{Map } (\lambda \_ c) \ \text{Eps} \\ \text{deriv}_c \ (\text{Char } c) & = \text{Map } (\lambda \_ c) \ \text{Eps} \\ \text{deriv}_c \ (\text{Alt } g_1 \ g_2) & = \text{Alt} (\text{deriv}_c \ g_1) (\text{deriv}_c \ g_2) \\ \text{deriv}_c \ (\text{Star } g) & = \text{Map } (\text{:}:)(\text{Cat}(\text{deriv}_c \ g)(\text{Star } g)) \\ \text{deriv}_c \ (\text{Cat } g_1 \ g_2) & = \text{Alt} (\text{Cat}(\text{deriv}_c \ g_1) \ g_2) \\ & \quad (\text{Cat}(\text{null } g_1) (\text{deriv}_c \ g_2)) \\ \text{deriv}_c \ (\text{Map } f \ g) & = \text{Map } f \ (\text{deriv}_c \ g) \\ \text{deriv}_c \ g & = \text{Void} \quad \text{otherwise} \end{array}$$

where  $\text{null } g$  is defined as:

$$\begin{array}{ll} \text{null } \text{Eps} & = \text{Eps} \\ \text{null } (\text{Alt } g_1 \ g_2) & = \text{Alt} (\text{null } g_1) (\text{null } g_2) \\ \text{null } (\text{Cat } g_1 \ g_2) & = \text{Cat} (\text{null } g_1) (\text{null } g_2) \\ \text{null } (\text{Star } g) & = \text{Map } (\lambda \_ \text{nil}) \ \text{Eps} \\ \text{null } (\text{Map } f \ g) & = \text{Map } f \ (\text{null } g) \\ \text{null } g & = \text{Void} \quad \text{otherwise} \end{array}$$

Effectively, `deriv` strips off a leading pattern that matches  $c$ , and adjusts the grammar with a `Map` so that it continues to calculate the same set of values. If the grammar cannot match a string that starts with  $c$ , then the resulting grammar is `Void`. The `null` function returns a grammar equivalent to `Eps` when its argument accepts the empty string, and `Void` otherwise. It is used to calculate the derivative of a `Cat`, which is simply the chain-rule for derivatives.

Once we calculate the iterated derivative of the grammar with respect to a string of bits, we can extract the set of related semantic values by running the `extract` function, which returns those semantic values associated with the empty string:

$$\begin{array}{ll} \text{extract } \text{Eps} & = \{\text{tt}\} \\ \text{extract } (\text{Star } g) & = \{\text{nil}\} \\ \text{extract } (\text{Alt } g_1 \ g_2) & = (\text{extract } g_1) \cup (\text{extract } g_2) \\ \text{extract } (\text{Cat } g_1 \ g_2) & = \{(v_1, v_2) \mid v_i \in \text{extract } g_i\} \\ \text{extract } (\text{Map } f \ g) & = \{f(v) \mid v \in \text{extract } g\} \\ \text{extract } g & = \emptyset \quad \text{otherwise} \end{array}$$

To be reasonably efficient, it is important that we optimize the grammar as we calculate derivatives. In particular, when we build a grammar, we always use a set of “smart” constructors, which are functions that perform local reductions, including:

$$\begin{array}{ll} \text{Cat } g \ \text{Eps} & \rightarrow g \\ \text{Cat } g \ \text{Void} & \rightarrow \text{Void} \\ \text{Alt } g \ \text{Void} & \rightarrow g \\ \text{Star } (\text{Star } g) & \rightarrow \text{Star } g \end{array} \quad \begin{array}{ll} \text{Cat } \text{Eps } g & \rightarrow g \\ \text{Cat } \text{Void } g & \rightarrow \text{Void} \\ \text{Alt } \text{Void } g & \rightarrow g \\ \text{Alt } g \ g & \rightarrow g \end{array}$$

Of course, the optimizations must add appropriate `Maps` to adjust the semantic actions. Proving the optimizations correct is an easy exercise using the denotational semantics. Unfortunately, the last of these optimizations ( $\text{Alt } g \ g \rightarrow g$ ) cannot be directly implemented as it demands a decidable notion of equality for grammars, yet our grammars include arbitrary Coq functions (and types). To work around these problems, we first translate grammars to an internal form, where all types and functions are replaced with a name that we can easily compare. An environment is used to track the mapping from names back to their definitions, and is consulted in the `extract` function to build appropriate semantic values.

In the end, we get a reasonably efficient parser that we can extract to executable OCaml code. Furthermore, we prove that the parser, when given a grammar  $g$  and string  $s$ , produces a (finite) set of values  $\{v_1, \dots, v_n\}$  such that  $(s, v_i) \in [\![g]\!]$ . Since we have proven that our instruction grammar is deterministic, we know that in fact, we will get out at most one instruction for each sequence of bytes that we feed to the parser.

Finally, we note that calculating derivatives in this fashion corresponds to a lazy, on-line construction of a deterministic finite-state transducer. Our efficient NaCl checker, described in Section 3

```

Machine locations
 $loc ::= PC \mid EAX \mid \dots \mid CF \mid \dots \mid SS \mid \dots$ 

Local variables
 $x, y, z \in \text{identifier}$ 

Arithmetic operators
 $op ::= add \mid xor \mid shl \mid \dots$ 

Comparison operators
 $cmp ::= lt \mid eq \mid gt$ 

RTL instructions
 $rt ::= x := y op z \mid x := y cmp z$ 
 $\mid x := imm \mid x := \text{load } loc$ 
 $\mid \text{store } loc \ x \mid x := \text{Mem}[y]$ 
 $\mid \text{Mem}[x] := y \mid x := \text{choose} \mid \dots$ 

```

---

**Figure 3.** The RTL Language

is built from a deterministic finite-state automaton (DFA) generated off-line, re-using the definitions for the grammars, derivatives, *etc.* in the parsing library.

### 2.3 Translation To RTL

After parsing bytes into abstract syntax, we translate the corresponding instruction into a sequence of RTL (register transfer list) operations. RTL is a small RISC-like language for computing with bit-vectors. The language abstracts over an architecture’s definition of machine state, which in the case of the x86 includes the various kinds of registers shown in Figure 1 as well as a memory, represented as a finite map from addresses to bytes. Internally, the language supports a countably infinite supply of local variables that can be used to hold intermediate bit-vector values.

The RTL instruction set is sketched in Figure 3 and includes standard arithmetic, logic, and comparison operations for bit vectors; operations to sign/zero-extend and truncate bit vectors; an operation to load an immediate value into a local variable; operations to load/store values in local variables from/to registers; operations to load and store bytes into memory; and a special operation for non-deterministically choosing a bit-vector value of a particular size. We use dependent types to embed the language into Coq and ensure that only bit-vectors of the appropriate size are used in instructions.

For each x86 constructor, we define a function that translates the abstract syntax into a sequence of RTL instructions. The translation is encapsulated in a monad that takes care of allocating fresh local variables, and that allows us to build higher-level operations out of sequences of RTL commands.

Figure 4 presents an excerpt of the translation of the ADD instruction. The ADD constructor is parameterized by a prefix record, a boolean mode, and two operands. The prefix record records modifiers including any segment, operand, or address override. The boolean mode is set when the default operand size is to be used (*i.e.*, 32-bits) and cleared when the operand size is set to a byte. The operands can be registers, immediate values, or effective addresses.

The first two local definitions specialize the load and store RTL to the given prefix and mode. The third definition selects the appropriate segment. Next, we load constant expressions 0 and 1 (of bit-size 1) into local variables `zero` and `up`. Then we fetch the bit-vector values from the operands and store them in local variables `p0` and `p1`. At this point, we actually add the two bit-vectors and place the result in local variable `p2`. Then we update the machine state at the location specified by the first operand. Afterwards, we set the various flag registers to hold the appropriate

```

Definition conv_ADD prefix mode op1 op2 :=
let load := load_op prefix mode in
let set := set_op prefix mode in
let seg := get_segment_op2 prefix DS op1 op2 in
zero ← load_Z size1 0;
up ← load_Z size1 1;
p0 ← load seg op1;
p1 ← load seg op2;
p2 ← arith add p0 p1;
set seg p2 op1;;
b0 ← test lt zero p0;
b1 ← test lt zero p1;
b2 ← test lt zero p2;
b3 ← arith xor b0 b1;
b3 ← arith xor up b3;
b4 ← arith xor b0 b2;
b4 ← arith and b3 b4;
set_flag OF b4;;
...

```

**Figure 4.** Translation Specification for the ADD instruction

1-bit value based on the outcome of the operation. Here, we have only shown the code needed to set the overflow (OF) flag.

Occasionally, the effect of an operation, particularly on flags, is under-specified or unclear. To over-approximate the set of possible behaviors, we use the `choose` operation, which non-deterministically selects a bit-vector value and stores this value in the appropriate location.

### 2.4 The RTL Interpreter

Once we have defined our decoder and translation to RTL, we need only give a semantics to the RTL instructions to complete the x86 model. One option would be to use a small-step operational semantics for modeling RTL execution, encoded as an inductive predicate. However, this would prevent us from extracting an executable interpreter which we need for validation.

Instead, we encode a step in the semantics as a function from RTL machine states to RTL machine states. RTL machine states record the values of the various x86 locations, the memory, and the values of the local variables. To support the non-determinism in the `choose` operation, the RTL machine state includes a stream of bits that serves as an *oracle*. Whenever we need to choose a new value, we simply pull bits from the oracle stream. Of course, when reasoning about the behavior of instructions, we must consider all possible oracle streams. This is a standard trick for turning a non-deterministic step relation into a function.

Most of the operations are simple bit-vector computations for which we use the CompCert integer bit-vector library [18]. Consequently, the definition of the interpreter is fairly straightforward and extracts to reasonable OCaml code that we can use for testing.

### 2.5 Model Validation

Any model of the x86 is complicated enough that it undoubtably has bugs. The only way we can gain any confidence is to test it against real x86 processors (and even they have bugs!). As described above, we have carefully engineered our model so that we can extract an executable OCaml simulator from our Coq definitions. We use this simulator to compare against an actual x86 processor.

One challenge in validating the simulator is extracting the machine state from the real processor. We use Intel’s Pin tool [20] to insert dynamic instrumentation into a binary. The instrumentation

dumps the values of the registers to a file after each instruction, and the values in memory after each system call. We then take the original binary and run it through our OCaml simulator, comparing the values of the registers after the RTL sequence for an instruction has been generated and interpreted. Unfortunately, this procedure sometimes generates false positives because of our occasional use of the oracle to handle undefined or under specified behaviors.

We use two different techniques to generate test cases to exercise the simulator. First, we generate small, random C programs using [Csmith](#) [36] and compile them using [GCC](#). This technique proved useful early in the development stage, especially to test standard instructions. In this way, we simulated and verified over 10 million instruction instances in about 60 hours on an 8 core intel Xeon running at 2.6Ghz.

However, this technique does not exercise instructions that are avoided by compilers, and even some common instructions have encodings that are rarely emitted by assemblers. For example, our previously discussed bug in the encoding of the `MOV` instruction was not uncovered by such testing because it falls in this category.

A more thorough technique is to fuzz test our simulator by generating random sequences of bytes, which has previously proved effective in debugging CPU emulators [21]. Using our generative grammar, we randomly produce byte sequences that correspond to instructions we have specified. This lets us exercise unusual forms of all the instructions we define. For instance, an instruction like `add` with `carry` comes in fourteen different flavors, depending on the width and types of the operands, whether immediates are sign-extended, etc. Fuzzing such an instruction guarantees with some probability that all of these forms will be exercised.

### 3. The RockSalt NaCl Checker

Recall that our high level goal is to produce a checker for Native Client, which when given an x86 binary image, returns true only when the image respects the sandbox policy: when executed, the code will only read/write data from specified contiguous segments in memory, will not directly execute a particular set of instructions (*e.g.*, system calls), and will only transfer control within its own image or to a specified set of entry points in the NaCl run-time.

The 32-bit x86 version of NaCl takes advantage of the segment registers to enforce most aspects of this policy. In particular, by setting the CS (code), DS (data), SS (stack), and GS (thread-local) segment registers appropriately, the machine itself will ensure that data reads and writes are contained in the data segments, and that jumps are contained within the code segment. However, we must make sure that the untrusted instructions do not change the values of the segment registers, nor override the segments inappropriately.

At first glance, it appears sufficient to simply parse the binary into a sequence of instructions, and check that each instruction in the sequence preserves the values of the segment registers and does not override the segment registers with a prefix. Unfortunately, this simple strategy does not suffice. The problem is that, since the x86 has variable length instructions, we must not only consider the parse starting at the first byte, but all possible parses of the image. While most programs will respect the initial parse, a malicious or buggy program may not. For example, in a program that has a buffer overrun, a return address may be overwritten by a value that points into the middle of an instruction from the original parse.

To avoid this problem, NaCl provides a modified compiler that rewrites code to respect a stronger *alignment policy*, following the ideas of McCamant and Morrisett [22]. The alignment policy requires that all computed jumps (*i.e.*, jumps through a register) are aligned on a 32-byte boundary. This is ensured by inserting code to mask the target address with an appropriate constant, and by inserting no-ops so that potential jump targets are suitably aligned. In more detail, the aligned, sandbox policy requires that:

1. Starting with the first byte, the image parses into a legal sequence of instructions that preserve the segment registers;
2. Every 32<sup>nd</sup> byte is the beginning of an instruction in our parse;
3. Every indirect jump through a register  $r$  is immediately preceded by an instruction that masks  $r$  so that it is 32-byte aligned;
4. The masking operation and jump are both contained within a 32-byte-aligned block of instructions;
5. Each direct jump targets the beginning of an instruction and that instruction is not an indirect jump.

Requirements 4 and 5 are needed to ensure that the code cannot jump over the masking operation that protects an indirect jump.

#### 3.1 Constructing a NaCl Checker

Google’s NaCl checker is a hand-written C program that is intended to enforce the aligned, sandbox policy. Their checker partially decodes the binary, looking at fields such as the op-codes and mod/rm bits to determine whether the instruction is legal, and how long it is. Two auxiliary data structures are used: One is a bit-map that records which addresses are the starts of instructions. Each time an instruction is parsed, the corresponding bit for the address of the first byte is set. The other is an array of addresses for forward, direct jumps. After checking that the instructions are legal, the bit-map is checked to ensure that every 32nd byte is the start of an instruction. Then, the array of direct jump targets is checked to make sure they are valid according to the policy above.

There are two disadvantages with Google’s checker: it is difficult to reason about because it is somewhat large (about 600 statements of code)<sup>3</sup> and the process of partial decoding is intertwined with policy enforcement. In particular, it is difficult to tell what instructions are supported and with what prefixes, and even more difficult to gain assurance that the resulting code enforces the appropriate sandbox policy. Furthermore, it is difficult to modify the code to *e.g.*, add new kinds of safe instructions or combinations of prefixes.

In contrast, the RockSalt checker we constructed and verified is relatively small, consisting of only about 80 lines of Coq code. This is because the checker uses table-driven DFA matching to handle the aspects of decoding, following an idea first proposed by Seaborn [33]. The basic idea is to break all instructions into four categories: (1) those that perform no control-flow, and are easily seen as okay; (2) those that perform a direct jump—we must check that the target is a valid instruction; (3) those that perform an indirect jump—we must check that the destination is appropriately masked; and (4) those instructions that should be rejected. Each of these classes, except the third one, can be described using a simple regular expression. The third class can be captured by a regular expression if we make the restriction that the masking operation must occur directly before the jump, which in practice is what the NaCl compiler does.

It is possible to extract OCaml code from our Coq definitions and use that as the core of the checker, but we elected to manually translate the code into C so that it would more easily integrate into the NaCl run-time. This avoids adding the OCaml compiler and run-time system to the trusted computing base, at the risk that our translation to C may have introduced an error. However, at under 100 lines of C code, we felt that this was a reasonable risk, since the vast majority of the information is contained in the DFA tables which are automatically generated and proven correct. Of course, one could try to use a verification tool, such as Frama-C/WP [10]

---

<sup>3</sup> To be fair, this includes CPU identification and support for floating-point and other instructions that we do not yet handle.

```

1. Bool verifier(DFA *NoControlFlow,
2.                 DFA *DirectJump, DFA *MaskedJump,
3.                 uint8_t *code, uint size)
4. {
5.     uint pos = 0, i, saved_pos;
6.     Bool b = TRUE;
7.     valid = (uint8_t *)calloc(size,sizeof(uint8_t));
8.     target = (uint8_t *)calloc(size,sizeof(uint8_t));
9.
10.    while (pos < size) {
11.        valid[pos] = TRUE;
12.        saved_pos = pos;
13.        if (match(MaskedJump,code,&pos,size)) continue;
14.        if (match(NoControlFlow,code,&pos,size)) continue;
15.        if (match(DirectJump,code,&pos,size) &&
16.            extract(code,saved_pos,pos,target)) continue;
17.        free(target); free(valid);
18.        return FALSE;
19.    }
20.
21.    for (i = 0; i < size; ++i)
22.        b = b && (!target[i] || valid[i]) &&
23.              (i & 0x1F || valid[i]);
24.
25.    free(target); free(valid);
26.    return b;
27. }
```

**Figure 5.** Main Routine of our NaCl Checker

or VCC [8], to prove the correctness of this version, in which case the functional code in Coq could serve as a specification.

Figure 5 shows the C code for the high-level verifier routine. This function relies upon two sub-routines, `match` and `extract` that we will explain later, but intuitively handle the aspects of decoding. Like Google’s checker, the routine uses two auxiliary arrays: the `valid` array records those addresses in the code that are valid jump destinations, whereas the `target` array records those addresses that are jumped to by some direct control-flow operation. We used byte arrays instead of bit arrays to avoid having to reason about shifts and masks to read/write bits.

The main loop (line 10) iterates through the bytes in the `code` starting at position 0. This position is marked as `valid` and then we attempt to `match` the bytes at the current position against three patterns. The first pattern, `MaskedJump`, matches only when the bytes specify a mask of register  $r$  followed immediately by an indirect jump or call through  $r$ . Note that a successful match increments the position by `size` which records the length of the instruction(s), whereas a failure to match leaves the position unmodified. The second pattern, `NoControlFlow`, matches only when the bytes specify a legal NaCl instruction that does not affect control flow (*e.g.*, an arithmetic instruction). The third pattern, `DirectJump` matches only when the bytes specify a direct JMP, Jcc or CALL instruction. The routine `extract` then extracts the destination address of the jump, and marks that address in the `target` array. If none of these cases match, then the checker returns `FALSE` indicating that an illegal sequence of bytes was found in the code.

After the main loop terminates, we must check that (a) if an address is the target of a direct jump, then that address is the beginning of an instruction in our parse (line 22), and (b) if an address is aligned on a 32-byte boundary, then that address is the beginning of an instruction in our parse (line 23).

The process of matching a sequence of bytes against a pattern is handled by the routine `match` which is shown in Figure 6. The function simply executes the transitions of a DFA using the bytes at the current position in the code. The DFA has four fields: a starting state, a boolean array of accepting states, a boolean array

```

1. Bool match(DFA *A, uint8_t *code,
2.             uint *pos, uint size)
3. {
4.     uint8_t state = A->start;
5.     uint off = 0;
6.
7.     while (*pos + off < size) {
8.         state = A->table[state][code[*pos + off]];
9.         off++;
10.        if (A->rejects[state]) break;
11.        if (A->accepts[state]) {
12.            *pos += off;
13.            return TRUE;
14.        }
15.    }
16.    return FALSE;
17. }
```

**Figure 6.** The DFA match routine

of rejecting states, and a transition table that maps a state and byte to a new state.

### 3.2 DFA Generation

What we have yet to show are the definitions of the DFAs for the `MaskedJump`, `NoControlFlow`, and `DirectJump` patterns, and the correctness of our checker hinges crucially upon these definitions. These are generated from within Coq using higher-level specifications. In particular, for each of the patterns, we specify a grammar re-using the parsing DSL described in Section 2.1, and then compile that grammar to appropriate DFA tables. For example, the grammar for a `MaskedJump` is given below:

```

Definition nacl_MASK_p (r: register) :=
  "1000" $$ "0011" $$ "11" $$ "100"
  $$ bitslist (register_to_bools r)
  $ bitslist (int_to_bools safeMask).

Definition nacl_JMP_p (r: register) :=
  "1111" $$ "1111" $$ "11" $$ "100"
  $$ bitslist (register_to_bools r).

Definition nacl_CALL_p (r: register) :=
  "1111" $$ "1111" $$ "11" $$ "010"
  $$ bitslist (register_to_bools r).

Definition nacljmp_p (r: register) :=
  nacl_MASK_p r $ (nacl_JMP_p r || nacl_CALL_p r).

Definition nacljmp_mask :=
  nacljmp_p EAX || nacljmp_p ECX || nacljmp_p EDX ||
  nacljmp_p EBX || nacljmp_p EBP || nacljmp_p ESI ||
  nacljmp_p EDI.
```

The `nacl_MASK_p` function takes a register name and generates a pattern for an “*AND r, safeMask*” instruction. The `nacl_JMP_p` and `nacl_CALL_p` functions take a register and generate patterns for a jump or call instruction (respectively) through that register. Thus, `nacljmp_mask` and the top-level grammar match any combination of a mask and jump through the same register (excluding ESP).

We compile grammars to DFAs from within Coq as follows: First, we strip off the semantic actions from the grammars so that we are left with a regular expression  $r_0$ . This regular expression corresponds to the starting state of the DFA. We use the `null` routine to check if this is an accepting state and a similar routine to check for rejection, and record this in a table. We then calculate the derivative of  $r_0$  with respect to all 256 possible input bytes. This yields a set of regular expressions  $\{r_1, r_2, \dots, r_n\}$ . Each  $r_i$  corresponds to a state in the DFA that is reachable from  $r_0$ . We assign each regular expression a state, and record whether that state

is an accepting or rejecting state. We continue calculating derivatives of each of the  $r_i$  with respect to all possible inputs until we no longer create a new regular expression. The fact that there are a finite number of unique derivatives (up to the reductions performed by our smart constructors) was proven by Brzozowski [5] so we are ensured that the procedure terminates.

In practice, calculating a DFA in this fashion is almost as good as the usual construction [26], but avoids the need to formalize and reason about graphs. The degree to which we simplify regular expressions as we calculate derivatives determines how few states are left in the resulting DFA. In our case, the number of states is small enough (61 for the largest DFA) that we do not need to worry about further minimization.

### 3.3 Testing the C Checker

In the following section, we discuss the formal proof of correctness for the Coq version of the RockSalt checker. But as noted above, in practice we expect to use the C version, partially shown in figures 5 and 6. Although this code is a rather direct translation from the Coq code, to gain further assurance, we did extensive testing, comparing both positive and negative examples against Google’s original checker.

For testing purposes, the `ncval` (Native Client Validator) command line tool was modified so that our verification routine can be used instead of Google’s. We ensured that both verifiers reject a set of hand-crafted unsafe programs, and we also ensured that they both accept a set of benchmark programs once processed by the NaCl version of GCC which inserts appropriate no-ops and mask instructions. To work around the lack of floating-point support in our checker, we use the “`-msoft-float`” flag so that GCC avoids generating floating-point instructions. The benchmark programs were drawn from the same set as used in CompCert [18] and include an implementation of AES, SHA1, a virtual machine, fractal computation, a Perl interpreter, and 16 other programs representing more than 4,000 lines of code. We also used `Csmith` [36] to automatically generate C programs, and compiled them with NaCl’s version of GCC. We then verified that our driver and Google’s always agreed on a program’s safety. Using this method we have verified over two thousand small C programs.

Finally, we measured the time it takes to check binaries using both our C checker and Google’s original code. For the small benchmarks mentioned above, there was no measurable difference in checking times. However, on an artificially generated C program of about 200,000 lines of code, running on a 2.6 GHz Intel Xeon core, Google’s checker took 0.90 seconds and our checker took 0.24 seconds (averaging over one hundred runs). Consequently, we believe that RockSalt is competitive with Google’s approach.

## 4. Proof of Correctness for the Checker

After building and testing the checker, we wanted to prove its correctness with respect to the sandbox policy. That is, we wanted a proof that if the checker returns TRUE for a given input binary, and if that binary is loaded and executed in an appropriate environment (in particular, where the code and data segments are disjoint), then executing the binary would ensure that only the prescribed data segments are read and written, and control only transfers within the prescribed code segment.

At a high-level, our proof shows that at every step of the program execution, the values of the segment registers are the same as those in the initial state, and furthermore, that the bytes that make up the code-segment are the same bytes that were analyzed by the checker. These invariants are sufficient to show NaCl’s sandbox policy is not violated. Furthermore, the checker should have ruled out system calls and other instructions that are not allowed. But of course, formalizing this argument requires a much more detailed

set of invariants that connect the matching work done in the checker to the semantics, along with the issues of alignment, masking, and jump-destination checks.

We begin by defining the notion of an *appropriate* machine state:

**DEFINITION 1.** *A machine state is appropriate when:*

1. *the original data and code segments are disjoint,*
2. *the DS, SS, and GS segment registers point to their respective original segments,*
3. *the CS segment registers point to the original code segment,*
4. *the program counter points within the code segment, and*
5. *the original bytes of the program are stored in the code segment.*

Appropriateness captures the key data invariants that we need to maintain throughout execution of the program. We augment these data invariants with a predicate on the program counter to reach the definition of a *locally-safe* machine state:

**DEFINITION 2.** *A machine state is locally-safe when it is appropriate and the program counter holds an address corresponding to the start of an instruction that was matched by the `verify` process using one of the three generated DFAs.*

In other words, for a locally safe state, the pc is marked as `valid`.

We would like to argue that, starting from a locally-safe state, we can always execute an instruction and end up in a locally-safe state. This would imply that the segment registers have not changed, that the code has not changed, that any read or write done by the instruction would be limited to the original data segments, and that control remains within the original code segment.

Alas, we do not immediately reach a locally-safe state after executing one instruction. The problem is that our `MaskedJump` DFA operates over two instructions (the mask of the register, followed by the indirect jump). Thus, we introduce the notion of a *k-safe* state:

**DEFINITION 3.** *An appropriate state  $s$  is  $k$ -safe when  $k > 0$  and, for any  $s'$  such that  $s \rightarrow s'$ , either  $s'$  is locally-safe or  $s'$  is  $(k - 1)$ -safe.*

With the definitions given above, it suffices to show that if a state is locally-safe, then it is also  $k$ -safe for some  $k$  (and in fact,  $k$  is either 1 or 2). Indeed, each locally-safe state  $s$  should be  $k$ -safe for some  $k$ : if  $s \rightarrow s'$  then either  $s'$  is locally-safe or we executed the mask of a `MaskedJump` and we should be in an appropriate state, ready to execute a branch instruction that will target a masked (and therefore valid) address. Then, assuming the computation starts in a locally-safe state (e.g., with the pc at any valid address), it is easy to see that the code cannot step to a state where the segment registers have changed, or the bytes in the code segment have changed.

**THEOREM 1.** *If  $s$  is locally-safe, then it is also  $k$ -safe for some  $k$ .*

Since a locally-safe instruction has a program counter drawn from the set of `valid` instructions, and since the verifier did not return FALSE, we can conclude that a prefix of the bytes starting at this address matches one of the three DFAs. We must then argue that for each class of instructions that match the DFAs, after executing the instruction, we either end up in a locally-safe state or else after executing one more instruction, end up in a locally safe-state.

In the Section 4.1, we sketch the connection we formalized between the DFAs and a set of inversion principles that characterize the possible instructions that they can match. These principles allow us to do a case analysis on a subset of the possible instructions. For example, in the case that the `MaskedJump` DFA matches, we know that the bytes referenced by the program counter must decode into

a masking operation on some register  $r$ , followed by bytes that decode into a jump or call to register  $r$ . The proof proceeds by case analysis for each of the three DFAs utilizing these inversion principles.

The easiest (though largest) case is when the `NoControlFlow` DFA has the successful match. We prove three properties for each non-control-flow instruction  $I$  that the inversion principle gives us:

- (1) executing  $I$  does not modify segment registers;
- (2) executing  $I$  modifies only the data segments' memory;
- (3) after executing  $I$ , the new program counter is equal to the old program counter plus the length of  $I$ .

For the most part, arguing these cases is simple: For the first property, we simply iterate over the generated list of RTLS for  $I$  and ensure there are no writes to the segment registers. The second property follows from the inversion principles which forbid the use of a segment override prefix, and the third property follows from the semantics of non-control-flow instructions. From these three facts, it follows that after executing the instruction, we are immediately in a locally-safe state. That is, the original state was 1-safe.

In the case where  $I$  was matched by `DirectJump`, we must argue that the final loop in `verify` ensures that the target of the jump is `valid`. Of course, we must also show that the segment registers are preserved, the code is preserved, etc. But then we can again argue that the original state was 1-safe.

For the `MaskedJump` case, we must argue that the state is 2-safe. The inversion principle for the DFA restricts the first instruction to an AND of a particular register  $r$  with a constant that ensures after the step, the value of  $r$  is aligned on a 32-byte boundary, the segments are preserved, and the pc points to bytes within the code segment that decode into either a jump or call through  $r$ . We then argue that this state is 1-safe. Since the destination of the jump or call is 32-byte aligned, the final loop of the verifier has checked that this address is `valid`. Consequently, it is easy to show that we end up in a locally-safe state.

#### 4.1 Inverting the DFAs

A critical piece in our proof of correctness is the relationship between the DFAs generated from the `NoControlFlow`, `DirectJump`, and `MaskedJump` regular expressions and our semantics for machine instructions. We sketch the key results that we have proven here.

One theorem specifies the connection between a regular expression, the DFA it generates, and the `match` procedure:

**THEOREM 2.** *If  $r$  is a regular expression, and  $D$  is the DFA generated from that regular expression, then executing `match` on  $D$  with a sequence of bytes  $b_1, \dots, b_n$  will return `true` if there is some  $j \leq n$  such that the string  $b_1, \dots, b_j$  is in the denotation of  $r$ .*

The theorem requires proving that our DFA construction process, where we iteratively calculate all derivatives, produces a well-formed DFA with respect to  $r$ . Here, a well-formed DFA basically provides a mapping from states to derivatives of  $r$  that respect certain closure properties. Fortunately, the algebraic construction of the DFA makes proving this result relatively straightforward. The theorem also requires showing that running `match` on  $D$  with  $b_1, \dots, b_n$  is correct which entails, among other things, showing that the array accesses are in bounds, and that when we return `TRUE`, we are in a state that corresponds to the derivative of the regular expression with respect to the string  $b_1, \dots, b_j$ .

Another key set of lemmas show that the languages accepted by the regular expressions are subsets of the languages accepted by our `x86grammar`. Additionally, we must prove an inversion principle for each regular expression that characterizes the possible abstract syntax we get when we run the semantics on the bytes. For

example, we must show that `DirectJump` only matches bytes that when parsed, produce either (near) `JMP`, `Jcc`, or `CALL` instructions with an immediate operand. Fortunately, proving the language containment property and inversion principles is simple to do using the denotational semantics for grammars.

One of the most difficult properties to prove about the decoder was the uniqueness of parsing. In particular, we needed to show that each bit pattern corresponded to at most one instruction, and no instruction's bit pattern was a prefix of another instruction's bit pattern—i.e., that our `x86grammar` was unambiguous. A naive approach, where we simply explore all possible bit patterns is obviously intractable, when there are instructions up to 15 bytes long. Another approach is to construct a DFA for the grammar and then show that each accepting state has at most one semantic value associated with it. While this is possible, the challenge is getting Coq to symbolically evaluate the DFA construction and reduce the semantic actions in a reasonable amount of time<sup>4</sup>.

Consequently, we constructed a simple procedure that checks whether the intersection of two grammars is empty. The procedure, which only succeeds on star-free grammars (stripped of their semantic actions) works by generalizing the notion of a derivative from characters to star-free regular expressions:

$$\begin{aligned} \text{Deriv } g \text{ Eps} &= g \\ \text{Deriv } g \text{ (Char } c\text{)} &= \text{deriv}_c g \\ \text{Deriv } g \text{ Any} &= \text{DrvAny } g \\ \text{Deriv } g \text{ Void} &= \text{Void} \\ \text{Deriv } g \text{ (Alt } g_1 g_2\text{)} &= \text{Alt}(\text{Deriv } g \text{ } g_1)(\text{Deriv } g \text{ } g_2) \\ \text{Deriv } g \text{ (Cat } g_1 g_2\text{)} &= \text{Deriv}(\text{Deriv } g \text{ } g_1) \text{ } g_2 \end{aligned}$$

where

$$\begin{aligned} \text{DrvAny Any} &= \text{Eps} \\ \text{DrvAny (Char } c\text{)} &= \text{Eps} \\ \text{DrvAny Eps} &= \text{Void} \\ \text{DrvAny Void} &= \text{Void} \\ \text{DrvAny (Alt } g_1 g_2\text{)} &= \text{Alt}(\text{DrvAny } g_1)(\text{DrvAny } g_2) \\ \text{DrvAny (Cat } g_1 g_2\text{)} &= \text{Alt}(\text{Cat}(\text{DrvAny } g_1) \text{ } g_2) \\ &\quad (\text{Cat}(\text{null } g_1)(\text{DrvAny } g_2)) \end{aligned}$$

When it is defined, it is easy to show that:

$$\text{Deriv } g_1 g_2 = \{s_2 \mid \exists s_1. s_1 \in g_2 \wedge s_1 s_2 \in g_1\}$$

and thus, when  $\text{Deriv } g_1 g_2 \rightarrow \text{Void}$ , we can conclude that there is no string in the intersection of the domains of  $g_1$  and  $g_2$ , and furthermore,  $g_2$ 's strings are not a prefix of those in  $g_1$ . This allowed us to easily prove (through Coq's symbolic evaluation) that the `x86grammar` is unambiguous: We simply recursively descend into the grammar, and each time we encounter an `Alt`, check that the intersection of the two sub-grammars is empty.

## 5. Related Work

With the growing interest in verification of software tools, formal models of processors that support machine-checked proofs have become a hot topic. Often, these models have limitations, not because of any inherent design flaw, but rather because they are meant only to prove specific properties. For example, work on formal verification of compilers [6, 18] only needs to consider the subset of the instructions that compilers use. Moreover, these compilers emit assembly instructions and do not prove semantics preservation all the way down to machine code, so their model leaves out the tricky problem of decoding. The same kinds of limitations exist for the processor models used in the formal verification of operating systems [7]. Some projects focus on one specific part of the model, for

<sup>4</sup> Recall that that the DFAs generated for the NaCl checker strip the semantic actions, so they do not need to worry about reducing semantic actions.

instance the media instructions [16], and some others [22] model just a few instructions mostly as a proof of concept. Even though we are focused here on NaCl verification, our long term goal is to develop a general model of the x86 so we have tried hard to achieve a more open and scalable design.

There are several projects focused on the development of general formal models of processors. Some projects have considered the formalizations of RISC processors [2, 13, 23]. As noted, developing a formal model for the x86 poses many new problems, partly because decoding is significantly more complex, but also for the definition and validation of such a vast number of instructions (over 1,000) with so many variations, from addressing modes to prefixes.

One model close in spirit to our own is the Y86 formalization in ACL2 by Ray [31]. Like our model, Ray’s provides an executable simulator. However, the Y86 is a much smaller fragment (about 30 instructions), and has a much simpler instruction encoding (*e.g.*, no prefixes).

Perhaps the closest related research project, and the one from which we took much inspiration in our design, is the work on modeling x86 multi-processor memory models [27, 32, 34]. This work comes with a formal model of about 20 instructions, and we borrowed many of the ideas, such as the use of high-level grammars for specifying the decoder. However, their focus was on issues of non-determinism where it is seemingly more natural to use predicates to describe the possible behaviors of programs. The price paid is that validation requires symbolic evaluation and theorem proving to compare abstract machine states to concrete ones. Although this was largely automated, we believe that our functional approach provides a more scalable way to test the model. Indeed, we have been able to run three orders of magnitude more tests. On the other hand, it remains to be seen how effective our approach will be when we add support for concurrency.

Our decoder, formalized in Coq, uses parsers generated from regular expressions using the idea of derivatives. Others have formalized derivative-based regular expression matching [3] but not parsing. However, more general parser generators for algorithms such as SLR and LR have recently been formalized [4, 15].

The original idea for Software-based fault isolation (SFI) was introduced by Wahbe *et. al.* [35] in the context of a RISC machine. This work used an invariant on dedicated registers to ensure that all reads, writes, and jumps were appropriately isolated. Of course, parsing was not a problem because instructions had a uniform length. As noted earlier, McCamant and Morrisett [22] introduced the idea of the alignment constraint to handle variable-length instruction sets. In that paper, they formalized a small subset of the x86 (7 instructions) using ACL2 and proved that their high-level invariants were respected by those instructions, but did not prove the correctness of their checker. In fact, even with the small number of instructions, Kroll and Dean found a number of bugs in the decoder [17], which reinforces our argument that one should be wary of a trusted decoder or disassembler.

Pilkiewicz [28] developed a formally verified SFI checker in Coq for a simple assembly language.

There has been much subsequent work on stronger policies than SFI, including CFI [1] and XFI [12]. Some of this work has been formalized, but typically for RISC machines and in a context where decoding is ignored.

## 6. Future work & Conclusions

We have presented a formal model for a significant subset of the x86, and a new formally verified checker for Native Client called RockSalt. The primary challenge in this work was building a model for an architecture as complicated as the x86. Although we only managed to model a small subset, we believe that the design is

relatively robust thanks to our ability to extract and test executable code. The experience in using the model to reason about a simple but real policy such as NaCl’s sandbox, provides some assurance that the model will be useful for reasoning in other contexts.

### 6.1 Future Work

As explained before, our x86 model is far from complete. We do not yet handle floating-point instructions, system programming instructions, nor any of the MMX, SSEn, 3dNow! or IA-64 instructions. On the other hand, we have managed to cover enough instructions that we can compile real applications and run them through the simulator. Moving forward, we would like to extend the model to cover at least those instructions that are used by compilers.

Our model of machine states is also overly simple. For example, we do not yet model concurrency, interrupts, or page tables. However, we believe that the use of RTL as a staging language makes it easier to add support for those features. For example, to model multiple processors and the total-store order (TSO) memory consistency model [34], we believe that it is sufficient to add a store buffer to the machine state for each processor. Of course, validating a concurrent model will present new challenges.

We believe that the use of domain-specific languages will further facilitate re-use and help to find and eliminate bugs. For example, one could imagine embedding these languages in other proof assistants (HOL, ACL2, *etc.*) to support portability of the specification across formal systems.

We would also like to close the gap on RockSalt so that the C code, derived from our verified Coq code, is itself verified and compiled with a proven-correct compiler such as CompCert. In fact, one fun idea is to simply bypass the compiler and write the checker directly in x86 assembly to see how easy it is to turn the process in on itself. Finally, there are richer classes of policies, such as XFI, for which we would like to write checkers and prove correctness.

### 6.2 Lessons Learned

The basic idea of using domain-specific languages to build a scalable semantics worked well for us. In our first iteration of the model, we tried to directly interpret x86 instructions, but soon realized that any reasoning work would be proportional to the number of distinct instructions. Compiling instructions to a small RISC-like core simplified our reasoning, and at the same time, made it easier to factor the model into smaller, more re-usable components.

One surprising aspect of the work was that the pressure to provide reasoning principles for parsers forced us to treat the problem more algebraically than is typically done. In particular, the use of derivatives, which operate directly on the abstract syntax of grammars, made our reasoning much simpler than it would be with graphs.

It goes without saying that constructing machine-checked proofs is still very hard. The definitions for our x86 model and NaCl checker are about 5,000 lines of heavily commented Coq code, but the RockSalt proofs are another 10,000 lines. Of course, many of these proofs will be useful in other settings (*e.g.*, that the decoder is unambiguous) but the ratio is still quite large. One reason for this is that reasoning about certain theories (*e.g.*, bit vectors) is still rather tedious in Coq, especially when compared to modern SAT or SMT solvers. Yet, the dependent types and higher-order features of the language were crucial for constructing the model, much less proving deep properties about it.

For us, another surprising aspect of the work was the difference that comes with scale. We have a fair amount of experience modeling simple abstract machines with proof assistants. Doing a case split on five or even ten instructions and manually discharging the cases is reasonable. But once you have hundreds of cases, any of

which may change as you validate the model, such an approach is no longer tenable. Consequently, many of our proofs were actually done through some form of reflection. For example, to prove that the `x86grammar` is unambiguous, we constructed a computable function that tests for ambiguity and proved its correctness. In turn, this made it easier to add new instructions to the grammar. Frankly, we couldn't stomach the idea of proving the correctness of a handwritten `x86` decoder, and so we were forced into finding a better solution. In short, when a mechanized development reaches a certain size, we are forced to develop more automated and robust proof techniques.

## References

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM Conf. on Computer and Commun. Security*, CCS '05, pages 340–353. ACM, 2005.
- [2] J. Alglave, A. C. J. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Z. Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. of the Workshop on Declarative Aspects of Multicore Programming*, pages 13–24. ACM, 2009.
- [3] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa. Partial derivative automata formalized in Coq. In *Proc. of the 15th Intl. Conf. on Implementation and Application of Automata*, number 6482 in CIAA '10, pages 59–68. Springer-Verlag, Aug. 2010.
- [4] A. Barthwal and M. Norrish. Verified, executable parsing. In *European Symp. on Programming*, ESOP '09, pages 160–174. LNCS, 2009.
- [5] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.
- [6] A. Chlipala. A verified compiler for an impure functional language. In *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–106. ACM, 2010.
- [7] D. Cock. Lyrebird: assigning meanings to machines. In *Proc. of the 5th Intl. Conf. on Systems Software Verification*, SSV'10, pages 6–15. USENIX Association, 2010.
- [8] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proc. of the 22nd Intl. Conf. on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42. Springer-Verlag, 2009.
- [9] Coq development team. The Coq proof assistant. <http://coq.inria.fr/>, 1989–2012.
- [10] L. Correnson, Z. Dargaye, and A. Pacalet. *WP plug-in manual*. CEA LIST.
- [11] J. Dias and N. Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proc. of the 37th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '10, pages 403–416. ACM, 2010.
- [12] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, OSDI '06, pages 75–88. USENIX Association, 2006.
- [13] A. C. J. Fox and M. O. Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 243–258. Springer, 2010.
- [14] Intel Corporation. *Pentium Processor Family Developers Manual*, volume 3. Intel Corporation, 1996.
- [15] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *European Symp. on Programming*, ESOP '12. Springer, 2012. To appear.
- [16] W. A. H. Jr. and S. Swords. Centaur technology media unit verification. In *Computer Aided Verification*, 21st Intl. Conf., volume 5643 of *LNCS*, pages 353–367. Springer, 2009.
- [17] J. Kroll and D. Dean. BakerSField: Bringing software fault isolation to x64. <http://www.cs.princeton.edu/~kroll/papers/bakersfield-sfi.pdf>.
- [18] X. Leroy. Formal verification of a realistic compiler. *Commun. of the ACM*, 52(7):107–115, 2009.
- [19] J. Lim. *Transformer Specification Language: A System for Generating Analyzers and its Applications*. PhD thesis, University of Wisconsin-Madison, May 2011.
- [20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.
- [21] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proc. of the 18th Intl. Symp. on Software Testing and Analysis*, pages 261–272. ACM, 2009.
- [22] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. of the 15th Conf. on USENIX Security Symp.*, pages 209–224. USENIX Association, 2006.
- [23] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher order logic. In *Automated Deduction - CADE-17, 17th Intl. Conf. on Automated Deduction*, volume 1831 of *LNCS*, pages 7–24. Springer, 2000.
- [24] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: a functional pearl. In *Proc. of the 16th ACM SIGPLAN Intl. Conf. on Functional Programming*, ICFP '11, pages 189–195. ACM, 2011.
- [25] Native Client team. Native client security contest. <http://code.google.com/contests/nativeclient-security/index.html>, 2009.
- [26] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19:173–190, March 2009.
- [27] S. Owens, P. Böhm, F. Z. Nardelli, and P. Sewell. Lem: A lightweight tool for heavyweight semantics. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 363–369. Springer, 2011.
- [28] A. Pilkiewicz. A proved version of the inner sandbox. In *native-client-discuss mailing list*, April 2011.
- [29] N. Ramsey and J. W. Davidson. Machine descriptions to build tools for embedded systems. In *Languages, Compilers, and Tools for Embedded Systems*, volume 1474 of *LNCS*, pages 176–192. Springer, 1998.
- [30] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19(3):492–524, 1997.
- [31] S. Ray. Towards a formalization of the X86 instruction set architecture. Technical Report TR-08-15, Department of Computer Science, University of Texas at Austin, March 2008.
- [32] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 379–391. ACM, 2009.
- [33] M. Seaborn. A DFA-based x86-32 validator for Native Client. In *native-client-discuss mailing list*, June 2011.
- [34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
- [35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of the 14th ACM Symp. on Operating Systems Principles*, SOSP '93, pages 203–216. ACM, 1993.
- [36] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 283–294. ACM, 2011.
- [37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. of the ACM*, 53(1):91–99, 2010.
- [38] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *11th Intl. Conf. on Embedded Software*. ACM, 2011.

---

# Augur: Data-Parallel Probabilistic Modeling

---

Jean-Baptiste Tristan<sup>1</sup>, Daniel Huang<sup>2</sup>, Joseph Tassarotti<sup>3</sup>,  
Adam Pocock<sup>1</sup>, Stephen J. Green<sup>1</sup>, Guy L. Steele, Jr<sup>1</sup>

<sup>1</sup>Oracle Labs {jean.baptiste.tristan, adam.pocock,  
stephen.x.green, guy.steele}@oracle.com

<sup>2</sup>Harvard University dehuang@fas.harvard.edu

<sup>3</sup>Carnegie Mellon University jtassaro@cs.cmu.edu

## Abstract

Implementing inference procedures for each new probabilistic model is time-consuming and error-prone. Probabilistic programming addresses this problem by allowing a user to specify the model and then automatically generating the inference procedure. To make this practical it is important to generate high performance inference code. In turn, on modern architectures, high performance requires parallel execution. In this paper we present Augur, a probabilistic modeling language and compiler for Bayesian networks designed to make effective use of data-parallel architectures such as GPUs. We show that the compiler can generate data-parallel inference code scalable to thousands of GPU cores by making use of the conditional independence relationships in the Bayesian network.

## 1 Introduction

Machine learning, and especially probabilistic modeling, can be difficult to apply. A user needs to not only design the model, but also implement an efficient inference procedure. There are many different inference algorithms, many of which are conceptually complicated and difficult to implement at scale. This complexity makes it difficult to design and test new models, or to compare inference algorithms. Therefore any effort to simplify the use of probabilistic models is useful.

Probabilistic programming [1], as introduced by BUGS [2], is a way to simplify the application of machine learning based on Bayesian inference. It allows a separation of concerns: the user specifies *what* needs to be learned by describing a probabilistic model, while the runtime automatically generates the *how*, i.e., the inference procedure. Specifically the programmer writes code describing a probability distribution, and the runtime automatically generates an inference algorithm which samples from the distribution. Inference itself is a computationally intensive and challenging problem. As a result, developing inference algorithms is an active area of research. These include deterministic approximations (such as variational methods) and Monte Carlo approximations (such as MCMC algorithms). The problem is that most of these algorithms are conceptually complicated, and it is not clear, especially to non-experts, which one would work best for a given model.

In this paper we present *Augur*, a probabilistic modeling system, embedded in Scala, whose design is guided by two observations. The first is that if we wish to benefit from advances in hardware we *must* focus on producing highly *parallel* inference algorithms. We show that many MCMC inference algorithms are highly data-parallel [3, 4] *within a single Markov Chain*, if we take advantage of the conditional independence relationships of the input model (e.g., the assumption of i.i.d. data makes the likelihood independent across data points). Moreover, we can *automatically* generate good data-parallel inference with a compiler. This inference runs efficiently on common highly parallel architectures such as Graphics Processing Units (GPUs). We note that parallelism brings interesting trade-offs to MCMC performance as some inference techniques generate less parallelism and thus scale poorly.

The second observation is that a high performance system begins by selecting an appropriate inference algorithm, and this choice is often the hardest problem. For example, if our system only implements Metropolis-Hastings inference, there are models for which our system will be of no use, even given large amounts of computational power. We must design the system so that we can include the latest research on inference while reusing pre-existing analyses and optimizations. Consequently, we use an intermediate representation (IR) for probability distributions that serves as a target for modeling languages and as a basis for inference algorithms, allowing us to easily extend the system. We will show this IR is key to scaling the system to very large networks.

We present two main results: first, some inference algorithms are highly data-parallel and a compiler can *automatically* generate effective GPU implementations; second, it is important to use a symbolic representation of a distribution rather than explicitly constructing a graphical model in memory, allowing the system to scale to much larger models (such as LDA).

## 2 The Augur Language

We present two example model specifications in Augur, latent Dirichlet allocation (LDA) [5], and a multivariate linear regression model. The supplementary material shows how to generate samples from the models, and how to use them for prediction. It also contains six more example probabilistic models in Augur: polynomial regression, logistic regression, a categorical mixture model, a Gaussian Mixture Model (GMM), a Naive Bayes Classifier, and a Hidden Markov Model (HMM). Our language is similar in form to BUGS [2] and Stan [6], except our statements are *implicitly parallel*.

### 2.1 Specifying a Model

The LDA model specification is shown in Figure 1a. The probability distribution is a Scala object (`object LDA`) composed of two declarations. First, we declare the support of the probability distribution as a class named `sig`. The support of the LDA model is composed of four arrays, one each for the distribution of topics per document (`theta`), the distribution of words per topic (`phi`), the topics assignments (`z`), and the words in the corpus (`w`). The support is used to store the inferred model parameters. These last two arrays are flat representations of ragged arrays, and thus we do not require the documents to be of equal length. The second declaration specifies the probabilistic model for LDA in our embedded domain specific language (DSL) for Bayesian networks. The DSL is marked by the `bayes` keyword and delimited by the enclosing brackets. The model first declares the parameters of the model: `K` for the number of topics, `V` for the vocabulary size, `M` for the number of documents, and `N` for the array of document sizes. In the model itself, we define the hyperparameters (values `alpha` and `beta`) for the Dirichlet distributions and sample `K` Dirichlets of dimension `V` for the distribution of words per topic (`phi`) and `M` Dirichlets of dimension `K` for the distribution of topics per document (`theta`). Then, for each word in each document, we draw a topic `z` from `theta`, and finally a word from `phi` conditioned on the topic we drew for `z`.

The regression model in Figure 1b is defined in the same way using similar language features. In this example the support comprises the  $(x, y)$  data points, the weights `w`, the bias `b`, and the noise `tau`. The model uses an additional `sum` function to sum across the feature vector.

### 2.2 Using a Model

Once a model is specified, it can be used as any other Scala object by writing standard Scala code. For instance, one may want to use the LDA model with a training corpus to learn a distribution of words per topic and then use it to learn the per-document topic distribution of a test corpus. In the supplementary material we provide a code sample which shows how to use an Augur model for such a task. Each Augur model forms a distribution, and the runtime system generates a `Dist` interface which provides two methods: `map`, which implements maximum a posteriori estimation, and `sample`, which returns a sequence of samples. Both of these calls require a similar set of arguments: a list of additional variables to be observed (e.g., to fix the `phi` values at test time in LDA), the model hyperparameters, the initial state of the model support, the model support that stores the inferred parameters, the number of MCMC samples and the chosen inference method.

```

1 object LDA {
2   class sig(var phi: Array[Double],
3             var theta: Array[Double],
4             var z: Array[Int],
5             var w: Array[Int])
6   val model = bayes {
7     (K:Int,V:Int,M:Int,N:Array[Int]) => {
8       val alpha = vector(K, 0.1)
9       val beta = vector(V, 0.1)
10      val phi = Dirichlet(V,beta).sample(K)
11      val theta = Dirichlet(K,alpha).sample(M)
12      val w =
13        for(i <- 1 to M) yield {
14          for(j <- 1 to N(i)) yield {
15            val z: Int =
16              Categorical(K,theta(i)).sample()
17              Categorical(V,phi(z)).sample()
18          }
19        observe(w)
20    }}}

```

```

1 object LinearRegression {
2   class sig(var w: Array[Double],
3             var b: Double,
4             var tau: Double,
5             var x: Array[Double],
6             var y: Array[Double])
7   val model = bayes {
8     (K:Int,N:Int,l:Double,u:Double) => {
9       val w = Gaussian(0,10).sample(K)
10      val b = Gaussian(0,10).sample()
11      val tau = InverseGamma(3.0,1.0).sample()
12      val x = for(i <- 1 to N) yield {
13        Uniform(l,u).sample(K)
14      }
15      val y = for(i <- 1 to N) yield {
16        val phi = for(j <- 1 to K) yield
17          w(j) * x(i)(j)
18        Gaussian(phi.sum + b,tau).sample()
19      }
20      observe(x, y)
21    }}}

```

(a) A LDA model in Augur. The model specifies the distribution  $p(\phi, \theta, z \mid w)$ .

(b) A multivariate regression in Augur. The model specifies the distribution  $p(\mathbf{w}, b, \tau \mid x, y)$ .

Figure 1: Example Augur programs.

### 3 System Architecture

We now describe how a model specification is transformed into CUDA code running on a GPU. Augur has two distinct compilation phases. The first phase transforms the block of code following the `bayes` keyword into our IR for probability distributions, and occurs when `scalac` is invoked. The second phase happens at runtime, when a method is invoked on the model. At that point, the IR is transformed, analyzed, and then CUDA code is emitted, compiled, and executed.

Due to these two phases, our system is composed of two distinct components that communicate through the IR: the front end, where the DSL is converted into the IR, and the back end, where the IR is compiled down to the chosen inference algorithm (currently Metropolis-Hastings, Gibbs sampling, or Metropolis-Within-Gibbs). We use the Scala macro system to define the modeling language in the front end. The macro system allows us to define a set of functions (called “macros”) that are executed by the Scala compiler on the code enclosed by the macro invocation. We currently focus on Bayesian networks, but other DSLs (e.g., Markov random fields) could be added without modifications to the back end. The implementation of the macros to define the Bayesian network language is conceptually uninteresting so we omit further details.

Separating the compilation into two distinct phases provides many advantages. As our language is implemented using Scala’s macro system, it provides automatic syntax highlighting, method name completion, and code refactoring in any IDE which supports Scala. This improves the usability of the DSL as we require no special tools support. We also use Scala’s parser, semantic analyzer (e.g., to check that variables have been defined), and type checker. Additionally we benefit from `scalac`’s optimizations such as constant folding and dead code elimination. Then, because we compile the IR to CUDA code *at run time*, we know the values of all the hyperparameters and the size of the dataset. This enables better optimization strategies, and also gives us important insights into how to extract parallelism (Section 4.2). For example, when compiling LDA, we know that the number of topics is much smaller than the number of documents and thus parallelizing over documents produces more parallelism than parallelizing over topics. This is analogous to JIT compilation in modern runtime systems where the compiler can make different decisions at runtime based upon the program state.

### 4 Generation of Data-Parallel Inference

We now explain how Augur generates data-parallel samplers by exploiting the conditional independence structure of the model. We will use the two examples from Section 2 to explain how the compiler analyzes the model and generates the inference code.

When we invoke an inference procedure on a model (e.g., by calling `model.map`), Augur compiles the IR into CUDA inference code for that model. Our aim with the IR is to make the parallelism explicit in the model and to support further analysis of the probability distributions contained within. For example, a  $\prod$  indicates that each sub-term in the expression can be evaluated in parallel. Informally, our IR expressions are generated from this Backus-Naur Form (BNF) grammar:

$$P ::= p(\vec{X}) \mid p(\vec{X} \mid \vec{X}) \mid PP \mid \frac{1}{P} \mid \prod_i^N P \mid \int_X P dx \mid \{P\}_c$$

The use of a symbolic representation for the model is key to Augur’s ability to scale to large networks. Indeed, as we show in the experimental study (Section 5), popular probabilistic modeling systems such as JAGS [7] or Stan [8] reify the graphical model, resulting in unreasonable memory consumption for models such as LDA. However, a consequence of our symbolic representation is that it is more difficult to discover conjugacy relationships, a point we return to later.

#### 4.1 Generating data-parallel MH samplers

To use Metropolis-Hastings (MH) inference, the compiler emits code for a function  $f$  that is proportional to the distribution to be sampled. This code is then linked with our library implementation of MH. The function  $f$  is the product of the prior and the model likelihood and is extracted automatically from the model specification. In our regression example this function is:  $f(\mathbf{x}, \mathbf{y}, \tau, b, \mathbf{w}) = p(b)p(\tau)p(\mathbf{w})p(\mathbf{x})p(\mathbf{y} \mid \mathbf{x}, b, \tau, \mathbf{w})$  which we rewrite to

$$f(\mathbf{x}, \mathbf{y}, \tau, b, \mathbf{w}) = p(b)p(\tau) \left( \prod_k^K p(w_k) \right) \left( \prod_n^N p(x_n)p(y_n \mid \mathbf{x}_n \cdot \mathbf{w} + b, \tau) \right)$$

In this form, the compiler knows that the distribution factorizes into a large number of terms that can be evaluated in parallel and then efficiently multiplied together. Each  $(x, y)$  contributes to the likelihood independently (i.e., the data is i.i.d.), and each pair can be evaluated in parallel and the compiler can optimize accordingly. In practice, we work in log-space, so we perform summations. The compiler then generates the CUDA code to evaluate  $f$  from the IR. This code generation step is conceptually simple and we will not explain it further.

It is interesting to note that the code scales well despite the simplicity of this parallelization: there is a large amount of parallelism because it is roughly proportional to the number of data points; uncovering the parallelism in the code does not increase the amount of computation performed; and the ratio of computation to global memory accesses is high enough to hide the memory latency.

#### 4.2 Generating data-parallel Gibbs samplers

Alternatively we can generate a Gibbs sampler for conjugate models. We would prefer to generate a Gibbs sampler for LDA, as an MH sampler will have a very low acceptance ratio. To generate a Gibbs sampler, the compiler needs to figure out how to sample from each univariate conditional distribution. As an example, to draw  $\theta_m$  as part of the  $(\tau + 1)$ th sample, the compiler needs to generate code that samples from the following distribution

$$p(\theta_m^{\tau+1} \mid w^{\tau+1}, z^{\tau+1}, \theta_1^{\tau+1}, \dots, \theta_{m-1}^{\tau+1}, \theta_{m+1}^{\tau}, \dots, \theta_M^{\tau}).$$

As we previously explained, our compiler uses a symbolic representation of the model: the advantage is that we can scale to large networks, but the disadvantage is that it is more challenging to uncover conjugacy and independence relationships between variables. To accomplish this, the compiler uses an algebraic rewrite system that aims to rewrite the above expression in terms of expressions it knows (i.e., the joint distribution of the model). We show a few selected rules below to give a flavor of the rewrite system. The full set of 14 rewrite rules are in the supplementary material.

$$\begin{array}{ll} (a) \frac{P}{P} \Rightarrow 1 & (c) \prod_i^N P(x_i) \Rightarrow \prod_i^N \{P(x_i)\}_{q(i)=T} \prod_i^N \{P(x_i)\}_{q(i)=F} \\ (b) \int P(x) Q dx \Rightarrow Q \int P(x) dx & (d) P(x \mid y) \Rightarrow \frac{P(x,y)}{\int P(x,y) dx} \end{array}$$

Rule (a) states that like terms can be canceled. Rule (b) says that terms that do not depend on the variable of integration can be pulled out of the integral. Rule (c) says that we can partition a product

over  $N$  terms into two products, one where a predicate  $q$  is true on the indexing variable and one where it is false. Rule (d) is a combination of the product and sum rule. Currently, the rewrite system is comprised of rules we found useful in practice, and it is easy to extend the system with more rules.

Going back to our example, the compiler rewrites the desired expression into the one below:

$$\frac{1}{Z} p(\theta_m^{\tau+1}) \prod_j^{N(m)} p(z_{mj} | \theta_m^{\tau+1})$$

In this form, it is clear that each  $\theta_1, \dots, \theta_m$  is independent of the others after conditioning on the other random variables. As a result, they may all be sampled in parallel.

At each step, the compiler can test for a conjugacy relation. In the above form, the compiler recognizes that the  $z_{mj}$  are drawn from a categorical distribution and  $\theta_m$  is drawn from a Dirichlet, and can exploit the fact that these are conjugate distributions. The posterior distribution for  $\theta_m$  is  $\text{Dirichlet}(\alpha + c_m)$  where  $c_m$  is a vector whose  $k$ th entry is the number of  $z$  of topic  $k$  from document  $m$ . Importantly, the compiler now knows that sampling each  $z$  requires a counting phase.

The case of the  $\phi$  variables is more interesting. In this case, we want to sample from

$$p(\phi_k^{\tau+1} | w^{\tau+1}, z^{\tau+1}, \theta^{\tau+1}, \phi_1^{\tau+1}, \dots, \phi_{k-1}^{\tau+1}, \phi_{k+1}^{\tau}, \dots, \phi_K^{\tau})$$

After applying the rewrite system to this expression, the compiler discovers that it is equal to

$$\frac{1}{Z} p(\phi_k) \prod_i^M \prod_j^{N(i)} \{p(w_i | \phi_{z_{ij}})\}_{k=z_{ij}}$$

The key observation that the compiler uses to reach this conclusion is the fact that the  $z$  are distributed according to a categorical distribution and are used to index into the  $\phi$  array. Therefore, they partition the set of words  $w$  into  $K$  disjoint sets  $w_1 \uplus \dots \uplus w_k$ , one for each topic. More concretely, the probability of words drawn from topic  $k$  can be rewritten in partitioned form using rule (c) as  $\prod_i^M \prod_j^{N(i)} \{p(w_{ij} | \phi_{z_{ij}})\}_{k=z_{ij}}$  as once a word's topic is fixed, the word depends upon only one of the  $\phi_k$  distributions. In this form, the compiler recognizes that it should draw from  $\text{Dirichlet}(\beta + c_k)$  where  $c_k$  is the count of words assigned to topic  $k$ . In general, the compiler detects this pattern when it discovers that samples drawn from categorical distributions are being used to index into arrays.

Finally, the compiler turns to analyzing the  $z_{ij}$ . It detects that they can be sampled in parallel but it does not find a conjugacy relationship. However, it discovers that the  $z_{ij}$  are drawn from discrete distributions, so the univariate distribution can be calculated exactly and sampled from. In cases where the distributions are continuous, it tries to use another approximate sampling method to sample from that variable.

One concern with such a rewrite system is that it may fail to find a conjugacy relation if the model has a complicated structure. So far we have found our rewrite system to be robust and it can find all the usual conjugacy relations for models such as LDA, GMMs or HMMs, but it suffers from the same shortcomings as implementations of BUGS when deeper mathematics are required to discover a conjugacy relation (as would be the case for instance for a non-linear regression). In the cases where a conjugacy relation cannot be found, the compiler will (like BUGS) resort to using Metropolis-Hastings and therefore exploit the inherent parallelism of the model likelihood.

Finally, note that the rewrite rules are applied deterministically and the process will always terminate with the same result. Overall, the cost of analysis is negligible compared to the sampling time for large data sets. Although the rewrite system is simple, it enables us to use a concise symbolic representation for the model and thereby scale to large networks.

### 4.3 Data-parallel Operations on Distributions

To produce efficient code, the compiler needs to uncover parallelism, but we also need a library of data-parallel operations for distributions. For instance, in LDA, there are two steps where we sample from many Dirichlet distributions in parallel. When drawing the per document topic distributions, each thread can draw a  $\theta_i$  by generating  $K$  Gamma variates and normalizing them [9]. Since the

number of documents is usually very large, this produces enough parallelism to make full use of the GPU’s cores. However, this may not produce sufficient parallelism when drawing the  $\phi_k$ , because the number of topics is usually small compared to the number of cores. Consequently, we use a different procedure which exposes more parallelism (the algorithm is given in the supplementary material). To generate  $K$  Dirichlet variates over  $V$  categories with concentration parameters  $\alpha_{11}, \dots, \alpha_{KV}$ , we first generate a matrix  $A$  where  $A_{ij} \sim \text{Gamma}(\alpha_{ij})$  and then normalize each row of this matrix. To sample the  $\theta_i$ , we could launch a thread per row. However, as the number of columns is much larger than the number of rows, we launch a thread to generate the gamma variates for each column, and then separately compute a normalizing constant for each row by multiplying the matrix with a vector of ones using CUBLAS. This is an instance where the two-stage compilation procedure (Section 3) is useful, because the compiler is able to use information about the relative sizes of  $K$  and  $V$  to decide that the complex scheme will be more efficient than the simple scheme.

This sort of optimization is not unique to the Dirichlet distribution. For example, when sampling a large number of multivariate normals by applying a linear transformation to a vector of normal samples, the strategy for extracting parallelism may change based on the number of samples to generate, the dimension of the multinormal, and the number of GPU cores. We found that issues like these were crucial to generating high-performance data-parallel samplers.

#### 4.4 Parallelism & Inference Tradeoffs

It is difficult to give a cost model for Augur programs. Traditional approaches are not necessarily appropriate for probabilistic inference because there are tradeoffs between faster sampling times and convergence which are not easy to characterize. In particular, different inference methods may affect the amount of parallelism that can be exploited in a model. For example, in the case of multivariate regression, we can use the Metropolis-Hastings sampler presented above, which lets us sample from all the weights in parallel. However, we may be better off generating a Metropolis-Within-Gibbs sampler where the weights are sampled one at a time. This reduces the amount of exploitable parallelism, but it may converge faster, and there may still be enough parallelism in each calculation of the Hastings ratio by evaluating the likelihood in parallel.

Many of the optimizations in the literature that improve the mixing time of a Gibbs sampler, such as blocking or collapsing, reduce the available parallelism by introducing dependencies between previously independent variables. In a system like Augur it is not always beneficial to eliminate variables (e.g., by collapsing) if it introduces more dependencies for the remaining variables. Currently Augur cannot generate a blocked or collapsed sampler, but there is interesting work on automatically blocking or collapsing variables [10] that we wish to investigate in the future. Our experimental results on LDA demonstrate this tradeoff between mixing and runtime. There we show that while a collapsed Gibbs sampler converges more quickly in terms of the number of samples compared to an uncollapsed sampler, the uncollapsed sampler converges more quickly in terms of runtime. This is due to the uncollapsed sampler having much more available parallelism. We hope that as more options and inference strategies are added to Augur, users will be able to experiment further with the tradeoffs of different inference methods in a way that would be too time-consuming to do manually.

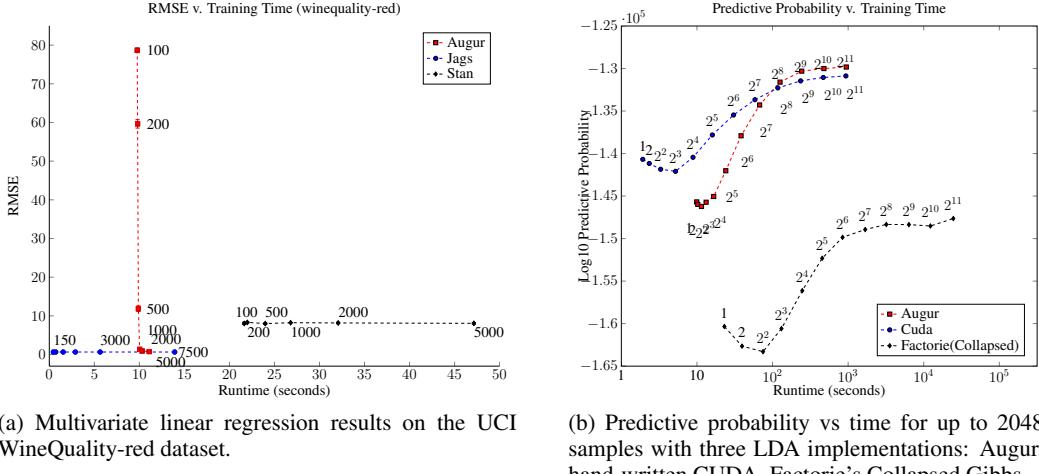
## 5 Experimental Study

We provide experimental results for the two examples presented throughout the paper and in the supplementary material for a Gaussian Mixture Model (GMM). More detailed information on the experiments can be found in the supplementary material.

To test multivariate regression and the GMM, we compare Augur’s performance to those of two popular languages for statistical modeling, JAGS [7] and Stan [8]. JAGS is an implementation of BUGS, and performs inference using Gibbs sampling, adaptive MH, and slice sampling. Stan uses a No-U-Turn sampler, a variant of Hamiltonian Monte Carlo. For the regression, we configured Augur to use MH<sup>1</sup>, while for the GMM Augur generated a Gibbs sampler. In our LDA experiments we also compare Augur to a handwritten CUDA implementation of a Gibbs sampler, and also to

---

<sup>1</sup>Augur could not generate a Gibbs sampler for regression, as the conjugacy relation for the weights is not a simple application of conjugacy rules[11]. JAGS avoids this issue by adding specific rules for linear regression.



(a) Multivariate linear regression results on the UCI WineQuality-red dataset.

(b) Predictive probability vs time for up to 2048 samples with three LDA implementations: Augur, hand-written CUDA, Factorie’s Collapsed Gibbs.

Figure 2: Experimental results on multivariate linear regression and LDA.

the collapsed Gibbs sampler [12] from the Factorie library [13]. The former is a comparison for an optimised GPU implementation, while the latter is a baseline for a CPU Scala implementation.

### 5.1 Experimental Setup

For the linear regression experiment, we used data sets from the UCI regression repository [14]. The Gaussian Mixture Model experiments used two synthetic data sets, one generated from 3 clusters, the other from 4 clusters. For the LDA benchmark, we used a corpus extracted from the simple English variant of Wikipedia, with standard stopwords removed. This corpus has 48556 documents, a vocabulary size of 37276 words, and approximately 3.3 million tokens. From that we sampled 1000 documents to use as a test set, removing words which appear only in the test set. To evaluate the model we measure the log predictive probability [15] on the test set.

All experiments ran on a single workstation with an Intel Core i7 4820k CPU, 32 GB RAM, and an NVIDIA GeForce Titan Black. The Titan Black uses the Kepler architecture. All probability values are calculated in double precision. The CPU performance results using Factorie are calculated using a single thread, as the multi-threaded samplers are neither stable nor performant in the tested release. The GPU results use all 960 double-precision ALU cores available in the Titan Black. The Titan Black has 2880 single-precision ALU cores, but single precision resulted in poor quality inference results, though the speed was greatly improved.

### 5.2 Results

In general, our results show that once the problem is large enough we can amortize Augur’s startup cost of model compilation to CUDA, nvcc compilation to a GPU binary, and copying the data to and from the GPU. This cost is approximately 9 seconds averaged across all our experiments. After this point Augur scales to larger numbers of samples in shorter runtimes than comparable systems. In this mode we are using Augur to find a likely set of parameters rather than generating a set of samples with a large *effective sample size* for posterior estimation. We have not investigated the effective sample size vs runtime tradeoff, though the MH approach we use for regression is likely to have a lower effective sample size than the HMC used in Stan.

Our linear regression experiments show that Augur’s inference is similar to JAGS in runtime and performance, and better than Stan. Augur takes longer to converge as it uses MH, though once we have amortized the compilation time it draws samples very quickly. The regression datasets tend to be quite small in terms of both number of random variables and number of datapoints, so it is harder to amortize the costs of GPU execution. However, the results are very different for models where the number of inferred parameters grows with the data set. In the GMM example in the supplementary,

we show that Augur scales to larger problems than JAGS or Stan. For 100,000 data points, Augur draws a thousand samples in 3 minutes while JAGS takes more than 21 minutes and Stan requires more than 6 hours. Each system found the correct means and variances for the clusters; our aim was to measure the scaling of runtime with problem size.

Results from the LDA experiment are presented in Figure 2b and use predictive probability to monitor convergence over time. We compute the predictive probability and record the time (in seconds) after drawing  $2^i$  samples, for  $i$  ranging from 0 to 11 inclusive. It takes Augur 8.1 seconds to draw its first sample for LDA. Augur’s performance is very close to that of the hand-written CUDA implementation, and much faster than the Factorie collapsed Gibbs sampler. Indeed, it takes the collapsed LDA implementation 6.7 hours longer than Augur to draw 2048 samples. We note that the collapsed Gibbs sampler appears to have converged after  $2^7$  samples, in approximately 27 minutes. The uncollapsed implementations converge after  $2^9$  samples, in approximately 4 minutes. We also implemented LDA in JAGS and Stan but they ran into scalability issues. The Stan version of LDA (taken from the Stan user’s manual[6]) uses 55 GB of RAM but failed to draw a sample in a week of computation time. We could not test JAGS as it required more than 128 GB of RAM. In comparison, Augur uses less than 1 GB of RAM for this experiment.

## 6 Related Work

Augur is similar to probabilistic modeling languages such as BUGS [16], Factorie [13], Dimple [17], Infer.net [18], and Stan [8]. This family of languages explicitly represents a probability distribution, restricting the expressiveness of the modeling language to improve performance. For example, Factorie, Dimple, and Infer.net provide languages for factor graphs enabling these systems to take advantage of specific efficient inference algorithms (e.g., Belief Propagation). Stan, while Turing complete, focuses on probabilistic models with continuous variables using a No-U-Turn sampler (recent versions also support discrete variables). In contrast, Augur focuses on Bayesian Networks, allowing a compact symbolic representation, and enabling the generation of data-parallel code.

Another family of probabilistic programming languages is characterized by their ability to express all computable generative models by reasoning over execution traces which implicitly represent probability distributions. These are typically a Turing complete language with probabilistic primitives and include Venture [19], Church [20], and Figaro [21]. Augur and the modeling languages described above are less expressive than these languages, and so describe a restricted set of probabilistic programs. However performing inference over program traces generated by a model, instead of the model support itself, makes it more difficult to generate an efficient inference algorithm.

## 7 Discussion

We show that it is possible to *automatically* generate parallel MCMC inference algorithms, and it is also possible to extract sufficient parallelism to saturate a modern GPU with thousands of cores. The choice of a Single-Instruction Multiple-Data (SIMD) architecture such as a GPU is central to the success of Augur, as it allows many parallel threads with low overhead. Creating thousands of CPU threads is less effective as each thread has too little work to amortize the overhead. GPU threads are comparatively cheap, and this allows for many small parallel tasks (like likelihood calculations for a single datapoint). Our compiler achieves this parallelization with no extra information beyond that which is normally encoded in a graphical model description and uses a symbolic representation that allows scaling to large models (particularly for latent variable models like LDA). It also makes it easy to run different inference algorithms and evaluate the tradeoffs between convergence and sampling time. The generated inference code is competitive in terms of model performance with other probabilistic modeling systems, and can sample from large problems much more quickly.

The current version of Augur runs on a single GPU, which introduces another tier into the memory hierarchy as data and samples need to be streamed to and from the GPU’s memory and main memory. We do not currently support this in Augur for problems larger than GPU memory, though it is possible to analyse the generated inference code and automatically generate the data movement code [22]. This movement code can execute concurrently with the sampling process. One area we have not investigated is expanding Augur to clusters of GPUs, though this will introduce the synchronization problems others have encountered when scaling up MCMC [23].

## References

- [1] N. D. Goodman. The principles and practice of probabilistic programming. In *Proc. of the 40th ACM Symp. on Principles of Programming Languages*, POPL ’13, pages 399–402, 2013.
- [2] A. Thomas, D. J. Spiegelhalter, and W. R. Gilks. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian Statistics*, 4:837 – 842, 1992.
- [3] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. of the ACM*, 29(12):1170–1183, 1986.
- [4] G. E. Blelloch. Programming parallel algorithms. *Comm. of the ACM*, 39:85–97, 1996.
- [5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- [6] Stan Dev. Team. *Stan Modeling Language Users Guide and Ref. Manual, Version 2.2*, 2014.
- [7] M. Plummer. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, pages 20–22, 2003.
- [8] M.D. Hoffman and A. Gelman. The No-U-Turn Sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623, 2014.
- [9] G. Marsaglia and W. W. Tsang. A simple method for generating gamma variables. *ACM Trans. Math. Softw.*, 26(3):363–372, 2000.
- [10] D. Venugopal and V. Gogate. Dynamic blocking and collapsing for Gibbs sampling. In *29th Conf. on Uncertainty in Artificial Intelligence*, UAI’13, 2013.
- [11] R. Neal. CSC 2541: Bayesian methods for machine learning, 2013. Lecture 3.
- [12] T. L. Griffiths and M. Steyvers. Finding scientific topics. In *Proc. of the National Academy of Sciences*, volume 101, 2004.
- [13] A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Neural Information Processing Systems 22*, pages 1249–1257, 2009.
- [14] K. Bache and M. Lichman. UCI machine learning repository, 2013.
- [15] M. Hoffman, D. Blei, C. Wang, and J. Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, 2013.
- [16] D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistics in Medicine*, 2009.
- [17] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda. Accelerating inference: Towards a full language, compiler and hardware stack. *CoRR*, abs/1212.2991, 2012.
- [18] T. Minka, J.M. Winn, J.P. Guiver, and D.A. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge.
- [19] V. K. Mansinghka, D. Selsam, and Y. N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014.
- [20] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *24th Conf. on Uncertainty in Artificial Intelligence, UAI 2008*, pages 220–229, 2008.
- [21] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.
- [22] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.
- [23] A. Smola and S. Narayananurthy. An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, 3(1-2):703–710, 2010.

---

# Efficient Training of LDA on a GPU by Mean-for-Mode Estimation

---

**Jean-Baptiste Tristan**  
Oracle Labs, USA

JEAN.BAPTISTE.TRISTAN@ORACLE.COM

**Joseph Tassarotti**  
Department of Computer Science, Carnegie Mellon University, USA

JTASSARO@CS.CMU.EDU

**Guy L. Steele Jr.**  
Oracle Labs, USA

GUY.STEELE@ORACLE.COM

## Abstract

We introduce Mean-for-Mode estimation, a variant of an uncollapsed Gibbs sampler that we use to train LDA on a GPU. The algorithm combines benefits of both uncollapsed and collapsed Gibbs samplers. Like a collapsed Gibbs sampler — and unlike an uncollapsed Gibbs sampler — it has good statistical performance, and can use sampling complexity reduction techniques such as sparsity. Meanwhile, like an uncollapsed Gibbs sampler — and unlike a collapsed Gibbs sampler — it is embarrassingly parallel, and can use approximate counters.

Since collapsed Gibbs sampling is inherently sequential, this earlier work has tried to adapt collapsed sampling to recover some parallelism. However, as the number of cores on GPUs continues to grow, it becomes more and more difficult to extract enough parallelism to scale effectively. In addition, the amount of memory required for collapsed Gibbs sampling and its memory access patterns can limit the amount of data that can be processed at a time. Furthermore, techniques that have been developed to reduce sampling complexity, such as working with sparse matrices (Yao et al., 2009), are difficult to adapt to GPUs.

In contrast, a standard uncollapsed Gibbs sampler is embarrassingly parallel. Tristan et al. (2014) show that a GPU implementation of uncollapsed Gibbs sampling for LDA can scale to a large number of GPU cores. However, Newman et al. (2009) have shown that the statistical performance of an uncollapsed Gibbs sampler is not as good as that of the collapsed Gibbs sampler: it requires more iterations to converge, and it generates estimates that are not as good. Moreover, as pointed out by Smola & Narayananmurthy (2010), the standard uncollapsed sampler cannot use the sparse representations that have been developed for collapsed samplers.

In this paper, we present Mean-for-Mode estimation, a modification of the uncollapsed Gibbs sampler that is related to stochastic expectation maximization as presented by Celeux et al. (1995). In §2 we describe the algorithm and empirically demonstrate that its convergence rate is similar to that of the collapsed Gibbs sampler. Next, we describe a simple GPU implementation in §3 and show that there is more than enough parallelism to scale effectively. One important difference between our work and previous work here is that instead of adapting popular inference methods for LDA that were implicitly designed for a CPU architecture, we rethink the inference method in light of the needs of the GPU architecture.

## 1. Introduction

The performance of GPUs makes them an appealing choice for machine learning. However, there are several challenges in using them efficiently. First, there must be enough parallelism to make use of all the many thousands of GPU cores. Second, GPUs have small amounts of memory, and with enough parallelism, memory bandwidth becomes a bottleneck. Finally, it is often unclear how to translate techniques that have been developed for CPUs into improvements for GPUs.

Topic models, such as LDA (Blei et al., 2003), are a good example of these challenges of using GPUs. Although there are several methods that can be used to train LDA (Asuncion et al., 2009), earlier work targeting the GPU (Yan et al., 2009; Lu et al., 2013) has primarily used collapsed Gibbs sampling (Griffiths & Steyvers, 2004) and adapted techniques from distributed implementations of LDA (Newman et al., 2009).

---

*Proceedings of the 32<sup>nd</sup> International Conference on Machine Learning*, Lille, France, 2015. JMLR: W&CP volume 37. Copyright 2015 by the author(s).

A key benefit of Mean-for-Mode estimation is that *it enables the use of several techniques that are key to scaling to larger datasets and managing memory issues*. First, Mean-for-Mode estimation can use single-precision floating points to store parameter values, and does not need to store the latent topic-assignment variables. Since there are as many latent variables in the model as there are tokens in the corpus, this is a very significant reduction in memory usage. In addition, Mean-for-Mode estimation can use approximate counters (Morris, 1978) to decrease memory use (§4.1). Next, we show that unlike traditional uncollapsed Gibbs samplers, it can also make use of sparsity (§5). In comparison to earlier work that tried to use sparsity with collapsed Gibbs samplers on GPUs, the implementation is straightforward. We believe these techniques are applicable to any large mixture models.

## 2. Mean-for-Mode Estimation for LDA

Before describing Mean-for-Mode estimation, we briefly review the LDA model. LDA is a categorical mixture model for text documents with one set of mixing coefficients per document, with the components shared across the corpus. It is a Bayesian model in which both the mixing coefficients and the components are given a Dirichlet prior. The distribution of LDA is described in Figure 1, and the reader can refer to the introduction written by Blei (2012) for more details about topic modeling and LDA. Note that we reserve the word “latent” only for the topic-assignment variables ( $z_{ij}$ ), not the parameters of the model.

In order to effectively train LDA on the GPU, we need an inference method that is embarrassingly parallel. For example, to make full use of a modern NVIDIA GPU, it is desirable to have an application with tens of thousands of threads, perhaps even millions. Unfortunately, the collapsed Gibbs sampler is a sequential algorithm. Indeed, what makes it a good algorithm is that integrating the parameters of LDA makes the latent variables directly dependent on each other. An obvious way to devise a highly parallel sampler is to not integrate the parameters before deriving the Gibbs sampler, thereby using an uncollapsed Gibbs sampler. In this case, the algorithm will sample not only the latent variables, but also the parameters of the model ( $\phi$  and  $\theta$ ). However, as noted by others (Newman et al., 2009), using such an uncollapsed Gibbs sampler for LDA requires more iterations to converge.

To address this problem, Mean-for-Mode estimation uses a point estimate of the  $\phi$  and  $\theta$  parameters instead of sampling them. The algorithm is shown in Figure 2. First, we draw the parameters from the prior.<sup>1</sup> Inside the main loop that

<sup>1</sup>For this presentation, we use this simple initialization. There are other, possibly better, initializations we could use (Wallach et al., 2009), but this is independent of the key ideas of our algorithm.

generates samples, we first draw all of the latent variables given the parameters, as we would with an uncollapsed Gibbs sampler for LDA. Then, we “simulate” the parameters by assigning to them the mean of their distribution conditioned on all other variables of the model.

Why use the mean? A different choice for a point estimate would be to use the mode of the conditional distribution. However, these conditional distributions are Dirichlet distributions, which do not necessarily have a mode. In contrast, the mean of a Dirichlet distribution is always defined, and in practice it results in good statistical performance. Moreover, note that if  $X \sim \text{dir}(\alpha)$  where  $X$  is a random vector of size  $K$ , then  $\mathbb{E}[X_i] = \frac{\alpha_i}{\sum_k \alpha_i}$ , which is equal to  $\frac{(\alpha_i + 1) - 1}{(\sum_k \alpha_i + 1) - K}$  which is the mode of the Dirichlet distribution  $\text{dir}(\alpha + 1)$ . This means that Mean-for-Mode estimation is an instance of stochastic expectation maximization (Celeux et al., 1995), and consequently it has an equilibrium.

A different way to think about the algorithm is with respect to the collapsed Gibbs sampler. In the collapsed Gibbs sampler, we draw the latent variables with the parameters integrated out:

$$p(z|w) = \int_{\theta} \int_{\phi} p(z|\theta, \phi, w)p(\theta)p(\phi)d\phi d\theta$$

The Mean-for-Mode algorithm corresponds to a plug-in approximation (Murphy, 2012) of the above equation using the estimates  $\mathbb{E}[\theta]$  and  $\mathbb{E}[\phi]$  instead of the MAP.

$$p(z|w) \approx p(z|\mathbb{E}[\theta], \mathbb{E}[\phi], w)$$

Using the posterior mean as opposed to the MAP is common to avoid overfitting.

We do not claim that this algorithm is sophisticated or very original: many inference algorithms are modifications of expectation-maximization or Gibbs sampling that use both stochastic simulation and point-estimation (e.g., Monte-Carlo Expectation maximization, iterated conditional modes, greedy Gibbs sampling (Bishop, 2006)). Rather, our contribution is in noting that this point in the design space is a sweet spot for GPU implementations (and possibly distributed implementations as well): it is embarrassingly parallel, while still allowing the use of space-saving optimizations such as sparsity and approximate counters. Although our focus in this paper is on LDA, these optimizations could be used for other mixture models by carefully using point estimates for the parameters.

We have run a series of experiments which show that in practice, Mean-for-Mode estimation converges in fewer samples than standard uncollapsed Gibbs sampling. In these experiments, we observed how the log-likelihood of LDA evolves with the number of samples. Figure 3 presents the results of one of our experiments, run on a subset of Wikipedia

$$p(\mathbf{w}, \mathbf{z}, \boldsymbol{\theta}, \boldsymbol{\phi}) = \left[ \prod_{i=1}^M \prod_{j=1}^{N_i} \text{cat}(w_{ij} | \phi_{z_{ij}}) \text{cat}(z_{ij} | \theta_i) \right] \left[ \prod_{i=1}^M \text{dir}(\theta_i | \alpha) \right] \left[ \prod_{i=1}^K \text{dir}(\phi_i | \beta) \right]$$

Figure 1. Mixed density for the LDA model.  $M$  is the number of documents,  $N_i$  is the size of document  $i$ ,  $K$  is the number of topics,  $w_{ij}$  is the  $j^{th}$  word in document  $i$ ,  $z_{ij}$  is the topic associated to  $w_{ij}$ ,  $\theta_i$  is the distribution of topics in document  $i$ ,  $\phi_k$  is the distribution of vocabulary words in topic  $k$ . cat and dir refer respectively to the probability mass function of the Categorical distribution and the probability density function of the Dirichlet distribution. The variables  $\boldsymbol{\theta}$  and  $\boldsymbol{\phi}$  can be integrated analytically; doing so leads to the collapsed form of LDA, from which we can derive the effective collapsed Gibbs sampler.

### 1. Initialize

- Sample  $\boldsymbol{\theta}^{(0)} \sim \text{Dir}(\alpha)$
- Sample  $\boldsymbol{\phi}^{(0)} \sim \text{Dir}(\beta)$

### 2. For $\tau = 1, \dots, T$

- Sample  $\mathbf{z}^{(\tau+1)} \sim p(\mathbf{z} | \boldsymbol{\theta}^{(\tau)}, \boldsymbol{\phi}^{(\tau)}, \mathbf{w})$
- Evaluate  $\boldsymbol{\theta}^{(\tau+1)}$  and  $\boldsymbol{\phi}^{(\tau+1)}$  given by

$$\begin{aligned}\boldsymbol{\theta}_i^{(\tau+1)} &= \mathbb{E}[\boldsymbol{\theta}_i | \mathbf{z}^{(\tau+1)}] \\ \boldsymbol{\phi}_i^{(\tau+1)} &= \mathbb{E}[\boldsymbol{\phi}_i | \mathbf{z}^{(\tau+1)}]\end{aligned}$$

Figure 2. Mean-for-Mode estimation for LDA producing  $T + 1$  samples. It is similar to an uncollapsed Gibbs sampler, but the stochastic steps that sample the parameters  $\boldsymbol{\theta}$  and  $\boldsymbol{\phi}$  are replaced with point estimates. If we were to use the mode as a point estimate, this algorithm would be a stochastic expectation maximization. However, because the mode of each distribution of interest may not exist, we use the conditional mean as a point estimate. Note that for simplicity, the equations are presented modulo conditional independence.

(50,000 documents, 3,000,000 tokens, 40,000 vocabulary words) with 20 topics, and both  $\alpha$  and  $\beta$  equal to 0.1. Unless otherwise noted, this is the dataset and configuration we use in the remainder of the paper. The experiment was run 10 times with a varying seed for the random number generator. We drew over 70 samples with each method. Note that we have also run many more experiments, by varying the values of the hyperparameters (36 combinations taken from  $\{0.01, 0.1, 0.25, 0.5, 0.75, 1\}$  for both  $\alpha$  and  $\beta$ ), and by varying the number of documents (from 5,000 to 50,000) and the number of topics (from 20 to 1000), but we present only one here. The result shown here is consistent with our other experiments and the conclusions we draw in the next paragraph.

The graph confirms the experiments presented by Newman et al. (2009). The uncollapsed Gibbs sampler does indeed converge significantly more slowly than the collapsed Gibbs sampler. Also, as they noted, trying to improve the conver-

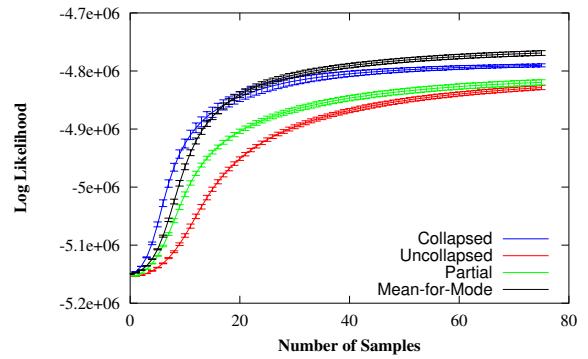


Figure 3. Comparison of the statistical performance of four samplers for LDA: collapsed Gibbs sampler, uncollapsed Gibbs sampler, Gibbs sampler with only the  $\boldsymbol{\theta}$  parameters collapsed, and the Mean-for-Mode estimation. This graph shows the result of 10 runs on a Wikipedia subset (50,000 documents, 3,000,000 tokens, 40,000 vocabulary words) using 20 topics.

gence while keeping a lot of the parallelism by integrating only  $\boldsymbol{\theta}$  does not improve the situation much. However, the Mean-for-Mode estimation seems to work well, with a convergence rate closer to that of the collapsed Gibbs sampler. In this specific experiment its log likelihood exceeds that of the collapsed Gibbs sampler after 20 iterations. Note that we have consistently examined the quality of the topics, and the topics seemed to be of the same quality as the ones of the collapsed Gibbs sampler (and most often, the topics are very similar).

### 3. Parallel GPU Implementation

The Mean-for-Mode estimation applied to LDA exposes a lot of fine-grained parallelism and enables effective GPU implementations. In this section, we describe a basic implementation, which is mostly straightforward. We also show how to refine this with space-saving optimizations such as approximate counters and sparse representations.

We store the data and state of the algorithm using the fol-

**Algorithm 1** Drawing the latent variables

```

1: memclear wpt
2: memclear tpd
3: memclear wt
4: for  $m = 0$  to  $M - 1$  in parallel do
5:   float p[K]
6:   for  $i = 0$  to  $N - 1$  do
7:     float sum = 0;
8:     int c_word = words[m][i]
9:     for  $j = 0$  to  $K - 1$  do
10:      sum += phis[j][c_word] * thetas[m][j]
11:      p[j] = sum
12:   end for
13:   float stop = uniform() * sum;
14:   for  $j = 0$  to  $K - 1$  do
15:     if  $stop < p[j]$  then
16:       break
17:     end if
18:   end for
19:   atomic increment wpt[j][c_word]
20:   atomic increment wt[j]
21:   increment tpd[m][j]
22: end for
23: end for

```

lowing arrays and matrices (where  $\mathbb{I}$  denotes integers and  $\mathbb{F}$  denotes floating-points):  $\text{words} \in \mathbb{I}^{M \times N}$ ;  $\text{phis} \in \mathbb{F}^{K \times V}$ ;  $\text{thetas} \in \mathbb{F}^{M \times K}$ ;  $\text{tpd} \in \mathbb{I}^{M \times K}$  (topics per document);  $\text{wpt} \in \mathbb{I}^{K \times V}$  (words per topic);  $\text{wt} \in \mathbb{I}^K$  (total words per topic).

Algorithm 1 shows how we sample the latent variables. Note that for each document, we launch a separate thread that samples topics for every word in that document. For the corpuses we are interested in, the number of documents ranges from tens of thousands to millions, so there is more than enough parallelism.

As the threads select topics for each token in their documents, they update the  $\text{wpt}$ ,  $\text{wt}$ , and  $\text{tpd}$  matrices. These keep count of how many times each word has been assigned to a particular topic, how many times each topic has been assigned throughout the corpus, and how many times each topic appears in each document. These counts are used to estimate  $\phi$  and  $\theta$ . Algorithm 2 shows how  $\mathbb{E}[\phi]$  is computed by launching a thread for each entry ( $\mathbb{E}[\theta]$  is computed similarly).

We implemented this algorithm on an NVIDIA Titan Black, as well as the uncollapsed Gibbs sampler. We also implemented a collapsed Gibbs sampler for comparison, on an Intel i7-4820K CPU. We present the resulting benchmarks in Figures 4 and 5 to show how the gap between the GPU algorithms' runtimes and that of a collapsed Gibbs sampler

**Algorithm 2** Estimation of the  $\phi$  variables

```

1: for  $v = 0$  to  $V - 1$  in parallel do
2:   for  $k = 0$  to  $K - 1$  in parallel do
3:     phis[k][v] = (wpt[k][v] +  $\beta$ ) / (wt[k] +  $\beta V$ )
4:   end for
5: end for

```

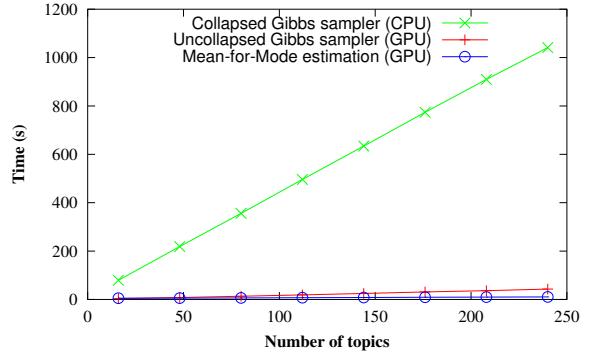


Figure 4. Time to draw 100 samples for 50,000 documents as a function of the number of topics. The uncollapsed Gibbs sampler and the Mean-for-Mode estimation benefit from the large number of parallel units of the GPU when we increase the number of topics. They have similar runtime performance, with the Mean-for-Mode estimation being faster.

scales. Moreover, note that the GPU algorithms scale with increased number of cores. As an example, while switching from an NVIDIA GeForce 640 (384 cores) to a Titan (2688 cores), we observed a  $10\times$  speed-up on the uncollapsed Gibbs sampler.<sup>2</sup> Note that the sequential sampler here is not the most efficient CPU implementation of collapsed Gibbs sampling, such as Fast-LDA (Porteous et al., 2008) or SparseLDA (Yao et al., 2009). However, our point is that an implementation of Mean-for-Mode estimation for GPUs that has not been overly tailored to LDA has good complexity and scales well, and the gap in runtime compared with that of a collapsed Gibbs sampler increases linearly.

## 4. Improving Memory Usage

As we suggested in the introduction, after exposing enough parallelism, the next barriers to performance are related to memory use. Given the degree of parallelism of the algorithm and the hardware, it is best to process as much data as possible on one GPU. In addition to reducing memory footprint, we can improve performance by reducing the amount of memory accesses, thereby improving memory bandwidth

<sup>2</sup>Not all of this performance gap is due to number of cores. The GeForce 640 is based on the earlier Kepler chipset and has different memory clockrate and bandwidth.

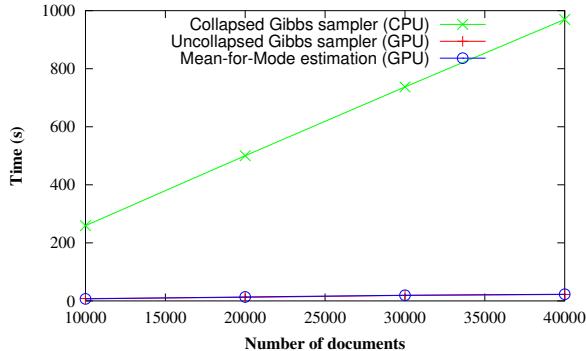


Figure 5. Time to draw 100 samples for 20 topics as a function of the number of documents. The uncollapsed Gibbs sampler and the Mean-for-Mode estimation benefit from the large number of parallel units of the GPU when we increase the number of documents.

usage within the GPU.

Unfortunately, the matrices used in Algorithms 1 and 2 are very large. However, note that even the basic version of Mean-for-Mode estimation described in the previous section has some advantages in these respects over both uncollapsed and collapsed Gibbs samplers.

Unlike the uncollapsed Gibbs sampler, Mean-for-Mode estimation does not sample the parameters from Dirichlet distributions. Dirichlet variates are often generated by sampling from Gamma distributions and then normalizing. The shape parameters of these Gamma distributions are computed by adding the priors to the appropriate count from  $wpt$  or  $tpd$ . If  $\alpha$  or  $\beta$  are small (for instance, 0.01) and the current count from  $wpt$  or  $tpd$  is very small, it is quite likely that the unnormalized Gamma variate is not representable with single-precision floating-points.<sup>3</sup> This means uncollapsed Gibbs samplers must store these samples as double-precision floating-point values to prevent rounding off to 0. In contrast, because it doesn't need to draw from a Gamma distribution, the Mean-for-Mode estimator can store these parameters as single-precision floats, which are smaller and faster to process. Moreover, some GPU architectures, including the ones used in our experiments, have many more single-precision cores than double-precision cores, so using single-precision parameters achieves greater parallelism on such GPUs.

Although the collapsed Gibbs sampler does not have to store these parameters at all, the trade-off is that it must store the latent variables. This requires space on the order of the

<sup>3</sup>It is common to use  $\beta$  as small as 0.01, and the smallest positive single precision float is  $2^{-149}$ . The CDF of the Gamma distribution with shape 0.01 and scale of 1 at  $2^{-149}$  is  $\approx 0.358$ .

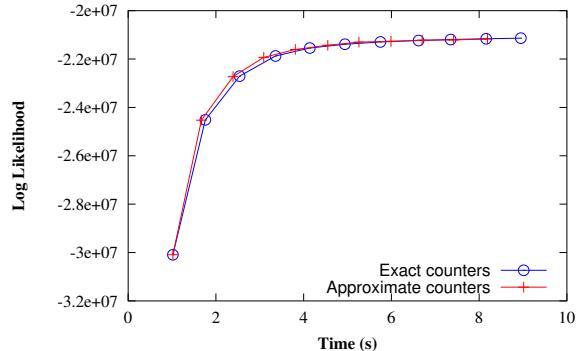


Figure 6. Evolution of log likelihood over time for the Mean-for-Mode estimator with exact counters and approximate counters. The experiment is plotted every 10 iterations for 20 topics and 50,000 documents. The sampler with approximate counters is designed to handle large data loads, but also happens to have a better runtime. This runtime advantage increases with larger number of topics.

total number of tokens in the corpus, and is typically much larger than the parameter matrices when analyzing many documents.

#### 4.1. Approximating the Counters

In Algorithm 1, note that the matrices  $wpt$  and  $tpd$  are cleared to zero at the very beginning of the algorithm, and that the only updates to their entries are increments. (In contrast, with the collapsed Gibbs sampler, the corresponding matrices are never cleared to zero, and updates can involve both increments and decrements.) This feature of the Mean-for-Mode algorithm (and in general of uncollapsed algorithms) makes it possible to use approximate counters for the counts in  $wpt$  and  $tpd$ .

The intuitive idea of approximate counters (Morris, 1978) is to estimate the order of magnitude of the number of increments. As a simple example, assume we want to increment a counter and the current value it stores is  $X$ . We increment  $X$  with probability  $2^{-X}$ , and otherwise do nothing. In the end, a statistically reasonable estimate of the number of increment attempts is  $2^X - 1$ . This idea can be improved in different ways to allow for different level of precision (Morris, 1978) or to have a behavior similar to that of a floating-point representation (Csúrös, 2010).

The benefit of using approximate counters is obvious: we can greatly reduce the amount of memory required to store the counts; for instance, we could decide to use only 8 bits per counter, or even 4. However, this raises two questions. First, can we use approximate counters while preserving the statistical performance? Second, what is the impact on runtime performance? In Figure 6, we show that, for

8-bit counters, the approximation has no consequence on the statistical performance. More surprisingly, perhaps, the approximate counters lead to a gain in runtime performance (in our experience, this gain increases with the number of topics) despite the fact that an increment now requires drawing from a uniform distribution. This is likely because we perform fewer writes to memory when incrementing approximate counters, since each write happens only with some probability. In addition, the reads from memory for a warp only need to load 32 bytes instead of 128.

## 4.2. Coalescing Memory Accesses

Finally, a key point about Algorithm 1 is the memory access to the  $\phi$  matrix in the innermost loop. Consider that several threads, working on distinct documents, are each processing a distinct word. As they all execute this memory access in a SIMD fashion, they access non-contiguous chunks of memory. This is a so-called un-coalesced memory access, and it is a significant bottleneck. To cope with this performance issue, we need to have the threads work together to bring the relevant data from memory to the registers, and indeed we can entirely redesign the algorithm so that the data is shared between the threads. This “butterfly” optimization (Steele & Tristan, 2015) is also applicable to sampling from categorical distributions in other mixture models. The runtime effects of this optimization for LDA with Mean-for-Mode estimation are presented in Figure 7.

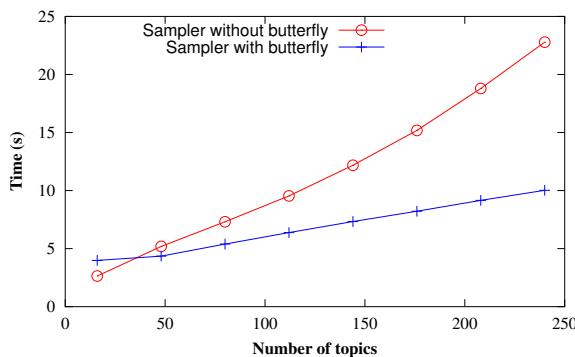


Figure 7. Time to draw 100 samples as a function of topic number for the Mean-for-Mode estimation with the butterfly optimization turned on or off. The butterfly optimization is very effective as the number of topics becomes larger.

## 5. Sparse Representation of Probability Matrices

The count matrices  $wpt$  and  $tpd$  are typically very sparse. It is possible for collapsed Gibbs samplers to take advantage of this sparsity (Porteous et al., 2008; Yao et al., 2009) to

significantly improve runtime performance. However, this sparsity technique is difficult to use with GPUs. As Lu et al. (2013) note, the problem is that as threads draw new values for the latent variables and update the count matrices, some of the entries which were zero will become non-zero. This is fine on CPUs, where we can use sparse matrix representations that can easily be resized dynamically. However, these representations are not efficient on GPUs.

Meanwhile, in an uncollapsed Gibbs sampler, the function that draws latent variables reads from the probability matrices  $\phi$  and  $\theta$ , but updates only the  $wpt$  and  $tpd$ . Since the  $\phi$  and  $\theta$  matrices are static during latent variable assignment, a sparse representation would not need to be dynamically resized. Unfortunately, these matrices are drawn from Dirichlet distributions, so they are not sparse. This is a considerable drawback, and in fact it makes it difficult to justify using an uncollapsed Gibbs sampler, despite the ample parallelism.

Fortunately, Mean-for-Mode estimation overcomes both of these issues. Since the algorithm uses deterministic point estimates for the values of the  $\theta$  and  $\phi$  matrices, these matrices can be stored sparsely. In addition, these matrices are not modified while drawing new values for the latent variables, so there is no need to resize them dynamically. We use a dense representation of  $wpt$  and  $tpd$  so that they can be updated during latent variable assignment. This makes it possible to use sparsity to reduce the sampling complexity, even on GPUs.

The total unnormalized probability mass for the topic assignment of word  $v$  appearing in document  $m$  is proportional to

$$\sum_k \left[ (tpd[m][k] + \alpha) \times \frac{wpt[k][v] + \beta}{wt[k] + \beta V} \right]$$

so, as pointed out by Yao et al. (2009), we can decompose the probability calculation as the sum of three terms (“buckets”)  $S$ ,  $R_m$ , and  $Q_{mv}$  where

$$S = \sum_k \frac{\alpha\beta}{wt[k] + \beta V} \quad R_m = \sum_k \frac{tpd[m][k]\beta}{wt[k] + \beta V}$$

$$Q_{mv} = \sum_k \frac{tpd[m][k] + \alpha}{wt[k] + \beta V} \times wpt[k][v]$$

where  $S$  is the same for every document,  $R_m$  is the same for every word in document  $m$ , and  $Q_{mv}$  depends on both the word and the document. To make use of this rewriting when choosing a topic (cf. Figure 1, lines 13–18) we pre-compute  $S$ , as well as  $R_m$  for each  $m$ . Then, as the thread processing document  $m$  needs to sample a topic for some word  $v$ , it computes  $Q_{mv}$ . It then decides which “bucket” to draw from based on their relative masses. If it selects buckets  $R_m$  or  $Q_{mv}$ , we consider only topics whose counts are non-zero,

---

**Algorithm 3** LDA sampler using both sparse and dense matrices

---

```

1: estimate_phi()
2: estimate_theta()
3: for  $i = 0$  to  $T$  do
4:   float S
5:   if  $i <= D$  then
6:     draw_z()
7:   else
8:     sparse_draw_z(S)
9:   end if
10:  if  $i < D$  or else  $i == T$  then
11:    estimate_phi()
12:    estimate_theta()
13:  else
14:    S = pre_compute()
15:    estimate_phi_sparse()
16:    estimate_theta_sparse()
17:    segmented reduction thetas
18:  end if
19: end for
```

---

which greatly reduces the number of multiplications that must be done.

Algorithm 3 presents the sampler. It corresponds to the high-level Mean-for-Mode algorithm presented in Figure 2. For performance reasons, it is best to start out using dense matrices, then switch to using sparse representation after a few iterations, once the count matrices become sufficiently sparse. The number of initial iterations that are done using dense probability matrices corresponds to the parameter  $D$ .

Once the sampler has reached the point where it uses sparse probability matrices, each iteration works as follows. Starting after the latent variables have been drawn, we first pre-compute (function “pre\_compute”) many useful terms, including  $Z = 1/(\beta V + \text{wt}[k])$ , and  $\alpha Z^{-1}$ ,  $\alpha\beta Z^{-1}$ ,  $\beta Z^{-1}$  for all  $k$ . In the remainder, for simplicity of presentation, we assume that these values are pre-computed and stored. We perform a reduction over the terms  $\alpha\beta Z^{-1}$  to obtain  $S$ .

Then the functions “estimate\_phi\_sparse” and “estimate\_theta\_sparse” take as input the counts and produce sparse matrices, respectively in Compressed Sparse Column representation and Compressed Sparse Row representation. Although we call these matrices `phis` and `thetas`, in analogy with the non-sparse algorithm, they are now really terms that are used in computing the  $R$  and  $Q$  buckets described above:

$$\text{thetas} = m \mapsto \left\{ \left( k, \frac{\text{tpd}[m][k]}{\text{wt}[k] + \beta V} \right) : \text{tpd}[m][k] \neq 0 \right\}$$

$$\text{phis} = v \mapsto \{(k, \text{wpt}[k][v]) : \text{wpt}[k][v] \neq 0\}$$

Note that when we refer to `tpd` and `wpt` above, we mean their values at the time that the `thetas` and `phis` matrices are produced. In the remainder, when using these sparse matrices, we use the following notations (described for `phis` but valid for all three matrices). For a given word  $v$ ,  $\text{phis}[v]$  refers to the set  $\{(k, \text{wpt}[k][v]) : \text{wpt}[k][v] \neq 0\}$ . We write  $k \in \text{phis}[v]$  if there exists a value  $a$  such that  $(k, a) \in \text{phis}[v]$ . If  $k \in \text{phis}[v]$ , we use  $\text{phis}[k][v]$  to refer to the location in the array `phis` as well as the value at that location.

Once `thetas` has been computed, it is simple to obtain the  $R_m$  term for each document by performing a segmented reduction over `thetas` (stored as an array):

$$R_m = \sum_k \text{thetas}[m][k] \cdot \beta$$

The function that draws the topics in the sparse case (function “sparse\_draw\_z”) is different from Algorithm 1. First, we need to decide which bucket to draw from by calculating  $Q_{mv}$ . Then we need to select from topics with non-null probabilities. To calculate  $Q_{mv}$ , the thread processing document  $m$  computes:

$$\left( \frac{\alpha}{\text{wt}[k] + \beta V} + \text{tpd}[m][k] \right) \times \text{wpt}[k][v]$$

for each  $k$  and sums them. Since these terms are zero whenever  $\text{phis}[k][v]$  is, the sum can be computed by iterating through the sparse representation of the `phis`[ $k$ ] row, instead of all possible values of  $k$ .<sup>4</sup>

In Figure 8 we show how different values of  $D$  affect the sampling complexity. The case where  $D = T$  is the standard Mean-for-Mode. Choosing  $D = 0$  (that is, using only sparse matrices) is initially much slower, but eventually, once the matrices become sparse, it becomes much quicker to draw a sample. Finally, by switching from dense to sparse at a good time, it is possible to benefit from both types of learning. For example, in the streaming setting, the dense version could be used to train an initial model quickly, and then the sparse version could be used to quickly update the model as subsequent documents become available (Yao et al., 2009).

## 6. Related Work

Many papers have been written about parallelizing and distributing Gibbs sampling for LDA. Most follow from the work on AD-LDA by Newman et al. (2009), which is based on the collapsed Gibbs sampler of Griffiths & Steyvers

---

<sup>4</sup>A careful reader may note that this computation requires accessing the `tpd` matrix, which could be costly since it is sparse. An effective implementation can implement a filter to quickly test whether `tpd`[ $m$ ][ $k$ ] is in the sparse matrix. One particularly space-efficient implementation could use a Bloom filter (Bloom, 1970).

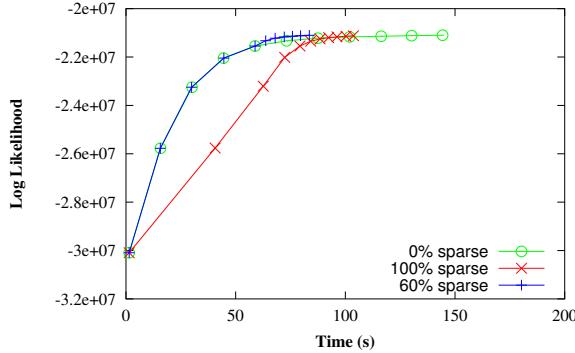


Figure 8. Evolution of log likelihood over time for the Mean-for-Mode estimator with different number of sparse iterations. The experiment is plotted every 10 iterations for 20 topics and 50,000 documents. The version of the sampler that uses sparse representations throughout takes longer to draw initial samples compared to the dense version, but as it begins to converge, subsequent samples are drawn much more quickly. Balancing the use of dense and sparse matrices produces the fastest version overall.

(2004). Distributed implementations include Wang et al. (2009); Smola & Narayananurthy (2010); Liu et al. (2011). Some of these, such as the implementation of Smola & Narayananurthy (2010) are very efficient.

Some authors have also implemented parallel versions of the collapsed Gibbs sampler on GPUs. Yan et al. (2009) provide an implementation based on AD-LDA. Of course, they cannot afford to replicate the matrix of counts  $w_{pt}$  as one must do for a distributed implementation, and to cope with this issue, they make sure that different threads are in charge of different parts of the vocabulary. This unfortunately introduces synchronization rounds that are linear with the number of threads, which will not scale well with more recent GPU architectures and the ever-increasing number of cores.

Lu et al. (2013) provide an implementation also based on AD-LDA, but the counts of  $w_{pt}$  are shared between all of the threads and are accessed concurrently. This implementation also makes use of sparsity, although the implementation is fairly complex. However, the use of a collapsed Gibbs sampler precludes the use of approximate counters which are critical to fit more data on the graphics card. Also, their benchmarks show that scalability is limited, and this is probably due to the much higher number of memory accesses required to keep track of the counts.

The experiments by Tristan et al. (2014) show that an uncollapsed Gibbs sampler is a good candidate for a GPU implementation. However, it suffers from the statistical performance issues described by Newman et al. (2009) and

cannot use the sparsity of the count matrices.

Instead, our algorithm is based on an improved uncollapsed Gibbs sampler and designed in the first place to scale on GPUs. Another example that designs a novel inference method for GPUs is the work of Zhao et al. (2014). Although our algorithm is quite different, we believe it could benefit from the techniques they describe, and vice-versa.

## 7. Conclusion

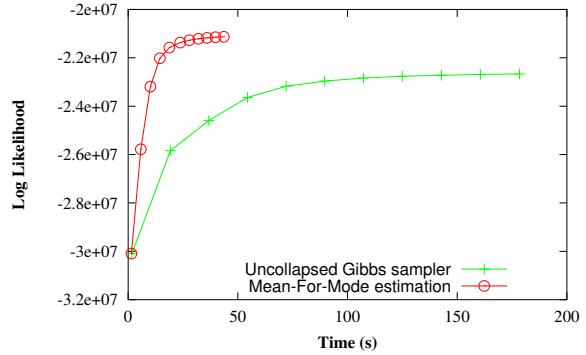


Figure 9. Evolution of log likelihood over time for a highly-optimized uncollapsed Gibbs sampler compared to the Mean-for-Mode estimator presented in this paper. The Mean-for-Mode estimation also has a much smaller memory footprint and can handle more data, which is useful given the scalability characteristics of GPU implementation of training for LDA.

We have shown how Mean-for-Mode estimation, a variant of uncollapsed Gibbs sampling that uses the mean of Dirichlet distributions to make point estimates, can be implemented efficiently on GPUs. The algorithm exposes a lot of parallelism and has good statistical performance. It also lends itself to reducing the total amount of computation by taking advantage of sparsity. The overall gain in performance on our running example is presented in Figure 9, and the gain grows larger with increased number of documents or topics. In addition to exposing sufficient parallelism, Mean-for-Mode estimation also enables the use of techniques to reduce memory footprint and bandwidth use. By combining sparse representations, approximate counters, and avoiding the need to store latent variables, we are able to process larger data sets on a single GPU node.

Although we have focused here on using Mean-for-Mode estimation for LDA, we believe similar techniques could be applied to parameter estimation for other large mixture models. In addition, the memory-saving optimizations we describe here may be applicable in the distributed setting, where minimizing communication and maximizing the amount of data processed per node is important.

## Acknowledgments

The authors thank Adam Pocock and Jeffrey Alexander for installing, configuring, and maintaining the software and hardware that we rely on for our research.

## References

- Asuncion, Arthur, Welling, Max, Smyth, Padhraic, and Teh, Yee Whye. On smoothing and inference for topic models. In *Proc. Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI '09*, pp. 27–34, Arlington, Virginia, USA, 2009. AUAI Press. ISBN 978-0-9749039-5-8. <http://dl.acm.org/citation.cfm?id=1795114.1795118>. See also <http://www.auai.org/uai2009/proceedings.html>.
- Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.
- Blei, David M. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, April 2012. ISSN 0001-0782. doi: 10.1145/2133806.2133826.
- Blei, David M., Ng, Andrew Y., and Jordan, Michael I. Latent Dirichlet allocation. *J. Machine Learning Research*, 3:993–1022, March 2003. ISSN 1532-4435. <http://dl.acm.org/citation.cfm?id=944919.944937>.
- Bloom, Burton H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692.
- Celeux, Gilles, Chauveau, Didier, and Diebolt, Jean. On stochastic versions of the EM algorithm. Technical Report RR-2514, Institut National de Recherche en Informatique et Automatique, 1995. <https://hal.inria.fr/inria-00074164>.
- Csűrös, Miklós. Approximate counting with a floating-point counter. In Thai, M. T. and Sahni, Sartaj (eds.), *Computing and Combinatorics (COCOON 2010)*, number 6196 in Lecture Notes in Computer Science, pp. 358–367. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14030-3. doi: 10.1007/978-3-642-14031-0\_39. See also <http://arxiv.org/pdf/0904.3062.pdf>.
- Griffiths, Thomas L. and Steyvers, Mark. Finding scientific topics. *Proc. National Academy of Sciences of the United States of America*, 101(suppl 1):5228–5235, 2004. doi: 10.1073/pnas.0307752101.
- Liu, Zhiyuan, Zhang, Yuzhou, Chang, Edward Y., and Sun, Maosong. PLDA+: Parallel latent Dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intelligent Systems Technologies*, 2(3):26:1–26:18, May 2011. ISSN 2157-6904. doi: 10.1145/1961189.1961198.
- Lu, Mian, Bai, Ge, Luo, Qiong, Tang, Jie, and Zhao, Jixun. Accelerating topic model training on a single machine. In Ishikawa, Yoshiharu, Li, Jianzhong, Wang, Wei, Zhang, Rui, and Zhang, Wenjie (eds.), *Web Technologies and Applications (APWeb 2013)*, volume 7808 of *Lecture Notes in Computer Science*, pp. 184–195. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37400-5. doi: 10.1007/978-3-642-37401-2\_20.
- Morris, Robert. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978. ISSN 0001-0782. doi: 10.1145/359619.359627.
- Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.
- Newman, David, Asuncion, Arthur, Smyth, Padhraic, and Welling, Max. Distributed algorithms for topic models. *J. Machine Learning Research*, 10:1801–1828, December 2009. ISSN 1532-4435. <http://dl.acm.org/citation.cfm?id=1577069.1755845>.
- Porteous, Ian, Newman, David, Ihler, Alexander, Asuncion, Arthur, Smyth, Padhraic, and Welling, Max. Fast collapsed Gibbs sampling for latent Dirichlet allocation. In *Proc. 14th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining, KDD '08*, pp. 569–577, New York, 2008. ACM. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401960.
- Smola, Alexander and Narayananamurthy, Shravan. An architecture for parallel topic models. *Proc. VLDB Endowment*, 3(1-2):703–710, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920931.
- Steele, Jr, Guy L. and Tristan, Jean-Baptiste. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *ArXiv e-prints*, arXiv:1505.03851 [cs.DC], 2015.
- Tristan, Jean-Baptiste, Huang, Daniel, Tassarotti, Joseph, Pocock, Adam C., Green, Stephen, and Steele, Guy L., Jr. Augur: Data-parallel probabilistic modeling. pp. 2600–2608. Curran Associates, Inc., 2014. <http://papers.nips.cc/book/year-2014>.
- Wallach, Hanna M., Mimno, David, and McCallum, Andrew. Rethinking LDA: Why priors matter. In *Advances in Neural Information Processing Systems 22*, pp. 1973–1981. Curran Associates, Inc., 2009. <http://papers.nips.cc/book/year-2009>.
- Wang, Yi, Bai, Hongjie, Stanton, Matt, Chen, Wen-Yen, and Chang, EdwardY. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In Goldberg, Andrew V. and Zhou, Yunhong (eds.), *Algorithmic Aspects in Information and Management (AAIM 2009)*, volume

5564 of *Lecture Notes in Computer Science*, pp. 301–314. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02157-2. doi: 10.1007/978-3-642-02158-9\_26.

Yan, Feng, Xu, Ningyi, and Qi, Yuan. Parallel inference for latent Dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems* 22, pp. 2134–2142. Curran Associates, Inc., 2009. <http://papers.nips.cc/book/year-2009>.

Yao, Limin, Mimno, David, and McCallum, Andrew. Efficient methods for topic model inference on streaming document collections. In *Proc. 15th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining*, KDD ’09, pp. 937–946, New York, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557121.

Zhao, Huasha, Jiang, Biye, and Canny, John. SAME but different: Fast and high-quality Gibbs parameter estimation. *CoRR*, September 2014. <http://arxiv.org/abs/1409.5402>.

---

# Exponential Stochastic Cellular Automata for Massively Parallel Inference

---

Manzil Zaheer\*  
Carnegie Mellon

Michael Wick  
Oracle Labs

Jean-Baptiste Tristan†  
Oracle Labs

Alex Smola  
Carnegie Mellon

Guy L Steele Jr  
Oracle Labs

## Abstract

We propose an embarrassingly parallel, memory efficient inference algorithm for latent variable models in which the complete data likelihood is in the exponential family. The algorithm is a stochastic cellular automaton and converges to a valid *maximum a posteriori* fixed point. Applied to latent Dirichlet allocation we find that our algorithm is over an order of magnitude faster than the fastest current approaches. A simple C++/MPI implementation on a 20-node Amazon EC2 cluster samples at more than **1 billion tokens per second**. We process 3 billion documents and achieve predictive power competitive with collapsed Gibbs sampling and variational inference.

## 1 Introduction

In the past decade, frameworks such as stochastic gradient descent (SGD) [28] and map-reduce [8] have enabled machine learning algorithms to scale to larger and large datasets. However, these frameworks are not always applicable to Bayesian latent variable models with rich statistical dependencies and intractable gradients. Variational methods [14] and Markov chain Monte-Carlo (MCMC) [10] have thus become the *sine qua non* for inferring the posterior in these models.

Sometimes—due to the concentration of measure phenomenon associated with large sample sizes—computing the full posterior is unnecessary and *maximum a posteriori* (MAP) estimates suffice. It is hence tempting to employ gradient descent. However, for latent variable models such as latent Dirichlet allocation (LDA), calculating gradients involves expensive expectations over rich sets of variables [26].

---

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

MCMC is an appealing alternative, but traditional algorithms such as the Gibbs sampler are inherently sequential and the extent to which they can be parallelized depends heavily upon how the structure of the statistical model interacts with the data. For instance, chromatic sampling [11] is infeasible for LDA, due to its dependence structure. We propose an alternate approach based on stochastic cellular automata (SCA). The automaton is massively parallel like conventional cellular automata, but employs stochastic updates.

Our proposed algorithm, exponential SCA (ESCA), is a specific way of mapping inference in latent variable models with complete data likelihood in the exponential family into an instance of SCA. ESCA has a minimal memory footprint because it stores only the data and the minimal sufficient statistics (by the very definition of sufficient statistics, the footprint cannot be further reduced). In contrast, variational approaches such as stochastic variational inference (SVI) [13] require storing the variational parameters, while MCMC-based methods, such as YahooLDA [30] require storing the latent variables. Thus, ESCA substantially reduces memory costs, enabling larger datasets to fit in memory, and significantly reducing communication costs in distributed environments.

Furthermore, the sufficient statistics dramatically improves efficiency. Typically, in the general case of SCA, updating a cell requires first assembling the values of all the neighboring cells before aggregating them into a local stochastic update. However, in ESCA, the sufficient statistics adequately summarize the states of the neighbors; the computational load is therefore small and perfectly balanced across the cells.

We demonstrate how ESCA is a flexible framework for exponential latent variable models such as LDA. In our experiments, we process over 3 billion documents at a rate of 570 million tokens per second on a small cluster of 4 commodity servers. That said, ESCA is much more general. Table 1 explicitly lists some of the more common modeling choices for which ESCA can

---

\*Work done when an intern at Oracle Labs.

†Corresponding author.

Table 1: Examples of some popular models to which ESCA is applicable.

mix. component/emitter	Bernoulli	Multinomial	Gaussian	Poisson
Categorical	Latent Class Model [9]	Unigram Document Clustering	Mixture of Gaussians [21]	
Dirichlet Mixture	Grade of Membership Model [35]	Latent Dirichlet Allocation [2]	Gaussian-LDA [6]	GaP Model [3]

be easily employed. Our algorithm implicitly simulates stochastic expectation maximization (SEM), and is thus provably correct in the sense that it converges in distribution to a stationary point of the *posterior*.

## 2 Exponential SCA

Stochastic cellular automata (SCA), also known as probabilistic cellular automata, or locally-interacting Markov chains, are a stochastic version of a discrete-time, discrete-space dynamical system in which a noisy local update rule is homogeneously and synchronously applied to every site of a discrete space. They have been studied in statistical physics, mathematics, and computer science, and some progress has been made toward understanding their ergodicity and equilibrium properties. A recent survey [18] is an excellent introduction to the subject, and a dissertation [17] contains a comprehensive and precise presentation of SCA.

The automaton, as a (stochastic) discrete-time, discrete-space dynamical system, is given by an evolution function  $\Phi : \mathcal{S} \rightarrow \mathcal{S}$  over the state space  $\mathcal{S} = \mathcal{Z} \rightarrow C$  which is a mapping from the space of cell identifiers  $Z$  to cell values  $C$ . The global evolution function applies a local function  $\phi_{\mathfrak{z}}(c_1, c_2, \dots, c_r) \mapsto c$  s.t.  $c_i = s(\mathfrak{z}_i)$  to every cell  $\mathfrak{z} \in \mathcal{Z}$ . That is,  $\phi$  examines the values of each of the neighbors of cell  $\mathfrak{z}$  and then stochastically computes a new value  $c$ . The dynamics begin with a state  $s_0 \in \mathcal{S}$  that can be configured using the data or a heuristic.

Exponential SCA (ESCA) is based on SCA but achieves better computational efficiency by exploiting the structure of the sufficient statistics for latent variable models in which the complete data likelihood is in the exponential family. Most importantly, the local update function  $\phi$  for each cell depends only upon the sufficient statistics and thus does *not* scale linearly with the number of neighbors.

### 2.1 Latent Variable Exponential Family

Latent variable models are useful when reasoning about partially observed data such as collections of text or images in which each *i.i.d.* data point is a document or image. Since the same local model is applied to each data point, they have the following form

$$p(\mathbf{z}, \mathbf{x}, \eta) = p(\eta) \prod_i p(z_i, x_i | \eta). \quad (1)$$

Our goal is to obtain a MAP estimate for the parameters  $\eta$  that explain the data  $\mathbf{x}$  through the latent variables  $\mathbf{z}$ . To expose maximum parallelism, we want each cell in the automaton to correspond to a data point and its latent variable. However, in general all latent variables depend on each other via the global parameters  $\eta$ , and thus the local evolution function  $\phi$  would have to examine the values of every cell in the automaton.

Fortunately, if we further suppose that the complete data likelihood is in the exponential family, i.e.,

$$p(z_i, x_i | \eta) = \exp(\langle T(z_i, x_i), \eta \rangle - g(\eta)) \quad (2)$$

then the complete and sufficient statistics are given by

$$T(\mathbf{z}, \mathbf{x}) = \sum_i T(z_i, x_i) \quad (3)$$

and we can thus express any estimator of interest as a function of just  $T(\mathbf{z}, \mathbf{x})$ . Further, when employing expectation maximization (EM), the M-step is possible in closed form for many members of the exponential family. This allows us to reformulate the cell level updates to depend only upon the sufficient statistics instead of the neighboring cells. The idea is that, unlike SCA (or MCMC in general) which produces a sequence of states that correspond to complete variable assignments  $s^0, s^1, \dots$  via a transition kernel  $q(s^{t+1} | s^t)$ , ESCA produces a sequence of sufficient statistics  $T^0, T^1, \dots$  directly via an evolution function  $\Phi(T^t) \mapsto T^{t+1}$ .

### 2.2 Stochastic EM

Before we present ESCA, we must first describe stochastic EM (SEM). Suppose we want the MAP estimate for  $\eta$  and employ a traditional expectation maximization (EM) approach:

$$\max_{\eta} p(\mathbf{x}, \eta) = \max_{\eta} \int p(\mathbf{z}, \mathbf{x}, \eta) \mu(d\mathbf{z})$$

EM finds a mode of  $p(\mathbf{x}, \eta)$  by iterating two steps:

**E-step** Compute in parallel  $p(z_i | x_i, \eta^{(t)})$ .

**M-step** Find  $\eta^{(t+1)}$  that maximizes the expected value of the log-likelihood with respect to the conditional probability, i.e.

$$\begin{aligned} \eta^{(t+1)} &= \arg \max_{\eta} \mathbb{E}_{\mathbf{z}|\mathbf{x}, \eta^{(t)}} [\log p(\mathbf{z}, \mathbf{x}, \eta)] \\ &= \xi^{-1} \left( \frac{1}{n + n_0} \sum_i \mathbb{E}_{\mathbf{z}|\mathbf{x}, \eta^{(t)}} [T(z_i, x_i)] + T_0 \right) \end{aligned}$$

where  $\xi(\eta) = \nabla g(\eta)$  is invertible as  $\nabla^2 g(\theta) \succ 0$  and  $n_0, T_0$  parametrize the conjugate prior.<sup>1</sup>

Although EM exposes substantial parallelism, it is difficult to scale, since the dense structure  $p(z_i|x_i, \eta^{(t)})$  defines values for all possible outcomes for  $z$  and thus puts tremendous pressure on memory bandwidth.

To overcome this we introduce sparsity by employing stochastic EM (SEM) [4]. SEM substitutes the E-step for an S-step that replaces the full distribution with a single sample:

**S-step** Sample  $z_i^{(t)} \sim p(z_i|x_i; \eta^{(t)})$  in parallel.

Subsequently, we perform the M-step using the imputed data instead of the expectation. This simple modification overcomes the computational drawbacks of EM for cases in which sampling from  $p(z_i|x_i; \eta^{(t)})$  is feasible. We can now employ fast samplers, such as the alias method, exploit sparsity, reduce CPU-RAM bandwidth while still maintaining massive parallelism.

The S-step has other important consequences. Notice that the M-step is now a simple function of current sufficient statistics. This implies that the conditional distribution for the next S-step is expressible in terms of the complete sufficient statistics

$$p(z_i|x_i; \eta^{(t)}) = f(z_i, T(\mathbf{x}, \mathbf{z}^{(t)})).$$

Thus each S-step depends only upon the sufficient statistics generated by the previous step. Therefore, we can operate directly on sufficient statistics without the need to assign or store latent variables/states. Moreover it opens up avenues for distributed and parallel implementations that execute on an SCA.

### 2.3 ESCA for Latent Variable Models

SEM produces an alternating sequence of S and M steps in which the M step produces the parameters necessary for the next S step. Since we can compute these parameters on the fly there is no need for an explicit M step. Instead, ESCA produces a sequence consisting only of S steps. We require the exponential family to ensure that these steps are both efficient and compact. We now present ESCA more formally.

Define an SCA over the state space  $\mathcal{S}$  of the form:

$$\mathcal{S} = \mathcal{Z} \longrightarrow \mathcal{K} \times \mathcal{X} \quad (4)$$

where  $\mathcal{Z}$  is the set of cell identifiers (e.g., one per token in a text corpus),  $\mathcal{K}$  is the domain of latent variables, and  $\mathcal{X}$  is the domain of the observed data.

The initial state  $s_0$  is the map defined as follows: for every data point, we associate a cell  $z$  to the pair

<sup>1</sup>The inversion may not be always available in closed form, in which case we resort to numerical techniques.

$(k_z, x)$  where  $k_z$  is chosen at random from  $\mathcal{K}$  and independently from  $k_{z'}$  for all  $z' \neq z$ . This gives us the initial state  $s_0$ .

$$s_0 = z \mapsto (k_z, x) \quad (5)$$

We now need to describe the evolution function  $\Phi$ . First, assuming that we have a state  $s$  and a cell  $z$ , we define the following distribution:

$$p_z(k|s) = f(z, T(s)) \quad (6)$$

Assuming that  $s(z) = (k, x)$  and that  $k'$  is a sample from  $p_z$  (hence the name “stochastic” cellular automaton) we define the local update function as:

$$\phi(s, z) = (k', x) \quad (7)$$

$$\text{where } s(z) = (k, x) \text{ and } k' \sim p_z(\cdot | s)$$

That is, the observed data remain unchanged, but we choose a new latent variable according to the distribution  $p_z$  induced by the state. We obtain the evolution function of the stochastic cellular automaton by applying the function  $\phi$  uniformly on every cell.

$$\Phi(s) = z \mapsto \phi(s, z) \quad (8)$$

Finally, the SCA algorithm simulates the evolution function  $\Phi$  starting with  $s_0$ .

As explained earlier, due to our assumption of complete data likelihood belonging to the exponential family, we never have to represent the states explicitly, and instead employ the sufficient statistics.

An implementation can, for example, have two copies of the data structure containing sufficient statistics  $T^{(0)}$  and  $T^{(1)}$ . We do not compute the values  $T(\mathbf{z}, \mathbf{x})$  but keep track of the sum as we impute values to the cells/latent variables. During iteration  $2t$  of the evolution function, we apply  $\Phi$  by reading from  $T^{(0)}$  and incrementing  $T^{(1)}$  as we sample the latent variables (See Figure 1). Then in the next iteration  $2t+1$  we reverse the roles of the data structures, i.e. read from  $T^{(1)}$  and increment  $T^{(0)}$ . We summarize in Algorithm 1.

---

#### Algorithm 1 ESCA

---

```

1: Randomly initialize each cell
2: for  $t = 0 \rightarrow \text{num iterations}$  do
3:   for all cell  $z$  independently in parallel do
4:     Read sufficient statistics from  $T^{(t \bmod 2)}$ 
5:     Compute stochastic updates using  $p_z(k|s)$ 
6:     Write sufficient statistics to  $T^{(t+1 \bmod 2)}$ 
7:   end for
8: end for

```

---

Use of such read/write buffers offer a virtually lock-free (assuming atomic increments) implementation scheme for ESCA and is analogous to double-buffering in computer graphics. Although there is a synchronization barrier after each round, its effect is mitigated because each cell’s work depends only upon the sufficient statistics and thus does the same amount of work. There-

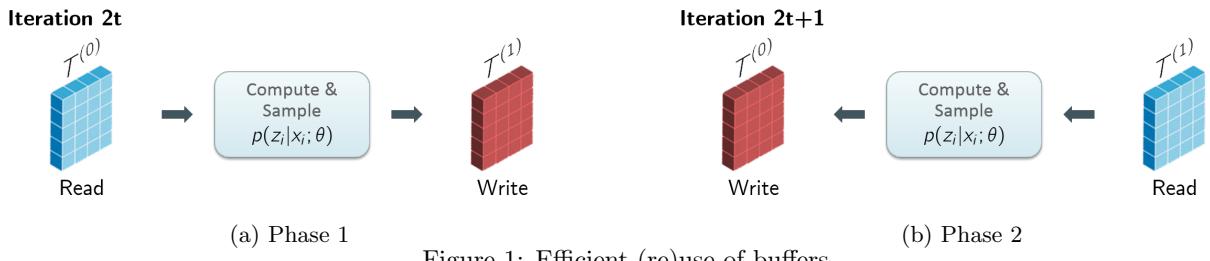


Figure 1: Efficient (re)use of buffers

fore, evenly balancing the work load across computation nodes is trivial, even for a heterogeneous cluster.

Furthermore, in the case of discrete latent variable, updating sufficient statics only requires increments to the data structure  $T^{(r)}$  allowing the use of approximate counters [20, 5]. Approximate counters greatly reduce memory costs for the counts: e.g., only 4 or 8 bits per counter. Recent empirical evidence demonstrates that approximate counters preserve statistical performance without compromising runtime performance [32]. In fact, speed often increases because not every increment to the counter results in a write to memory. Note, due to the compression, maintaining two buffers requires less memory than one. Finally, if the latent variables are discrete valued then we can leverage the fast Vose' alias method [34] to sample. The  $O(|\mathcal{K}|)$  construction cost for the alias method can be easily amortized because the rule is homogeneous and thus alias tables can be shared. Details about alias sampling method is provided in Appendix E.

## 2.4 Wide Applicability of ESCA

As stated previously, ESCA is technically applicable to any model in which the complete data likelihood is in the exponential family. Designing an ESCA algorithm for a model of interest requires simply deriving the S-step for the local update function in the automaton. The S-step is the full conditional (Equation 6) which is easy to derive for many models; for example, mixture models in which (1) the data and parameter components are conjugate and (2) the latent variables and priors are conjugate. We list a few examples of such models in Table 1 and provide additional details in Appendix F. Of course, the extent to which the models enable exploitation of sparsity varies.

ESCA is also applicable to models such as restricted Boltzmann machines (RBMs). For example, if the data were a collection of images, each cell could independently compute the S-step for its respective image. For RBMs the cell would flip a biased coin for each latent variable, and for deep Boltzmann machines, the cells could perform Gibbs sampling. We save a precise derivation and empirical evaluation for future work.

## 2.5 Understanding the limitations of ESCA

While ESCA has tremendous potential as a computational model for machine learning, in some cases, using it to obtain MAP estimates is not clear.

Consider an Ising model on an  $d$ -dimensional torus  $\mathbb{H}$

$$p(x) \propto \prod_{\langle i,j \rangle \in \mathbb{H}} \exp(w_{ij}x_i x_j) \quad (9)$$

in which  $x_i$  takes on values in  $\{-1, 1\}$ . The equilibrium distribution of SCA with a Gibbs update is then [23]

$$q(x) \propto \prod_{\langle i,j \rangle \in \mathbb{H}} \cosh(w_{ij} x_i x_j). \quad (10)$$

Note that the hyperbolic cosine function ( $\cosh$ ) is symmetric in the sense that  $\cosh(r) = \cosh(-r)$ . For values  $r \geq 0$   $\cosh$  is a good approximation and has a maximum that corresponds to the exponential function; however, for values  $r < 0$ , the  $\cosh$  is a poor approximation for the exponential function.

Let  $x_1, x_2$  be two random variables taking on values in  $\{-1, 1\}$ . We define a simple two-variable Ising model on a trivial one-dimensional torus:

$$p(x_1, x_2) \propto \exp(x_1 x_2) \quad (11)$$

We can enumerate and quantify the state space under both SCA  $q(x_1, x_2)$  and the true distribution  $p(x_1, x_2)$ :

state	$x_1$	$x_2$	$x_1 * x_2$	$q(x_1, x_2) \propto$	$p(x_1, x_2) \propto$
0	-1	-1	1	$\cosh(1)$	$\exp(1)$
1	-1	1	-1	$\cosh(-1)$	$\exp(-1)$
2	1	-1	-1	$\cosh(-1)$	$\exp(-1)$
3	1	1	1	$\cosh(1)$	$\exp(1)$

Since  $\cosh$  is symmetric, all states are equally probable for SCA and states 1 and 2 are MAP states. Yet, under the true distribution, they are not. Consequently, SCA with a Gibbs rule for the local evolution function can yield incorrect MAP estimates.

Fortunately, in most cases we are interested in a model over a dataset in which the data is *i.i.d.* That is, we can fix our example as follows. Rather than parallelizing a single Ising model at the granularity of pixels (over a single torus or grid), we instead parallelize the Ising model at the granularity of the data (over multiple tori, one for each image). Then, we could employ Gibbs sampling on each image for the S-step.

## 2.6 Convergence

We now address the critical question of how the invariant measure of ESCA for the model presented in Section 2.1 is related to the true MAP estimates. First, note that SCA is ergodic [17], a result that immediately applies if we ignore the deterministic components of our automata (corresponding to the observations). Now that we have established ergodicity, we next study the properties of the stationary distribution and find that the modes correspond to MAP estimates.

We make a few mild assumptions about the model:

- The observed data Fisher information is non-singular, i.e.  $I(\eta) \succ 0$ .
- For the Fisher information for  $\mathbf{z}|\mathbf{x}$ , we need it to be non-singular and central limit theorem, law of large number to hold, i.e.  $\mathbb{E}_{\eta_0}[I_X(\eta_0)] \succ 0$  and

$$\sup_{\eta} \left| \frac{1}{n} \sum_{i=1}^n I_{x_i}(\eta) - \mathbb{E}_{\eta_0}[I_X(\eta)] \right| \rightarrow 0 \text{ as } n \rightarrow \infty$$

- We assume that  $\frac{1}{n} \sum_{i=1}^n \nabla_{\eta} \log p(x_i; \eta) = 0$  has at least one solution, let  $\hat{\eta}$  be a solution.

These assumptions are reasonable. For example in case of mixture models (or topic models), it just means all component must be exhibited at least once and all components are unique. The details of this case are worked out in Appendix D. Also when the number of parameters grow with the data, e.g., for topic models, the second assumption still holds. In this case, we resort to corresponding result from high dimensional statistics by replacing the law of large numbers with Donsker's theorem and everything else falls into place.

Consequently, we show ESCA converges weakly to a distribution with mean equal to some root of the score function ( $\nabla_{\eta} \log p(x_i; \eta)$ ) and thus a MAP fixed point by borrowing the results known for SEM [25]. In particular, we have:

**Theorem 1** *Let the assumptions stated above hold and  $\tilde{\eta}$  be the estimate from ESCA. Then as the number of i.i.d. data point goes to infinity, i.e.  $n \rightarrow \infty$ , we have*

$$\sqrt{n}(\tilde{\eta} - \hat{\eta}) \xrightarrow{\mathcal{D}} \mathcal{N}(0, I(\eta_0)^{-1}[I - F(\eta_0)^{-1}] \quad (12)$$

where  $F(\eta_0) = I + \mathbb{E}_{\eta_0}[I_X(\eta_0)](I(\eta_0) + \mathbb{E}_{\eta_0}[I_X(\eta_0)])$ .

This result implies that SEM flocks around a stationary point under very reasonable assumptions and tremendous computational benefits. Also, for such complicated models, reaching a stationary point is the best that most methods achieve anyway. Now we switch gears to adopt ESCA for LDA and perform some simple experimental evaluations.

## 3 ESCA for LDA

Topic modeling, and latent Dirichlet allocation (LDA) [2] in particular, have become a must-have of analytics platforms and consequently needs to scale to larger and larger datasets. In LDA, we model each document  $m$  of a corpus of  $M$  documents as a distribution  $\theta_m$  that represents a mixture of topics. There are  $K$  such topics, and we model each topic  $k$  as a distribution  $\phi_k$  over the vocabulary of words that appear in our corpus. Each document  $m$  contains  $N_m$  words  $w_{mn}$  from a vocabulary of size  $V$ , and we associate a latent variable  $z_{mn}$  to each of the words. The latent variables can take one of  $K$  values indicating the topic for the word. Both distributions  $\theta_m$  and  $\phi_k$  have a Dirichlet prior, parameterized respectively with a constant  $\alpha$  and  $\beta$ . See Appendix B for more details.

### 3.1 Existing systems

Many of the scalable systems for topic modeling are based on one of two core inference methods: the collapsed Gibbs sampler (CGS) [12], and variational inference (VI) [2] and approximations thereof [1]. To scale LDA to large datasets, or for efficiency reasons, we may need to distribute and parallelize them. Both algorithms can be further approximated to meet such implementation requirements.

**Collapsed Gibbs Sampling** In collapsed Gibbs sampling the full conditional distribution of the latent topic indicators given all the others is

$$p(z_{mn} = k | \mathbf{z}^{-mn}, \mathbf{w}) \propto \left[ (D_{mk} + \alpha) \frac{W_{kw_{mn}} + \beta}{T_k + \beta V} \right] \quad (13)$$

where  $D_{mk}$  is the number of latent variables in document  $m$  that equal  $k$ ,  $W_{kv}$  is the number of latent variables equal to  $k$  and whose corresponding word equals  $v$ , and  $T_k$  is the number of latent variables that equal  $k$ , all excluding current  $z_{mn}$ .

CGS is a sequential algorithm in which we draw latent variables in turn, and repeat the process for several iterations. The algorithm performs well statistically, and has further benefited from breakthroughs that lead to a reduction of the sampling complexity [37, 16]. This algorithm can be approximated to enable distribution and parallelism, primarily in two ways. One is to partition the data, perform one sampling pass and then assimilate the sampler states, thus yielding an approximate distributed version of CGS (AD-LDA) [24]. Another way is to partition the data and allow each sampler to communicate with a distributed central storage continuously. Here, each sampler sends the differential to the global state-keeper and receives from it the latest global value. A scalable

system built on this principle and leveraging inherent sparsity of LDA is YahooLDA [30]. Further improvement and sampling using alias table was incorporated in lightLDA [39]. Contemporaneously, a nomadic distribution scheme and sampling using Fenwick tree was proposed in F+LDA [38].

**Variational Inference** In variational inference (VI), we seek to optimize the parameters of an approximate distribution that assumes independence of the latent variables to find a member of the family that is close to the true posterior. Typically, for LDA, document-topic proportions and topic indicators are latent variables and topics are parameter. Then, coordinate ascent alternates between them.

One way to scale VI is stochastic variational inference (SVI) which employs SGD by repeatedly updating the topics via randomly chosen document subsets [13]. Adding a Gibbs step to SVI introduces sparsity for additional efficiency [19]. In some ways this is analogous to our S-step, but in the context of variational inference, the conditional is much more expensive to compute, requiring several rounds of sampling.

Another approach, CVB0, achieves scalability by approximating the collapsed posterior [31]. Here, they minimize the free energy of the approximate distribution for a given parameter  $\gamma_{mnk}$  and then use the zero-order Taylor expansion [1].

$$\gamma_{mnk} \propto \boxed{(D_{mk} + \alpha) \times \frac{W_{kw_{mn}} + \beta}{T_k + \beta V}} \quad (14)$$

where  $D_{mk}$  is the fractional contribution of latent variables in document  $m$  for topic  $k$ ,  $W_{kv}$  is the contribution of latent variables for topic  $k$  and whose corresponding word equals  $v$ , and  $T_k$  is the the contribution of latent variables for topic  $k$ . Inference updates the variational parameters until convergence. It is possible to distribute and parallelize CVB0 over tokens [1]. VI and CVB0 are the core algorithms behind several scalable topic modeling systems including Mr.LDA [40] and the Apache Spark machine-learning suite.

**Remark** It is worth noticing that Gibbs sampling and variational inference, despite being justified very differently, have at their core the very same formulas (shown in a box in formula (13) and (14)). Each of which are literally deciding how important is some topic  $k$  to the word  $v$  appearing in document  $m$  by asking the questions: “How many times does topic  $k$  occur in document  $m$ ?” , “How many times is word  $v$  associated with topic  $k$ ?” , and “How prominent is topic  $k$  overall?” . It is reassuring that behind all the beautiful mathematics, something simple and intuitive is happening. As we see next, ESCA addresses the same questions via analogous formulas for SEM.

### 3.2 An ESCA Algorithm for LDA

To re-iterate, the point of using such a method for LDA is that the parallel update dynamics of the ESCA gives us an algorithm that is simple to parallelize, distribute and scale. In the next section, we will evaluate how it works in practice. For now, let us explain how we design our SCA to analyze data.

We begin by writing the stochastic EM steps for LDA (derivation is in Appendix B):

**E-step:** independently in parallel compute the conditional distribution locally:

$$q_{mnk} = \frac{\theta_{mk}\phi_{kw_{mn}}}{\sum_{k'=1}^K \theta_{mk'}\phi_{k'w_{mn}}} \quad (15)$$

**S-step:** independently in parallel draw  $z_{ij}$  from the categorical distribution:

$$z_{mn} \sim \text{Categorical}(q_{mn1}, \dots, q_{mNK}) \quad (16)$$

**M-step:** independently in parallel compute the new parameter estimates:

$$\begin{aligned} \theta_{mk} &= \frac{D_{mk} + \alpha - 1}{N_m + K\alpha - K} \\ \phi_{kv} &= \frac{W_{kv} + \beta - 1}{T_k + V\beta - V} \end{aligned} \quad (17)$$

We simulate these inference steps in ESCA, which is a dynamical system with evolution function  $\Phi : \mathcal{S} \rightarrow \mathcal{S}$  over the state space  $\mathcal{S}$ . For LDA, the state space  $\mathcal{S}$  is

$$\mathcal{S} = \mathcal{Z} \rightarrow \mathcal{K} \times \mathcal{M} \times \mathcal{V} \quad (18)$$

where  $\mathcal{Z}$  is the set of cell identifiers (one per token in our corpus),  $\mathcal{K}$  is a set of  $K$  topics,  $\mathcal{M}$  is a set of  $M$  document identifiers, and  $\mathcal{V}$  is a set of  $V$  identifiers for the vocabulary words.

The initial state  $s_0$  is the map defined as follows: for every occurrence of the word  $v$  in document  $m$ , we associate a cell  $z$  to the triple  $(k_z, m, v)$  where  $k_z$  is chosen uniformly at random from  $\mathcal{K}$  and independently from  $k_{z'}$  for all  $z' \neq z$ . This gives us

$$s_0 = z \mapsto (k_z, m, v) \quad (19)$$

We now need to describe the evolution function  $\Phi$ . First, assuming that we have a state  $s$  and a cell  $z$ , we define the following distribution:

$$p_z(k|s) \propto \boxed{(D_{mk} + \alpha) \times \frac{W_{kv} + \beta}{T_k + \beta V}} \quad (20)$$

where  $D_{mk} = |\{z \mid \exists v. s(z) = (k, m, v)\}|$ ,

$W_{kv} = |\{z \mid \exists m. s(z) = (k, m, v)\}|$ , and

$T_k = |\{z \mid \exists m. \exists v. s(z) = (k, m, v)\}|$ . Note that we have chosen our local update rule slightly different without an offset of  $-1$  for the counts corresponding to

the mode of the Dirichlet distributions and requiring  $\alpha, \beta > 1$ . Instead, our local update rule allows us to have the relaxed requirement  $\alpha, \beta > 0$  which is more common for LDA inference algorithms.

Assuming that  $s(z) = (k, m, v)$  and that  $k'$  is a sample from  $p_z$  (hence the name “stochastic” cellular automaton) we define the local update function as:

$$\phi(s, z) = (k', m, v) \quad (21)$$

where  $s(z) = (k, m, v)$  and  $k' \sim p_z(\cdot | s)$

That is, the document and word of the cell remain unchanged, but we choose a new topic according to the distribution  $p_z$  induced by the state. We obtain the evolution function of the stochastic cellular automaton by applying the function  $\phi$  uniformly on every cell.

$$\Phi(s) = z \mapsto \phi(s, z) \quad (22)$$

Finally, the SCA algorithm simulates the evolution function  $\Phi$  starting with  $s_0$ . Of course, since LDA’s complete data likelihood is in the exponential family, we never have to represent the states explicitly, and instead employ the sufficient statistics.

Our implementation has two copies of the count matrices  $D^i$ ,  $W^i$ , and  $T^i$  for  $i = 0$  or  $1$  (as in CGS or CVB0, we do not compute the values  $D_{ik}$ ,  $W_{kv}$ , and  $T_k$  but keep track of the counts as we assign topics to the cells/latent variables). During iteration  $i$  of the evolution function, we apply  $\Phi$  by reading  $D^{i \bmod 2}$ ,  $W^{i \bmod 2}$ , and  $T^{i \bmod 2}$  and incrementing  $D^{(i+1) \bmod 2}$ ,  $W^{(i+1) \bmod 2}$ , and  $T^{(i+1) \bmod 2}$  as we assign topics.

### 3.3 Advantages of ESCA for LDA

The positive consequences of ESCA as a choice for inference on LDA are many:

- Our memory footprint is minimal since we only store the data and sufficient statistics. In contrast to MCMC methods, we do not store the assignments to latent variables  $\mathbf{z}$ . In contrast to variational methods, we do not store the variational parameters  $\gamma$ . Further, variational methods require  $K$  memory accesses (one for each topic) per word. In contrast, the S-step ensures we only have a single access (for the sampled topic) per word. Such reduced pressure on the memory bandwidth can improve performance significantly for highly parallel applications.
- We can further reduce the memory footprint by compressing the sufficient statistics with approximate counters [20, 5]. This is possible because updating the sufficient statistics only requires increments as in Mean-for-Mode [32]. In contrast, CGS decrements counts, preventing the use of approximate counters.
- Our implementation is lock-free (in that it does not use locks, but assumes atomic increments) because the double buffering ensures we never read or write

to the same data structures. There is less synchronization, which at scale is significant.

- Finally, our algorithm is able to fully benefit from Vose’s alias method [34] because homogeneous update rule for SCA ensures that the cost for constructing the alias table is amortized across the cells. To elaborate, the SCA update Equation (20) decomposes as

$$p_z(k|s) \propto \left[ D_{mk} \frac{W_{kv} + \beta}{T_k + \beta V} \right] + \left[ \alpha \frac{W_{kv} + \beta}{T_k + \beta V} \right] \quad (23)$$

allowing us to treat it as a discrete mixture and divide the sampling procedure into a two steps. First, we toss a biased coin to decide which term of the equation to sample, and second, we employ a specialized sampler depending on the chosen term. The first term is extremely sparse (documents comprise only a small handful of topics) and a basic sampling procedure suffices. The second term is not sparse, but is independent of the current document  $m$  and depends only on the  $W$  and  $T$  matrices. Moreover, as mentioned earlier, during iteration  $i$ , we will be only reading values from non-changing  $W^{i \bmod 2}$ , and  $T^{i \bmod 2}$  matrices. As a result, at the start of each iteration we can precompute, from the  $W$  and  $T$  matrices, tables for use with Vose’s alias method, which enables sampling from the second term in a mere 3 CPU operations. Thus, the evolution for ESCA is extremely efficient.

#### 3.3.1 Connection to SGD

We can view ESCA as implicit SGD on MAP for LDA. This connection alludes to the convergence rate of ESCA. To illustrate, we consider  $\theta$  only. As pointed out in [36, 29], one EM step is:

$$\theta_m^+ = \theta_m + M \frac{\partial \log p}{\partial \theta_{mk}}$$

which is gradient descent with a Frank-Wolfe update and line search. Similarly, for ESCA using stochastic EM, one step is

$$\theta_{mk}^+ = \frac{Dn_{mk}}{N_m} = \frac{1}{N_m} \sum_{n=1}^{N_m} \delta(z_{mn} = k)$$

Again vectorizing and re-writing as earlier:

$$\theta_m^+ = \theta_m + Mg$$

where  $M = \frac{1}{N_m} [\text{diag}(\theta_m) - \theta_m \theta_m^T]$  and  $g = \frac{1}{\theta_{mk}} \sum_{n=1}^{N_m} \delta(z_{mn} = k)$ . The vector  $g$  can be shown to be an unbiased noisy estimate of the gradient, i.e.

$$\mathbb{E}[g] = \frac{1}{\theta_{mk}} \sum_{n=1}^{N_i} \mathbb{E}[\delta(z_{ij} = k)] = \frac{\partial \log p}{\partial \theta_{mk}}$$

Thus, a single step of SEM on our SCA is equivalent to a single step of SGD. Consequently, we could further embrace the connection to SGD and use a subset of the

## ESCA for Massively Parallel Inference

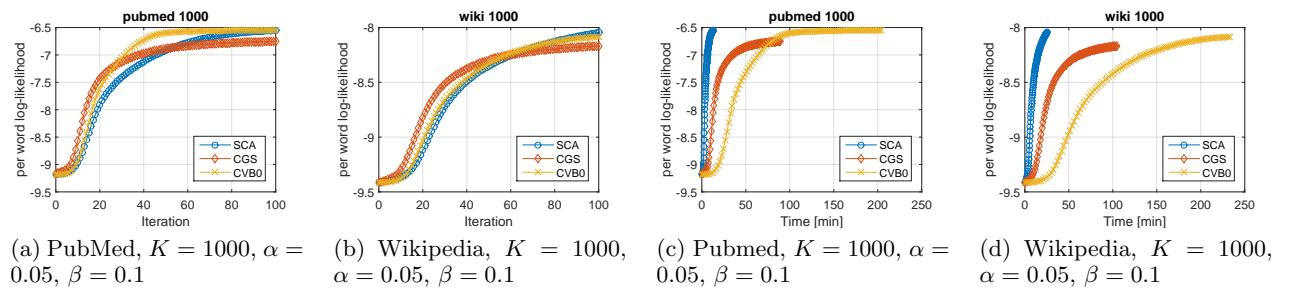


Figure 2: Evolution of log likelihood on Wikipedia and Pubmed over number of iterations and time.

data for the S and M steps, similar to incremental EM [22]. Note that in the limit in which batches comprise just a single token, the algorithm emulates a collapsed Gibbs sampler. This interpretation strengthens the theoretical justification for many existing approximate Gibbs sampling approaches.

## 4 Experiments

To evaluate the strength and weaknesses of our algorithm, we compare against parallel and distributed implementations of CGS and CVB0. We also compare our results to performance numbers reported in the literature including those of F+LDA and lightLDA.

**Software & hardware** All three algorithms are implemented in simple C++11. We implement multithreaded parallelization within a node using the work-stealing Fork/Join framework, and the distribution across multiple nodes using the process binding to a socket over MPI. We also implemented a version of ESCA with a sparse representation for the array  $D$  of counts of topics per documents and Vose’s alias method to draw from discrete distributions. We run our experiments on a small cluster of 8 Amazon EC2 c4.8xlarge nodes connected through 10Gb/s Ethernet. Each node has a 36 virtual threads per node. For random number generation we employ Intel®Digital Random Number Generators through instruction RDRAND, which uses thermal noise within the silicon to output a random stream of bits at 3 Gbit/s, producing true random numbers.

**Datasets** We experiment on two public datasets, both of which are cleaned by removing stop words and rare words: PubMed abstracts and English Wikipedia. To demonstrate scalability, run on 100 copies of English Wikipedia and a third proprietary dataset.

Dataset	V	M	Tokens
PubMed	141,043	8,200,000	737,869,085
Wikipedia	210,233	6,631,176	1,133,050,514
Large	~140,000	~3 billion	~171 billion

**Evaluation** To evaluate the proposed method we use predicting power as a metric by calculating the per-word log-likelihood (equivalent to negative log of perplexity) on 10,000 held-out documents conditioned on the trained model. We set  $K = 1000$  to demonstrate performance for a large number of topics. The hyper parameters are set as  $\alpha = 50/K$  and  $\beta = 0.1$  as suggested in [12]; other systems such as YahooLDA and Mallet also use this as the default parameter setting. The results are presented in Figure 2 and some more experiments in Appendix G.

Finally, for the larger datasets, our implementation of ESCA (only 300 lines of C++) processes more than **1 billion tokens per second** (tps) by using a 20-node cluster. In comparison, some of the best existing systems achieve 112 million tps (F+LDA, personal communication) and 60 million tps (lightLDA) [39] using more hardware. Detailed Appendix G Table 3)

## 5 Discussion

We have described a novel inference method for latent variable models that simulates a stochastic cellular automaton. The equilibrium of the dynamics are MAP fixed points and the algorithm has many desirable computational properties: it is embarrassingly parallel, memory efficient, and like HOGWILD! [27], is virtually lock-free. Further, for many models, it enables the use of approximate counters and the alias method. Thus, we were able to achieve an order of magnitude speed-up over the current state-of-the-art inference algorithms for LDA with accuracy comparable to collapsed Gibbs sampling.

In general, we cannot always guarantee the correct invariant measure [7], and found that parallelizing improperly causes convergence to incorrect MAP fixed points. Even so, SCA is used for simulating Ising models in statistical physics [33]. Interestingly, in previous work [15], it has been shown that stochastic cellular automata are closely related to equilibrium statistical models and the stationary distribution is known for a large class of finite stochastic cellular automata.

## References

- [1] Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee Whye Teh. On smoothing and inference for topic models. In *Proc. Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI '09*, pages 27–34, Arlington, Virginia, USA, 2009. AUAI Press.
- [2] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003.
- [3] J. Canny. Gap: a factor model for discrete data. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 122–129. ACM, 2004.
- [4] Gilles Celeux and Jean Diebolt. The sem algorithm: a probabilistic teacher algorithm derived from the em algorithm for the mixture problem. *Computational statistics quarterly*, 2(1):73–82, 1985.
- [5] Miklós Csűrös. Approximate counting with a floating-point counter. In M. T. Thai and Saritaj Sahni, editors, *Computing and Combinatorics (COCOON 2010)*, number 6196 in Lecture Notes in Computer Science, pages 358–367. Springer Berlin Heidelberg, 2010. See also <http://arxiv.org/pdf/0904.3062.pdf>.
- [6] Rajarshi Das, Manzil Zaheer, and Chris Dyer. Gaussian lda for topic models with word embeddings. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 795–804, Beijing, China, July 2015. Association for Computational Linguistics.
- [7] Donald A. Dawson. Synchronous and asynchronous reversible Markov systems. *Canadian mathematical bulletin*, 17:633–649, 1974.
- [8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [9] Anton K Formann and Thomas Kohlmann. Latent class analysis in medical research. *Statistical methods in medical research*, 5(2):179–211, 1996.
- [10] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter. *Markov Chain Monte Carlo in Practice*. Chapman & Hall, 1995.
- [11] Joseph Gonzalez, Yucheng Low, Arthur Gretton, and Carlos Guestrin. Parallel gibbs sampling: from colored fields to thin junction trees. In *International Conference on Artificial Intelligence and Statistics*, pages 324–332, 2011.
- [12] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.
- [13] Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, May 2013.
- [14] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Mach. Learn.*, 37(2):183–233, November 1999.
- [15] Joel L. Lebowitz, Christian Maes, and Eugene R. Speer. Statistical mechanics of probabilistic cellular automata. *Journal of statistical physics*, 59:117–170, April 1990.
- [16] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the sampling complexity of topic models. In *20th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining*, 2014.
- [17] Pierre-Yves Louis. *Automates Cellulaires Probabilistes : mesures stationnaires, mesures de Gibbs associées et ergodicité*. PhD thesis, Université des Sciences et Technologies de Lille and il Politecnico di Milano, September 2002.
- [18] Jean Mairesse and Irène Marcovici. Around probabilistic cellular automata. *Theoretical Computer Science*, 559:42–72, November 2014.
- [19] David Mimno, Matt Hoffman, and David Blei. Sparse stochastic inference for latent dirichlet allocation. In John Langford and Joelle Pineau, editors, *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, ICML '12, pages 1599–1606, New York, NY, USA, July 2012. Omnipress.
- [20] Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978.
- [21] R. Neal. Markov chain sampling methods for dirichlet process mixture models. Technical Report 9815, University of Toronto, 1998.
- [22] Radford M Neal and Geoffrey E Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. In *Learning in graphical models*, pages 355–368. Springer, 1998.

- [23] A. U. Neumann and B. Derrida. Finite size scaling study of dynamical phase transitions in two dimensional models: Ferromagnet, symmetric and non symmetric spin glasses. *J. Phys. France*, 49:1647–1656, 08 1988.
- [24] David Newman, Arthur Asuncion, Padhraic Smyth, and Max Welling. Distributed algorithms for topic models. *J. Machine Learning Research*, 10:1801–1828, December 2009. <http://dl.acm.org/citation.cfm?id=1577069.1755845>.
- [25] Søren Feodor Nielsen. The stochastic em algorithm: estimation and asymptotic results. *Bernoulli*, pages 457–489, 2000.
- [26] Sam Patterson and Yee Whye Teh. Stochastic gradient riemannian langevin dynamics on the probability simplex. In *Advances in Neural Information Processing Systems*, pages 3102–3110, 2013.
- [27] B. Recht, C. Re, S.J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In Peter Bartlett, Fernando Pereira, Richard Zemel, John Shawe-Taylor, and Kilian Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011.
- [28] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951.
- [29] Ruslan Salakhutdinov, Sam Roweis, and Zoubin Ghahramani. Relationship between gradient and em steps in latent variable models.
- [30] Alexander Smola and Shravan Narayananurthy. An architecture for parallel topic models. *Proc. VLDB Endowment*, 3(1-2):703–710, September 2010.
- [31] Whye Yee Teh, David Newman, and Max Welling. A collapsed variational Bayesian inference algorithm for latent Dirichlet allocation. In *Advances in Neural Information Processing Systems 19*, NIPS 2006, pages 1353–1360. MIT Press, 2007.
- [32] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. Efficient training of LDA on a GPU by Mean-For-Mode Gibbs sampling. In *32nd International Conference on Machine Learning*, volume 37 of *ICML 2015*, 2015. Volume 37 of the Journal in Machine Learning Research: Workshop and Conference Proceedings.
- [33] Gérard Y. Vichniac. Simulating physics with cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):96–116, January 1984.
- [34] Michael D Vose. A linear algorithm for generating random numbers with a given distribution. *Software Engineering, IEEE Transactions on*, 17(9):972–975, 1991.
- [35] Max A Woodbury, Jonathan Clive, and Arthur Garson. Mathematical typology: a grade of membership technique for obtaining disease definition. *Computers and biomedical research*, 11(3):277–298, 1978.
- [36] Lei Xu and Michael I Jordan. On convergence properties of the em algorithm for gaussian mixtures. *Neural computation*, 8(1):129–151, 1996.
- [37] Limin Yao, David Mimno, and Andrew McCallum. Efficient methods for topic model inference on streaming document collections. In *Proc. 15th ACM SIGKDD Intl. Conf. Knowledge Discovery and Data Mining*, KDD ’09, pages 937–946, New York, 2009. ACM.
- [38] Hsiang-Fu Yu, Cho-Jui Hsieh, Hyokun Yun, SVN Vishwanathan, and Inderjit S Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1340–1350. International World Wide Web Conferences Steering Committee, 2015.
- [39] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jin-liang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1351–1361. International World Wide Web Conferences Steering Committee, 2015.
- [40] Ke Zhai, Jordan Boyd-Graber, Nima Asadi, and Mohamad L Alkhouja. Mr. lda: A flexible large scale topic modeling package using variational inference in mapreduce. In *Proceedings of the 21st international conference on World Wide Web*, pages 879–888. ACM, 2012.



# Compiling Markov Chain Monte Carlo Algorithms for Probabilistic Modeling

Daniel Huang

Harvard University  
Cambridge, MA, USA  
dehuang@fas.harvard.edu

Jean-Baptiste Tristan

Oracle Labs  
Burlington, MA, USA  
jean.baptiste.tristan@oracle.com

Greg Morissett

Cornell University  
Ithaca, NY, USA  
jgm19@cornell.edu

## Abstract

The problem of probabilistic modeling and inference, at a high-level, can be viewed as constructing a (*model, query, inference*) tuple, where an *inference* algorithm implements a *query* on a *model*. Notably, the derivation of inference algorithms can be a difficult and error-prone task. Hence, researchers have explored how ideas from *probabilistic programming* can be applied. In the context of constructing these tuples, probabilistic programming can be seen as taking a language-based approach to probabilistic modeling and inference. For instance, by using (1) appropriate languages for expressing models and queries and (2) devising inference techniques that operate on encodings of models (and queries) as program expressions, the task of inference can be automated.

In this paper, we describe a compiler that transforms a probabilistic model written in a restricted modeling language and a query for posterior samples given observed data into a Markov Chain Monte Carlo (MCMC) inference algorithm that implements the query. The compiler uses a sequence of intermediate languages (ILs) that guide it in gradually and successively refining a declarative specification of a probabilistic model and the query into an executable MCMC inference algorithm. The compilation strategy produces composable MCMC algorithms for execution on a CPU or GPU.

**CCS Concepts** • Mathematics of computing → Bayesian networks; Bayesian computation; Markov-chain Monte Carlo methods; • Software and its engineering → Compilers

**Keywords** probabilistic programming, intermediate languages, Markov-chain Monte Carlo kernels

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [Permissions@acm.org](mailto:Permissions@acm.org).

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

*PLDI'17*, June 18–23, 2017, Barcelona, Spain  
ACM, 978-1-4503-4988-8/17/06...\$15.00  
<http://dx.doi.org/10.1145/3062341.3062375>

## 1. Introduction

Consider the problem of clustering a set of data points in  $D$ -dimensional Euclidean space  $\mathbb{R}^D$  into  $K$  clusters. One approach is to construct a *probabilistic (generative) model* to explain how we believe the observations are generated. For instance, one explanation might be that there are (1)  $K$  cluster centers chosen randomly according to a normal distribution and (2) that each data point (independently of each other data point) is normally distributed around a randomly chosen cluster center. This describes what is known as a Gaussian Mixture Model (GMM).

More generally, a probabilistic model induces a probability distribution, which we can *query* for quantities of interest. For example, the query “what is the most likely cluster assignment of each observation under the GMM model?” formulates our original problem of clustering a collection of points in probabilistic terms. To answer such a query, practitioners implement an *inference algorithm*. Inference is analytically intractable in general. Consequently, practitioners devote a significant amount of time to developing and implementing *approximate* inference algorithms to (approximately) answer a query for a given model.

In summary, we can think of the problem of probabilistic modeling and inference as constructing a (*model, query, inference*) tuple, where the *inference* algorithm implements a *query* on a *model*. To simplify this task, researchers have explored how ideas from *probabilistic programming* can be applied. In the context of constructing these tuples, probabilistic programming can be seen as taking a language-based approach to probabilistic modeling and inference. For instance, by using (1) appropriate languages for expressing models and queries and (2) devising inference techniques that operate on encodings of models (and queries) as program expressions, the task of inference can be automated. Due to the promise of this approach, many probabilistic programming languages (PPLs) have been proposed in the literature to explore the various points of the design space [5, 10, 11, 15, 16, 23, 26, 28, 29].

In this paper, we present a tool called AugurV2 for constructing (*model, query, inference*) tuples of a restricted

form, where (1) the *model* is expressed in a domain-specific modeling language similar in expressive power to Bugs [23] or Stan [8] (as compared to modeling languages embedded in general-purpose languages [11, 28]), (2) the *query* is for posterior samples given observed data, and (3) the *inference* is automatically derived and is restricted to a family of algorithms based on MCMC sampling (as compared to languages that provide a set of fixed inference strategies [8, 23, 29] or those that provide domain-specific languages for the specification of inference [5, 14, 26]). Even in this restricted setting, automating posterior inference based on MCMC poses at least three challenges.

- As inference is intractable in general, it is unlikely that a black-box approach to MCMC will work well for all probabilistic models of interest. Consequently, we would like a way to derive model-specific facts useful for constructing MCMC algorithms. For instance, systems such as Bugs [23] leverage conjugacy relations and Stan [8] leverages gradients.
- There are many kinds of MCMC algorithms. Ideally, our tool should support as many as possible, given that these algorithms may exhibit different computational and statistical behaviors depending on the model to which it is applied. More generally, end users may want more fine-grained control over how to perform inference. For example, Blaise [5] provides a domain-specific graphical language for expressing models and inference algorithms. Other systems such as Edward [26] and Venture [14] explore other designs for providing fine-grained control of inference.
- MCMC sampling is computationally expensive [7]. To improve the computational efficiency, we may want to leverage parallelism such as in our previous work on Augur [27] and other systems such as Anglican [28].

Our solution is to come up with a sequence of intermediate languages (ILs) that enable our tool to gradually and successively refine a *declarative* specification of a model and a query for posterior samples into an *executable* Markov Chain Monte Carlo (MCMC) inference algorithm. For the purposes of this paper, we will refer to the process above as *compilation*, and hence, will also refer to our tool as a compiler. We address the challenges listed above with the following aspects of our compiler design.

- The compiler uses a *Density IL* to express the density factorization of a model (Section 3.1). This IL can be analyzed to symbolically compute the conditionals of a model so that the compiler can generate *composable* MCMC algorithms.
- The compiler uses a *Kernel IL* to express the high-level structure of a MCMC algorithm as a composition of MCMC algorithms (Section 4.1). In particular, the AugurV2 compiler supports multiple kinds of basic

MCMC algorithms, including ones that leverage conjugacy relations (e.g., Gibbs) and gradient information (e.g., HMC). The Kernel IL is similar to the subset of the Blaise language [5] that corresponds to inference. We use the IL for the purposes of compilation, whereas Blaise focuses on the expressivity of the language.

- The compiler uses *Low++* and *Low-- ILs* to express the details of MCMC inference (Section 4.3). The first exposes parallelism and the second exposes memory management. The AugurV2 compiler leverages these ILs to support both CPU and GPU (Section 5) compilation of composable MCMC algorithms.

Our preliminary experiments show that such a compilation strategy both enables us to leverage the flexibility of composable MCMC inference algorithms on the CPU and GPU as well as improve upon the scalability of automated inference (Section 7). We have also found it relatively easy to add new base MCMC updates to the compiler without restructuring its design. Hence, the compiler is relatively extensible. It would be an interesting direction to see what aspects of AugurV2’s ILs and compilation strategy could be reused in the context of a larger infrastructure for compiling PPLs, but that is not in scope for this paper. AugurV2 is available at <https://github.com/danehuang/augurv2/>.

## 2. AugurV2 Overview

At a high-level, the AugurV2 modeling language expresses *Bayesian networks* whose graph structure is fixed. This class contains many practical models of interest, including regression models, mixture models, topic models, and deep generative models such as sigmoid belief networks. Models that cannot be expressed include ones that use non-parametric distributions (which can be encoded in a general-purpose language) and models with undirected dependency structure (e.g., Markov random fields whose dependency structure is hard to express in a functional setting). Note that even in this setting, inference can still be difficult due to the presence of high-dimensional distributions. For example, the dimensionality of a mixture model such as the GMM scales with the number of observations—each observed point introduces a corresponding latent (*i.e.*, unobserved) cluster assignment. Consequently, the design of AugurV2 focuses on generating efficient MCMC algorithms for posterior inference on this restricted class of models.

### 2.1 Probabilistic Modeling Setup

AugurV2 provides a simple, first-order modeling language for expressing probabilistic generative models with *density factorization*

$$p(\theta, y) = p(\theta) p(y | \theta),$$

where  $p(\theta)$  is a distribution over parameters  $\theta$  called the *prior* and  $p(y | \theta)$  is the *conditional distribution* of the data  $y$  given the parameters  $\theta$ . In this paper, we will only

```
(K, N, mu_0, Sigma_0, pis, Sigma) => {
    param mu[k] ~ MvNormal(mu_0, Sigma_0)
    for k <- 0 until K ;
    param z[n] ~ Categorical(pis)
    for n <- 0 until N ;
    data x[n] ~ MvNormal(mu[z[n]], Sigma)
    for n <- 0 until N ;
}
```

**Figure 1:** An AugurV2 program encoding a GMM. At a high-level, the modeling language mirrors random variable notation.

consider probability distributions with densities<sup>1</sup>, and hence, will interchangeably use the two terms. Given observed data, written  $y^*$  to distinguish it from the formal parameter  $y$  in the distribution  $p(y | \theta)$ , the AugurV2 system implements a MCMC algorithm to draw samples from the *posterior distribution*

$$p(\theta | y^*) = \frac{1}{Z} p(\theta) p(y^* | \theta),$$

where  $Z = \int p(\theta) p(y^* | \theta) d\theta$ . The notation  $p(y^* | \theta)$  is perhaps better written as  $p(y = y^* | \theta)$  to indicate that it is a function of  $\theta$  and is known as the *likelihood function*. In the rest of the section, we will show how to encode the GMM we gave in the introduction in AugurV2’s modeling language, how to use the system to perform posterior inference, and overview the compilation process.

## 2.2 Modeling Language

Figure 1 contains an AugurV2 program expressing the GMM we introduced earlier. At a high-level, the modeling language is designed to mirror random variable notation. The *model body* is a sequence of declarations, each consisting of a random variable and its distribution. Each declaration is annotated as either a model parameter (`param`) or observed data (`data`). Model parameters are inferred (*i.e.*, output) whereas model data is supplied by the user (*i.e.*, input). In the case of a GMM, the means  $\mu$  and cluster assignments  $z$  are model parameters<sup>2</sup>, while the  $x$  values are model data. It is also possible to define a random variable as a deterministic transformation of existing variables, although this feature is not needed to express the GMM in this example.

At the top-level, the model closes over any free variables mentioned in the model body. These include the model hyper-parameters (`mu_0, Sigma_0, pis, Sigma`) and any other variables over which the model is parameterized by (`K, N`). The modeling language can be considered as alternative notation for expressing the density factorization, which

<sup>1</sup> That is, with respect to Lebesgue measure, counting measure, or their products.

<sup>2</sup> To be pedantic, we should call these latent variables.

```
import AugurV2Lib
import numpy as np

# Part 1: Load data
x = load_gmm_data('/path/to/data')
N, D = x.shape; K = 3
mu0 = np.zeros(D); S0 = np.eye(D);
S = np.eye(D); pis = np.full(K, 1.0/K)

# Part 2: Invoke AugurV2
with AugurV2Lib.Infer('path/to/model') as aug:
    opt = AugurV2Lib.Opt(target='cpu')
    aug.setCompileOpt(opt)
    sched = 'ESlice mu (*) Gibbs z'
    aug.setUserSched(sched)
    aug.compile(K, N, mu0, S0, pis, S)(x)
    samples = aug.sample(numSamples=1000)
```

**Figure 2:** Fitting a GMM with AugurV2 using a Python interface.

we give for the GMM below.

$$\prod_{k=1}^K p(\mu_k | \mu_0, \Sigma_0) \prod_{n=1}^N p(z_n | \pi) p(y_n^* | \mu_{z_n})$$

Random vectors (*e.g.*, `mu[k]`) are specified using comprehensions with the `for` construct. The semantics of AugurV2 comprehensions are *parallel*, meaning that they do not depend on the order of evaluation. The idea is to provide a syntactic construct that corresponds to the mathematical phrase “let  $\mu_k \sim \mathcal{N}(\vec{\mu}_0, \Sigma)$  for  $0 \leq k < K$ .” Because such a mathematical statement is implicitly parallel and occurs frequently in the definition of probabilistic models, we opt for syntactic constructs that capture standard statistical practice, instead of more standard programming language looping constructs (*e.g.*, an imperative `for` loop).

As AugurV2 provides only parallel comprehensions, we discourage users from expressing models with sequential dependencies. For example, we would need to write a Hidden Markov Model, where each hidden state depends on the previous state, by unfolding the entire model. This is *doable*, but does not take advantage of the design of AugurV2.

AugurV2 imposes two further restrictions. First, comprehension bounds cannot mention model parameters. This forces the comprehension bounds to be constant (although they can still be ragged). For this reason, we say that AugurV2 expresses *fixed-structure* models. Second, AugurV2 provides only primitive distributions whose probability density function (PDF) or probability mass function (PMF) has known functional form. Hence, the models AugurV2 expresses are *parametric*. Currently, AugurV2 does not support non-parametric distributions (*i.e.*, distributions with an infinite number of parameters so that the number of parameters used scales with the number of observations).

### 2.3 Using AugurV2

Figure 2 contains an example of how to invoke AugurV2’s inference capabilities contained in the Python module AugurV2. The first part of the code loads the data and hyper-parameters and the second part invokes AugurV2.

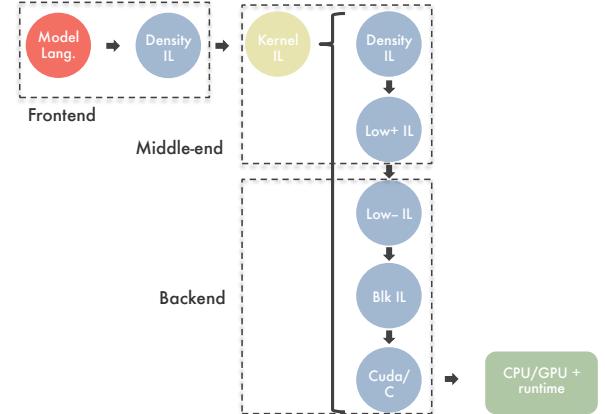
Instances of the `AugurV2Infer` class provide methods for obtaining posterior samples (*e.g.*, `sample`). The class takes a path to a file containing the model. Once we have created an instance of an AugurV2 inference object (`aug`), we need to indicate to the compiler what kind of MCMC sampler it should generate (via `setCompileOpt` and `setUserSched`). For example, we can set the target for compilation as either the CPU or GPU (`target`). We can also customize the MCMC algorithm by choosing our own MCMC schedule (via `setUserSched`).

In this example, we decide to apply Elliptical Slice sampling [17] to the cluster means (`ESlice mu`) and Gibbs sampling to the cluster assignments (`Gibbs z`). Hence, this schedule indicates a *compositional* MCMC algorithm, where we apply different MCMC updates to different portions of the model. This feature is inspired by the *programmable inference* proposed by other probabilistic programming systems (*e.g.*, Venture [14]). If a user schedule is not specified, the compiler uses a heuristic to select which combination of MCMC methods to use.

To compile the model, we supply the model arguments, hyper-parameters, and data (as Python variables) in the order that they are specified in the model. Thus, the AugurV2 compiler is invoked at *runtime*. Consequently, given different data sizes and hyper-parameter settings, the AugurV2 compiler may choose to generate a different MCMC algorithm. The compiler generates Cuda/C code depending on whether the target is the GPU or the CPU. The native inference code is then further compiled using Nvcc (the Cuda compiler) or Clang into a shared library which contains inference code for a specific instantiation of a model. After compilation, the object `aug` contains a collection of inference methods that wrap the native inference code. The Python interface handles the conversion of Python values to and from Cuda/C via the Python CTypes interface. Hence, the user can work exclusively in Python. In this example, we ask for 1000 samples from the posterior distribution.

### 2.4 Compilation Overview

The compiler uses ILs to structure the compilation (summarized in Figure 3) from model and query into inference.<sup>3</sup> Notably, the ILs enable the AugurV2 compiler to successively and gradually refine a *declarative* specification of a model and query into an *executable* inference algorithm that contains details such as memory consumption and parallelism. The successive removal of layers of abstraction is the typ-



**Figure 3:** Overview of the AugurV2 compilation process. It is comprised of a Frontend (model to density factorization), Middle-end (density factorization to executable inference), and Backend (executable inference to native inference).

ical manner in which ILs aid the compilation of traditional language.

Of course, we will need a semantics to justify the compilation process. The semantics of probabilistic programs is an active area of research due to the need to model continuous distributions and its interaction with standard programming language features [6, 12, 25]. As AugurV2 provides a simple language that intuitively expresses Bayesian networks, its semantics is not an issue, in contrast to more expressive languages that provide higher-order functions and recursion. Nevertheless, we point out that AugurV2 can be given semantics in terms of (Type-2) computable distributions [12]. This enables us to think of a compiler as a (computable) function that witnesses the result that (Type-2) computable distributions are realizable by (Type-2) computable sampling algorithms. In our case, the compiler realizes a MCMC sampling algorithm.

### 3. Frontend

In the first step of compilation, the compiler transforms a model expressed in the modeling language into its corresponding density factorization in the *Density IL*. This follows standard statistical practice, where models expressed using random variables (AugurV2’s modeling language) are converted into its description in terms of densities (Density IL). The compiler can analyze the density factorization and symbolically compute the *conditionals* (up to a normalizing constant) of the model for each model parameter (see Section 3.3) to support the generation of composable MCMC algorithms. The analysis is based off the one implemented in our previous work [27], although the results of the analysis then were only used to generate Gibbs samplers.

<sup>3</sup> The first author’s dissertation contains more details about AugurV2 and its semantics.

```

obj ::=  $\lambda(\vec{x}). fn$            gen ::= e until e
fn ::=  $p_{dist(\vec{e})}(e)$  | fn fn |  $\prod_{x \leftarrow gen} fn$ 
      | let  $x = e$  in fn |  $[fn]_{x=e}$ 
e ::= x | i | r | dist( $\vec{e}'$ ) | opn( $\vec{e}'$ ) | e[e]
 $\sigma$  ::= Int | Real        $\tau ::= \sigma$  | Vec  $\tau$  | Mat  $\sigma$ 

```

**Figure 4:** The Density IL encodes the density factorization of a probabilistic model.

### 3.1 Representing Models: The Density IL

The syntax for the Density IL is summarized in Figure 4. It encodes the density factorization of a model. As an example, the GMM from before (Figure 1) is encoded in the Density IL as

$$\lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, x). \prod_{k \leftarrow gen_1} p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k]) \\ \prod_{n \leftarrow gen_2} p_{\mathcal{D}(\pi)}(z[n]) \prod_{n \leftarrow gen_2} p_{\mathcal{N}(\mu[z[n]], \Sigma)}(x[n] \mid \mu[z[n]])$$

where  $gen_1 = 0$  until  $K$  and  $gen_2 = 0$  until  $N$ .

At the top-level, a model is represented as a single function  $\lambda(\vec{x}). fn$  with bindings  $\vec{x}$  for model hyper-parameters, meta-parameters, and data, and a density function body  $fn$ . A density function is either (1) the density of a primitive, parameterized distribution  $p_{dist(\vec{e})}$ , (2) the composition of two density functions  $fn_1 fn_2$  (*i.e.*, multiplication) of two density functions, (3) a structured product  $\prod_{x \leftarrow gen} fn$  that specifies a product of density functions according to the comprehension  $x \leftarrow gen$ , (4) a standard let-binding `let  $x = e$  in fn`, or (5) an indicator function  $[fn]_{x=e}$  that takes on the value inside the brackets if the condition  $x = e$  is satisfied and 1 otherwise.

The Density IL is simply-typed. Base types include integers `Int` and reals `Real`. Compound types include vectors `Vec  $\tau$`  and matrices `Mat  $\sigma$` . Thus, compound types such as vectors of matrices are allowed, whereas matrices of vectors are rejected. The type system is used to check simple properties. For example, the type system checks that densities are defined on the appropriate spaces and that the comprehension bounds  $gen$  in a product of density function  $\prod_{x \leftarrow gen} fn$  is indeed a vector of integers.

### 3.2 MCMC and Conditionals

Before we describe the symbolic computation on the Density IL, we briefly introduce background on MCMC. Suppose we would like to construct a sampler to draw from the probability density  $p(x, y)$ . The Metropolis Hastings (MH) algorithm is a MCMC algorithm that gives a sampler for  $p(x, y)$  given a transition function  $q(x, y \rightarrow x', y')$ , *i.e.*, a

conditional density  $q(x', y' \mid x, y)$ <sup>4</sup>. The transition function is also called a *proposal*. In particular, every MCMC algorithm can be seen as a special case of the MH algorithm with a proposal of a specific form. For instance, the HMC algorithm uses a gradient-based proposal [7].

Given a multivariate distribution such as  $p(x, y)$ , it may be undesirable to construct the entire proposal  $q(x, y \rightarrow x', y')$  in a single step. For instance, the GMM has both discrete and continuous variables, so one cannot directly apply the HMC algorithm (which uses gradients) to the entire model without handling the discrete variables in a special manner. Here, it is useful to construct the MCMC sampler as the *composition* of two MCMC samplers, one that targets the conditional  $p(x \mid y)$  and another that targets the conditional  $p(y \mid x)$ . In particular, this involves constructing the two simpler proposals  $q_1(x \rightarrow x'; y)$  (holding  $y$  fixed) and  $q_2(y \rightarrow y'; x)$  (holding  $x$  fixed). For situations like this, the AugurV2 compiler supports the symbolic computation of conditionals.

### 3.3 Approximating Conditionals

In a more traditional setting, one might represent the density factorization of a model using a Bayesian network. As AugurV2 programs denote Bayesian networks, we could compile the Density IL into a Bayesian network and use standard, graph-based algorithms for determining how the conditionals factorize. However, the resulting graph could be quite large and would be difficult to use in other phases of compilation (*e.g.*, for generating GPU code). Instead, the compiler symbolically computes the conditionals of the model.

Computing the conditional of the density factorization  $p(x)p(y \mid x)p(y \mid z)$  with respect to (w.r.t.)  $x$  up to a normalizing constant is equivalent to simplifying the expression below.

$$p(x \mid y, z) = \frac{p(x)p(y \mid x)p(y \mid z)}{\int p(x)p(y \mid x)p(y \mid z)dx}$$

The expression simplifies to

$$p(x \mid y, z) \propto p(x)p(y \mid x)$$

where we cancel the terms that have no functional dependence on  $x$ . This computation is isomorphic to the computation of conditional independence relationships in Bayesian networks [4].

The challenge with symbolically computing conditionals of the density factorization comes from handling structured products in the Density IL. Conceptually, these products can be unfolded because we compile at runtime when the size of the structured product is known. However, the resulting density factorization loses regularity and can be quite large. Hence, the compiler treats structured products symbolically at the cost of precision of the conditional computed.

<sup>4</sup>More generally,  $q$  would be a *probability kernel*.

For example, suppose we would like to compute the conditional of

$$\prod_{k \leftarrow \text{gen}} p(x_k) \prod_{k' \leftarrow \text{gen}} p(y_{k'} | x_{k'})$$

w.r.t.  $x_k$  for some  $k$ . In particular, the compiler cannot tell that this expression simplifies to  $p(x_k) p(y_k | x_k)$  because the terms are syntactically different. To handle this, the compiler uses a factoring rewrite rule, given below.

$$\prod_{i \leftarrow \text{gen}_1} fn_1 \prod_{j \leftarrow \text{gen}_2} fn_2 \rightarrow \prod_{i \leftarrow \text{gen}_1} fn_1 fn_2 \text{ when } \text{gen}_1 = \text{gen}_2$$

As a reminder, comprehension bound expressions in AugurV2 cannot refer to random variables and so are constant. If the compiler cannot determine the equality of comprehension bounds, it will not be factored, and precision in the approximation of the conditional can be lost.

The compiler also implements the normalization rule

$$\prod_{i \leftarrow \text{gen}_i} fn \rightarrow \prod_{k \leftarrow \text{gen}_k} \prod_{i \leftarrow \text{gen}_i} [fn]_{k=z},$$

where  $z$  is a Categorical variable with range  $\text{gen}_k$  mentioned in  $fn$ . This rule deals with mixture models, which is a common pattern in probabilistic models. For example, the conditional of  $\mu$  for the GMM (omitting the bounds)

$$\prod_k p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k]) \prod_n p_{\mathcal{N}(\mu[z[n]], \Sigma)}(y[n])$$

can be transformed into

$$\prod_k p_{\mathcal{N}(\mu_0, \Sigma_0)}(\mu[k]) \prod_k \prod_n [p_{\mathcal{N}(\mu[z[n]], \Sigma)}(y[n])]_{k=z[n]},$$

after which we can apply the factoring rule. This says that each cluster location  $\mu_k$  depends only on the data points  $y_n$  whose cluster assignment  $z_n$  is associated with the current cluster  $k$ . Currently, the compiler attempts to apply the categorical indexing rule first and then attempts to factor. For the restricted modeling language that we consider, we have found the two normalization rules to be precise enough for practical use.

#### 4. Middle-End

The AugurV2 middle-end converts a model encoded as its density factorization into a high-level, executable inference algorithm. This phase uses two additional ILs. First, the *Kernel IL* represents the high-level structure of a MCMC algorithm as a composition of basic MCMC updates applied to the model’s conditionals. Second, the *Low++ IL* serves as the first pass target for executable MCMC code. It is an imperative language that makes sources of parallelism explicit, but abstracts away memory management. Importantly, we can leverage domain-specific knowledge of each base MCMC update to directly annotate parallelism in the code. Thus, we do not need to (re)discover parallelism at a lower-level of abstraction.

$$\begin{aligned} \text{sched } \alpha &::= \lambda(\vec{x}). k \alpha \\ k \alpha &::= (\kappa \alpha) \text{ku } \alpha \mid k \alpha \otimes k \alpha \\ \text{ku} &::= \text{Single}(x) \mid \text{Block}(\vec{x}) \\ \kappa \alpha &::= \text{Prop}(\text{Maybe } \alpha) \mid \text{FC} \\ &\quad \mid \text{Grad}(\text{Maybe } \alpha) \mid \text{Slice} \end{aligned}$$

**Figure 5:** The Kernel IL encodes the structure of an MCMC algorithm. It is parametric in  $\alpha$ , which is instantiated with successively lower level ILs that encode how the MCMC algorithm is implemented.

#### 4.1 Representing MCMC I: The Kernel IL

The AugurV2 compiler represents a MCMC algorithm as a composition of base updates in the *Kernel IL*, whose syntax is summarized in Figure 5. As an example, the user schedule presented in Figure 2 is encoded in the Kernel IL as

$$\begin{aligned} \lambda(K, N, \mu_0, \Sigma_0, \pi, \Sigma, \mu, z, x). \\ \text{Slice Single}(\mu) fn_\mu \otimes \text{FC Single}(z) fn_z, \end{aligned}$$

where  $fn_\mu$  and  $fn_z$  correspond to the conditionals of  $\mu$  and  $z$  up to a normalizing constant respectively.

The syntax

$$\text{Slice Single}(\mu) fn_\mu$$

encodes a base update indicating that we apply Slice sampling to the variable  $\mu$  with proportional conditional  $fn_\mu$ . The kernel unit  $ku$  specifies whether we should sample a variable  $x$  by itself (*i.e.*, `Single(x)`), or whether to sample a list of variables  $\vec{x}$  jointly (*i.e.*, `Block(x)`). Sampling variables jointly, also known as blocking, is useful when those variables are heavily correlated. In general, a base update  $\kappa \ ku \ \alpha$  is parametric in the representation of the proportional conditional  $\alpha$ . This enables the compiler to successively instantiate  $\alpha$  with lower-level ILs that expose more computational details of inference (*e.g.*, parallelism or memory usage). Given two MCMC updates  $k_1 \ \alpha$  and  $k_2 \ \alpha$ , we can sequence (*i.e.*, compose) the two updates with the syntax  $k_1 \ \alpha \otimes k_2 \ \alpha$ . Sequencing is not commutative, *i.e.*, the MCMC update  $k_2 \ \alpha \otimes k_1 \ \alpha$  is different from  $k_1 \ \alpha \otimes k_2 \ \alpha$ .

In addition to `Slice` (`Slice`) and closed-form conditional updates (`FC`), the Kernel IL also supports proposal-based updates (`Prop`) and gradient-based updates (`Grad`). These updates each contain an optional piece of code of type  $\alpha$  that specifies the proposal and the gradient respectively for the corresponding conditional that the base update is applied to. In contrast, `Slice` and closed-form conditional updates can be derived purely from the model conditionals.

The ergodic convergence of base MCMC kernels in the Kernel IL can be established via standard proofs in the MCMC literature [7, 24]. Composite kernels created with

```

 $\text{decl ::= } \text{name}(\vec{x})\{\text{global} : \vec{g}, \text{body} : e, \text{ret} : e\}$ 
 $s ::= e \mid x \text{ } sk \text{ } e \mid e[\vec{e}] \text{ } sk \text{ } e \mid s \text{ } s$ 
 $\mid \text{if}(e)\{s\}\{s\} \mid \text{loop } lk(i \leftarrow \text{gen})\{s\}$ 
 $sk ::= = \mid +$ 
 $lk ::= \text{Seq} \mid \text{Par} \mid \text{AtmPar}$ 
 $e ::= x \mid i \mid r \mid \text{dist}(\vec{e}).dop \mid op^n(\vec{e}) \mid e[e]$ 
 $dop ::= \text{ll} \mid \text{samp} \mid \text{gradi}$ 

```

**Figure 6:** The Low++ IL is an imperative language that exposes parallelism in the computation of a MCMC update, but abstracts away from details of memory management.

⊗ can be shown to preserve the weaker property of invariance of the target distribution [24]. Ergodic convergence of a compound MCMC kernel, which requires considerations of irreducibility and aperiodicity, is not checked by the compiler. Static checking of such properties would be an interesting direction for future work.

#### 4.2 Specifying a High-Level MCMC Algorithm

Once the compiler has decomposed the model according to its unnormalized conditionals, it will determine which base MCMC updates can be applied to each conditional. In this step, the compiler simply chooses *which* updates to apply and not *how* to implement them. Thus, the output is a program in the Kernel IL with conditionals specified in the Density IL. As a reminder, the user can supply the MCMC schedule, in which case the compiler will check that it can indeed generate the desired schedule and fail otherwise. When a user does not supply a MCMC schedule, the compiler uses a heuristic to accomplish this task. First, it determines which variables it can perform Gibbs sampling with conjugacy relations. For the remaining discrete variables, it will also apply Gibbs sampling by approximating the closed-form conditional. For the remaining continuous variables, it will apply HMC sampling to take advantage of gradient information.

#### 4.3 Representing MCMC II: The Low++ IL

The Low++ IL expresses parallelism available in a MCMC algorithm. After the compiler generates code in this IL, the compiler only needs to reason at the level of an inference algorithm—all aspects of the model are eliminated. Figure 6 summarizes the syntax for the Low++ IL. The language is largely standard, so we highlight the aspects useful for encoding MCMC.

First, the IL provides additional distributions operations *dop*, including the log-likelihood *ll*, sampling *samp*, and gradients *gradi*, where the integer *i* refers to the position of

	MCMC update	likelihood	full-conditional	gradient
MH	✓		✗	✗
Gibbs	✗		✓	✗
HMC	✓		✗	✓
Reflective slice	✓		✗	✓
Elliptical slice	✓		✗	✗

**Figure 7:** A summary of base MCMC updates and the primitives required to implement them.

the argument to take the gradient with respect to. At the level of the Density IL, the distribution operation was implicitly its density. For expressing inference algorithms, we may need other operations on distributions such as sampling from them.

Second, the IL contains a dedicated increment and assign statement  $x += e$ . (We can also increment and store to locations  $e[\vec{e}] += e$ .) We added this additional syntactic category because many MCMC updates require incrementing some quantity. For example, we may need to count the number of occurrences satisfying some predicate to compute a conjugacy relation or accumulate derivative computations after an application of the chain-rule. Because we hope to generate parallel inference code, this separate syntactic category indicates to the compiler that the increment and assign must be done atomically.

Third, The ILs annotate loops with whether they can be executed sequentially (Seq), in parallel (Par), or in parallel given that all increment and assign operations are done atomically (AtmPar). For instance, we can annotate a loop that samples a collection of conditionally independent variables in parallel (e.g., when implementing a conjugacy relation) with Par. As we will see later when we describe AugurV2’s implementation of gradients (Section 4.4), these computations will use the loop annotation AtmPar.

#### 4.4 Primitive Support for Base MCMC Updates

The AugurV2 compiler currently supports user-supplied MH proposals, Gibbs updates, HMC updates<sup>5</sup>, and (reflective and Elliptical) Slice updates. Fortunately, each base update can be decomposed into yet further primitives, summarized in Figure 7. The rest of the functionality can be supported as library code—the primitives encapsulate the parts of the MCMC algorithm that are specific to the full-conditional. This helps us manage the complexity of the compiler. These primitives include (1) likelihood evaluation, (2) closed-form full-conditional derivation, and (3) gradient evaluation. The compiler will implement these primitives in the Low++ IL.

**Likelihood evaluation** It is straightforward to generate Low++ code that reifies a likelihood computation from a

<sup>5</sup>There is also a prototype of No-U-Turn sampling.

density factorization. It is also straightforward to parallelize these computations as a map-reduce.

**Closed-form conditional derivation** The compiler only computes the conditionals of the model’s density factorization up to a normalizing constant. To obtain a closed-form solution for the conditional, the compiler needs to solve for the normalizing constant, which requires solving an integral. As this is not analytically possible in general, the AugurV2 compiler supports closed-form conditionals in two cases.

First, like Bugs and Augur, AugurV2 exploits conjugacy relations. Recall that a *conjugacy relation* exists when the form of the conditional distribution  $p(\theta \mid x)$  takes on the same functional form as  $p(\theta)$ . There is a well-known list of conjugacy relations. Consequently, the AugurV2 compiler supports conjugacy relations via table lookup. Computing a conjugacy relation typically involves traversing the involved variables and computing some simple statistic of the variables.

The compiler may fail to detect a conjugacy relation if (1) the approximation of the conditional is imprecise or (2) the compiler needs to perform mathematical rearrangements beyond structural pattern matching. It would be interesting to see if we can improve upon the latter situation by combining AugurV2 with a computer algebra system (CAS) as other systems have done (*e.g.*, [18]) and leveraging the CAS to solve the integral, but we leave this for future work.

Second, AugurV2 can also approximate the closed-form conditional for a discrete variable as a finite sum, even if a conjugacy relation does not exist. That is, it can generate code that directly sums over the support of the discrete variable up to some predetermined bound.

$$p(z = k \mid x) = \frac{p(z = k) p(x \mid z = k)}{\sum_{k'} p(z = k') p(x \mid z = k')}$$

**Gradient evaluation** The compiler implements source-to-source, reverse-mode automatic differentiation (AD) to support gradient evaluation of the model likelihood. For more background, we refer the reader to the literature on AD (*e.g.*, [2]). Figure 8 summarizes the translation implemented by the AugurV2 compiler, where the input program is assumed to contain only simple expressions. The output is an *adjoint program*, *i.e.*, a program that computes the derivative. We highlight two interesting aspects of AD in the context of AugurV2.

First, the AugurV2 compiler implements source-to-source AD, in contrast with other systems (*e.g.*, Stan [8]) that implement AD by instrumenting the program. This choice was largely motivated by the lack of complex control flow in the AugurV2 modeling language, which is the primary difficulty of implementing source-to-source AD. Here, the AugurV2 compiler leverages the semantics of parallel comprehensions to optimize the gradient code. Observe that the translation reverses the order of evaluation (see the sequence translation). In particular, this means that a sequential looping construct

needs a stack to save the order of evaluation. However, as parallel comprehensions have order-independence semantics, the stack can be optimized away. We also found that it was easier to support the composition of different MCMC algorithms by directly generating code that implements gradients when needed, instead of designing the system runtime to support the instrumentation required for AD. Lastly, this choice makes it easier to support compilation to the GPU as we do not need to write two runtimes, one for the CPU and one for the GPU.

Second, for the purposes of parallelization, the compiler generates atomic increments in the AD translation when accumulating gradient computations (*e.g.*, see Figure 8a). The presence of these atomic increments means that parallelization is not straightforward. To illustrate the problem, consider applying the AD transformation to the GMM model conditional  $p(\mu \mid z, y^*)$ , which results in the (excerpted) code:

```
grad_mu_k(K, N, ..., Sigma, mu, z, y) {
    ...
    loop AtmPar (n <- 0 until N) {
        t0 = z[n];
        t1 = MvNormal(mu[t0], Sigma[t0], y[n]).grad2;
        adj_mu[t0] += adj_ll * t1;
    }
    ret adj_mu[k];
}
```

When the number of data points  $N$  greatly exceeds the number of clusters  $K$ , launching  $N$  threads in parallel while executing the increments atomically can lead to high contention, and consequently, poor performance. Hence, the compiler needs to parallelize loops marked *AtmPar* carefully to reduce contention in an atomic increment and assign (Section 5.4).

## 5. Backend

The AugurV2 backend performs the last step of compilation and generates *native* inference code. Except for the final step where the compiler synthesizes a complete MCMC algorithm and eliminates the Kernel IL, this step is largely similar to a traditional compiler backend in terms of the transformations it performs. For AugurV2, this includes (1) size-inference to statically bound the amount of memory usage an AugurV2 MCMC algorithm consumes and (2) reifying parallelism. Towards this end, the compiler uses an IL called the *Low-- IL*.

### 5.1 Representing MCMC III: The Low-- IL

The Low-- IL is structurally the same as the Low++ IL (Figure 5), except that programs must manage memory explicitly. However, details of how to reify parallelism are still left abstract.

$$\begin{aligned}
\overline{(x, y)} &= \bar{y} += \bar{x} \\
\overline{(x, dist(y_1, \dots, y_n))} &= \bar{y}_1 += \bar{x} * dist(y_1, \dots, y_n).grad1 \\
&\dots \\
&\quad \bar{y}_n += \bar{x} * dist(y_1, \dots, y_n).gradn \\
\overline{(x, y_1[y_2])} &= \bar{y}_1[y_2] += \bar{x}
\end{aligned}$$

**(a)** Selected adjoint expression translation. A variable notated  $\bar{x}$  is the adjoint of variable  $x$ . It contains the result of the partial derivative with respect to  $x$ .

**Figure 8:** The construction of an adjoint program for source-to-source, reverse-mode AD in AugurV2 from the Density IL to the Low++ IL.

$$\begin{aligned}
b ::= & \text{seqBlk } \{s\} \mid \text{parBlk } lk \ x \leftarrow gen \ \{s\} \\
& \mid \text{loopBlk } x \leftarrow gen \ \{b\} \\
& \mid e_{acc} = \text{sumBlk } e_0 \ x \leftarrow gen \ \{s\} ; \text{ret } e
\end{aligned}$$

**Figure 9:** The Blk IL exposes the different kinds of parallelism, including data-parallel (parBlk), reduction (sumBlk), and the absence of parallelism (seqBlk).

## 5.2 Size Inference

As a reminder, AugurV2 programs express fixed-structure models. Consequently, we can bound the amount of memory an inference algorithm uses and allocate it up front. Moreover, for GPU inference, it is necessary to determine how much memory a MCMC algorithm will consume up front because we cannot dynamically allocate memory while executing GPU code. To accomplish this, the compiler first makes all sources of memory usage explicit. For instance, primitives such as vector addition that produce a result that requires allocation will be converted into a side-effecting primitive that updates an explicitly allocated location. These functional primitives made the initial lowering step from model and query into algorithm tractable and can be removed at this step. Next, the compiler performs size inference to determine how much memory to allocate. Currently, the compiler does not support standard optimizations such as destructive updates.

## 5.3 Representing Parallelism: The Blk IL

When AugurV2 is set to target the GPU, the compiler produces Cuda/C code from Low-- IL code. Here, the compiler can finally leverage the loop annotations present in the inference code. As a reminder, the loops were annotated when the compiler first generated a base MCMC update. At this point, the compiler chooses *how* to use utilize the GPU. As with

$$\begin{aligned}
\overline{p_{dist(\vec{e})}(x)} &= \overline{(x, dist(\vec{e}))} \\
\overline{\overline{fn_1} \overline{fn_2}} &= \overline{fn_2}; \overline{fn_1} \\
\overline{\text{let } x = e \text{ in } fn} &= x = e; \overline{fn}; \overline{(x, e)} \\
&\dots \\
\overline{\prod_{x \leftarrow gen} fn} &= \text{loopAtmPar } (x \leftarrow gen) \{ \overline{fn} \} \\
\overline{[fn]_{x=e}} &= \text{if}(x = e) \{ \overline{fn} \} \{ 0; \}
\end{aligned}$$

**(b)** The adjoint density function translation. The absence of complex control-flow in the Density IL simplifies the implementation.

size-inference and memory management, this step is also similar to a more traditional code-generation step. Towards this end, the compiler introduces an additional IL called the *Blk IL*. The syntax is summarized in Figure 9.

The design of the IL is informed by the SIMD parallelism provided by a GPU. For example, the construct  $\text{parBlk } lk \ x \leftarrow gen \ \{s\}$  indicates a parallel block of code, where  $gen$  copies of the statement  $s$  (in the Low-- IL) can be run in parallel according to the loop annotation  $lk$ . This corresponds to launching  $gen$  threads on the GPU. The syntax  $e_{res} = \text{sumBlk } e_0 \ x \leftarrow gen \ \{s\} ; \text{ret } e$  indicates a summation block, where the body  $s$  maps some computation across  $gen$  and returns the expression  $e$ . The result is summed with  $e_0$  as the initial value and is assigned to the expression in  $e_{res}$ . Hence, this corresponds to a GPU map-reduce. The construct  $\text{loopBlk } x \leftarrow gen \ \{b\}$  loops a block  $b$ . Thus, this corresponds to launching  $gen$  parallelized computations encoded in  $b$  in sequence. The construct  $\text{seqBlk } \{s\}$  encodes a sequential block of code consisting of a statement  $s$ . This corresponds to the absence of parallelism.

## 5.4 Parallelizing Code

To parallelize the body of a declaration in the Low-- IL, the compiler first translates it into the Blk IL. Every top-level loop we encounter in the body is converted to a parallel block with the same loop annotation. The remaining top-level statements that are not nested within a loop are generated as a sequential block. Loop blocks and summation blocks are not generated during the initial translation step, but are generated as the result of additional transformations. The current parallelization strategy is a proof-of-concept that illustrates how to reify the parallelism specific to the base MCMC updates the compiler generates and we expect that there are many opportunities for improvement here. We summarize some optimizations that we have found useful in the context of the MCMC algorithms generated by AugurV2.

**Commuting loops** As a reminder, AugurV2 compiles at runtime so it has access to the sizes of the data and parameters. It can use this information to commute IL blocks of the form

```
parBlk Par (k <- 0 until K) {
    loop Par (n <- 0 until N) {
        ...
    }
}
```

when  $K \ll N$  so that the code utilizes more GPU threads.

**Inlining** The compiler inlines primitive functions that are implemented with loops. For example, the compiler can inline the sampling of a Dirichlet distribution, which samples a vector of Gamma distributed variables and then normalizes the vector (with `normalize`).

```
sample_dirichlet(alpha) {
    loop Par (v <- 0 until V) {
        x[v] = Gamma(alpha).samp;
    }
    normalize(x);
    ret x;
}
```

Inlining expose additional sources of parallelism. For example, if we inline the sampling of the Dirichlet distribution `sample_dirichlet` above in a `ParBlk`, then the loops may be commuted.

**Conversion to summation blocks** Lastly, the compiler analyzes parallel blocks marked atomic parallel to see which should be converted to summation blocks. To illustrate this more concretely, suppose we have the following code produced by AD:<sup>6</sup>

```
parBlk AtmPar (n <- 0 until N) {
    adj_var += adj_ll * Normal(y[n], 0, var).grad3;
}
```

Parallelizing this code by launching  $N$  threads leads to high contention in updating the variable `adj_var`. Instead, the compiler estimates the contention rate as the ratio of the number of threads we are parallelizing with (*i.e.*,  $N$  in this example) compared to the number of locations the atomic additions are accessing (*i.e.*, 1 in this example). If the ratio is high as it is in this example ( $N/1$ ), then the compiler converts it to a summation block.

```
adj_var = sumBlk adj_var (n <- 0 until N) {
    t = adj_ll * Normal(y[n], 0, var).grad3;
    ret t;
}
```

---

<sup>6</sup>This code could arise from a model with density factorization  $p_{\text{Exp}}(\sigma^2) p_{\mathcal{N}}(x_n^* | 0, \sigma^2)$  where  $p_{\text{Exp}}(\sigma^2)$  is an Exponential distribution and  $p_{\mathcal{N}}(x_n^* | 0, \sigma^2)$  is a Normal distribution with variance  $\sigma^2$ . Importantly, any model where there is a higher-level parameter (*e.g.*, the variance parameter) that controls the shape of lower-level distributions (*e.g.*, the likelihood), will result in gradient code of this form.

Importantly, the compiler is invoked at runtime so the symbolic values can be resolved.

The compiler uses the following basic heuristic to optimize a block of code. First, it inlines primitive function as described above. If inlining code results in a loop being commuted or a conversion to a summation block, then it stops and returns that block of code. Otherwise, it does not inline and returns the block as is. We have not investigated other optimizations, different orderings of the optimizations, or more generally, a cost-model to guide optimization. As an example of an area for improvement, consider the following piece of code, which the AugurV2 compiler will currently fail to parallelize well. This code could result from inlining the log-likelihood computation for a Dirichlet distribution (ignoring the normalizing constant) evaluated at the vector `x[k]` for each `k` with concentration parameter `alpha`.

```
...
parBlk AtmPar (k <- 0 until K) {
    tmp_ll = 0;
    loop AtmPar (v <- 0 until V) {
        tmp_ll += (alpha[v] - 1) * log(x[k][v]);
    }
    ll += tmp_ll;
}
...
```

In this example, the compiler would map-reduce over the outer-loop, which would be inefficient if  $K \ll V$  (*e.g.*, consider a model such as LDA that has  $K$  topics and  $V$  words in the vocabulary). By inspection, a better strategy would be to map-reduce over both loops ( $K \times V$  elements) as addition is *associative*.

Once the compiler has translated the body into the Blk IL and performed the optimizations above, it will generate Cuda/C code. The Blk IL maps in a straightforward manner onto Cuda/C code. In general, such a compilation strategy will generate multiple GPU kernels for a single Low-- declaration.

## 5.5 Synthesizing a Complete MCMC Algorithm

In this step, the compiler eliminates the Kernel IL and synthesize a complete MCMC algorithm. More concretely, the compiler generates code to compute the *acceptance ratio* (AR) associated with each base MCMC update.

Recall that a base MCMC update targeting the distribution  $p(x)$  can be thought of as a MH algorithm with a proposal  $q(x \rightarrow x')$  of a specific form. To ensure that the distribution  $p(x)$  is sampled from appropriately, the proposals are taken (or *accepted*) with probability

$$\alpha(x, x') = \min \left( 1, \frac{p(x')q(x \rightarrow x')}{p(x)q(x' \rightarrow x)} \right),$$

where  $\alpha(x, x')$  is known as the AR. If the proposal is not taken, it is said to be *rejected*. Some base MCMC updates such as Gibbs updates are always accepted, *i.e.*, have AR

$\alpha(x, x') = 1$  for any  $x$  and  $x'$ . Thus, the acceptance ratio does not need to be computed for such updates. Other MCMC updates such as HMC updates require the computation of the AR. The compiler generates code that computes the AR after every base update that requires it. Because such base updates can be rejected, the compiler maintains two copies of the MCMC state space, one for the current state and one for the proposal state, and enforces the invariant that the two are equivalent after the execution of a base MCMC update. The invariant ensures that the execution of a base MCMC update always uses the most current state.

## 6. System Implementation

In this section, we summarize AugurV2’s system implementation, which includes the compiler, the user interface, and the runtime library.

### 6.1 Compiler Implementation

The AugurV2 compiler is written in Haskell. The frontend is roughly 950 lines of Haskell code, the middle-end is roughly 1860 lines, and the backend is roughly 3420 lines. The entire compiler is roughly 9000 lines, which includes syntax, pretty printing, and type checking. We found the backend to be the most tedious to write, particularly the details of memory transfer between the Python interface and Cuda/C.

### 6.2 Runtime Library

The AugurV2 runtime library is written in Cuda/C. It provides functionality for primitive functions, primitive distributions, additional MCMC library code, and vector operations. The libraries are written for both CPU and GPU inference. We believe there is room for improvement, particularly in the GPU inference libraries. For example, in the GMM example, we need to perform the same matrix operation on many small matrices in parallel. In contrast, the typical GPU use case is to perform one matrix operation on one large matrix.

The runtime representation of AugurV2 vectors are flattened. That is, AugurV2 supports vectors of vectors (*i.e.*, ragged arrays) in its surface syntax, but the eventual representation of the data will be contained in a flattened, contiguous region of memory. This enables us to use a GPU efficiently when we want to map an operation across all the data in a vector of vectors, without following a pointer-directed structure. For this reason, the runtime representation of vectors of vectors pairs a separate pointer-directed structure with a flattened contiguous array holding the actual data. The former provides random access capabilities, while the latter enables an efficient mapping operation across the data structure. The flattened representation is also beneficial for CPU inference algorithms because of the increased locality.

## 7. Evaluation

In this section, we evaluate AugurV2’s design and compare against Jags (a variant of Bugs) and Stan, systems with similar modeling languages.

### 7.1 Extensibility

As MCMC methods are improved, it is important to design systems that can be extended to incorporate the latest advances. In this part of the evaluation, we comment on the extensibility of AugurV2’s design in supporting base MCMC updates.

To support a new base update, we need to (1) add a node to the Kernel IL AST and modify the parser, (2) extend common Kernel IL operations to support the new node, and (3) implement the Cuda/C code for the base update against the AugurV2 runtime. From experience, we have found the first two items to be simple, provided that the base update uses the MCMC primitives already implemented. For instance, once we have an implementation of AD, we can easily support both reflective Slice sampling and HMC using the same AD transformation. In contrast, supporting Gibbs updates were difficult because we need to implement a separate code-generator for each conjugacy relation. Fortunately, there is a well-known list of conjugacy relations so this can be done once. The third item is between 0 lines of C code (for a Gibbs update) to 30 lines of C code (*e.g.*, an implementation of Leapfrog integration for the HMC update) depending on the complexity of the base update. We have found that it often takes on the order of days to add a new base update, where most of the time is spent understanding the statistics and implementing the C code.

### 7.2 Performance

In this part of the evaluation, we compare AugurV2 against Jags and Stan, concentrating on how design choices made in the AugurV2 system—(1) compositional MCMC inference, (2) compilation, and (3) parallelism—compare against those made in Jags and Stan. We consider three probabilistic models commonly used to assess the performance of PPLs (*e.g.*, see the Stan user manual [22]): (1) Hierarchical Logistic Regression (HLR), (2) Hierarchical Gaussian Mixture Model (HGMM), and (3) Latent Dirichlet Allocation (LDA). We ran all the experiments on an Ubuntu 14.04 desktop with a Core-i7 CPU and Nvidia Titan Black GPU.

**Models** The generative model for a HLR is summarized below. It is a model that can be used to construct classifiers.

$$\begin{aligned}\sigma^2 &\sim \text{Exponential}(\lambda) \\ b &\sim \text{Normal}(0, \sigma^2) \\ \theta_k &\sim \text{Normal}(0, \sigma^2) \\ y_n &\sim \text{Bernoulli}(\text{sigmoid}(x_n \cdot \theta_k + b))\end{aligned}$$

The generative model for a HGMM is summarized below. It is a model that clusters points on  $D$ -dimensional Eu-

clidean space.

$$\begin{aligned}\pi &\sim \text{Dirichlet}(\alpha) \\ \mu_k &\sim \text{Normal}(\mu_0, \Sigma_0) \\ \Sigma_k &\sim \text{InvWishart}(\nu, \Psi) \\ z_n &\sim \text{Categorical}(\pi) \\ y_n &\sim \text{Normal}(\mu_{z_n}, \Sigma_{z_n})\end{aligned}$$

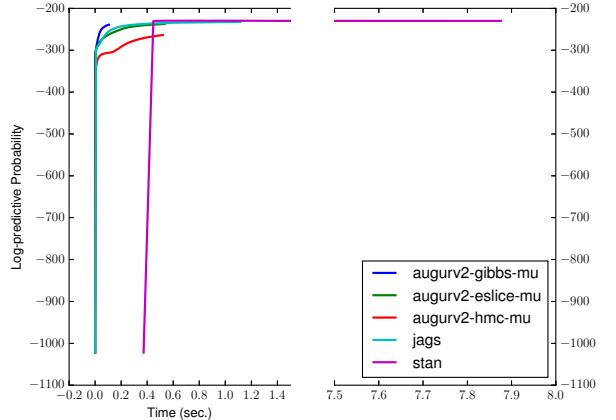
The generative model for LDA is summarized below. It is a model that can be used to infer topics from a corpus of documents.

$$\begin{aligned}\theta_d &\sim \text{Dirichlet}(\alpha) \\ \phi_k &\sim \text{Dirichlet}(\beta) \\ z_{dj} &\sim \text{Categorical}(\theta_d) \\ w_{dj} &\sim \text{Categorical}(\phi_{z_{dj}})\end{aligned}$$

**Compositional MCMC** To assess the impact of generating compositional MCMC algorithms, we use the HLR model and the HGMM model. The HLR model contains only continuous parameters. Hence, a system such as Stan, which is specifically designed for gradient-based MCMC algorithms, should perform well. The HGMM model is fully-conjugate. Hence, a system such as Jags, which is specifically designed for Gibbs sampling, should perform well.

For the HLR model, we visually verified the trace plots (*i.e.*, a plot of the values of the parameter from one sample to the next) of each system. On the German Credit dataset [13], we found that AugurV2 configured to generate a CPU HMC sampler with manually picked parameters to be roughly 25 percent slower than Stan set to use the same HMC sampling algorithm in generating 1000 samples (with no thinning). It would be interesting to investigate more the differences in AugurV2’s implementation of AD and Stan’s. Jags had the poorest performance as it defaults to adaptive rejection sampling. It takes roughly 35 seconds for Stan to compile the model (due to the extensive use of C++ templates in its implementation of AD). AugurV2 compiles almost instantaneously when generating CPU code, while it takes roughly 8 seconds to generate GPU code. The difference between CPU and GPU compile times is due to the difference in speed between Clang and Nvcc (the GPU compiler we use). Although fast compilation times are not an issue in our setting where we just need to compile the model (and query) once, this may become more of an issue in structure learning [19].

Figure 10 contains plots of the log-predictive probability versus training time for a 2D-HGMM model with 1000 synthetically-generated data points and 3 clusters. A log-predictive probability plot can be seen as a proxy for learning—as training time increases, the algorithm should be able to make better predictions. We configured AugurV2 to generate 3 different MCMC samplers corresponding to sampling the cluster locations with Elliptical Slice updates,



**Figure 10:** The log-predictive probability of a HGMM when using the samplers generated by AugurV2, Jags, and Stan. We use AugurV2 to generate 3 different MCMC inference algorithms. We set AugurV2 and Jags to draw 150 samples with no burn-in and no thinning. We set Stan to draw 100 samples with an initial tuning period of 50 samples and no thinning.

$(k, d, n)$	AugurV2	Jags	Speedup
(3, 2, 1000)	0.2	1.1	$\sim 5.5x$
(3, 2, 10000)	1.4	17.4	$\sim 12.4x$
(10, 2, 10000)	3.7	51.5	$\sim 13.9x$
(3, 10, 10000)	15.6	93.0	$\sim 5.9x$
(10, 10, 10000)	17.8	301.9	$\sim 16.9x$

**Figure 11:** Approximate timing results comparing the performance of AugurV2’s compiled Gibbs sampler versus Jags’ Gibbs sampler on a HGMM with varying clusters ( $k$ ), dimensions ( $d$ ), and data points ( $n$ ).

Gibbs updates, and HMC updates. The plot shows that every system converges to roughly the same log-predictive probability, the difference being the amount of time it takes. For instance, Jags and AugurV2’s Gibbs sampler have better computational performance because they can leverage the conjugacy relation, whereas Stan uses gradient-based MCMC.

**Compilation** Figure 11 summarizes the timing results to generate 150 samples for a HGMM on a synthetically generated dataset for a varying number of clusters, dimensions, and data points. We set AugurV2 to generate a Gibbs update for all the variables and compare against Jags’ Gibbs sampler so that both are running the same high-level inference algorithm. The difference is that Jags reifies the Bayesian network structure and performs Gibbs sampling on the graph structure, whereas AugurV2 directly generates code that performs Gibbs sampling using symbolically computed condi-

Dataset-Topics	CPU (sec.)	GPU (sec.)	Speedup
Kos-50	159	60	$\sim 2.7x$
Kos-100	265	73	$\sim 3.6x$
Kos-150	373	82	$\sim 4.6x$
Nips-50	504	161	$\sim 3.1x$
Nips-100	880	168	$\sim 5.2x$
Nips-150	1354	235	$\sim 5.8x$

**Figure 12:** Approximate timing results comparing the performance of AugurV2’s CPU Gibbs inference against GPU Gibbs inference for LDA. The Kos dataset [13] has a vocabulary size of 6906 and contains roughly 460k words. The Nips dataset [13] has a vocabulary size of 12419 and roughly 1.9 million words.

tions. The experiments show that AugurV2’s approach outperforms Jags’ approach. We also compared to the performance of Stan as well as checked the log-predictive probabilities for all three systems, but didn’t include timing results for Stan because they aren’t particularly meaningful. For instance, we applied Stan’s No-U-Turn sampler to the largest model with 10 clusters in 10 dimensions, for which it takes approximately 19 hours to draw 150 samples. Notably, Stan does not natively support discrete distributions so the user must write the model to marginalize out all discrete variables, which increases the complexity of computing gradients. This highlights the importance of being able to support compositional MCMC algorithms.

**Parallelism** Jags and Stan support parallel MCMC by running multiple copies of a chain in parallel. In contrast, AugurV2 supports parallel MCMC by parallelizing the computations *within* a single chain. As these methods are not comparable, we will focus on AugurV2’s GPU inference capabilities. For these experiments, we will use all three models. We found that GPU inference can improve scalability of inference, but it is highly model-dependent.

For example, when we fit the HLR to the German Credit dataset using AugurV2’s GPU HMC sampler, the computational performance was roughly an order of magnitude worse compared to AugurV2’s CPU HMC sampler. This can be attributed to the small dataset size (roughly 1000 points) and the low dimensionality of the parameter space (26 parameters). When we apply the GPU HMC sampler to the Adult Income dataset [13], which has roughly 50000 observations and 14 parameters, the gradients were parallelized differently due to the summation block optimization—it is more efficient to run 14 map-reduces over 50000 elements as opposed to launching 50000 threads all contending to increment 14 locations.

In contrast to a model such as a HLR, models such as HGMM and LDA have much higher dimensional spaces. Indeed, the number of latent variables scales with the number of data points. In these cases, GPU inference was much more effective. Figure 12 summarizes the (approximate)

timing results for AugurV2’s performance on LDA across a variety of datasets, comparing its CPU Gibbs performance against its GPU Gibbs performance. We also checked the log-predictive probabilities to make sure that they were roughly the same for the CPU and GPU samplers. The general trend is that the GPU provides more benefit on larger datasets, with larger vocabulary sizes, and with more topics. We have also tried this with the HGMM and found the same general trend. We could not get Jags or Stan to scale to LDA, even for the smallest dataset.

## 8. Related Work

The design of AugurV2 builds upon a rich body of prior work on PPLs. In this section, we compare AugurV2’s design with related PPLs, using the *(model, query, inference)* tuple (*i.e.*, the probabilistic modeling and inference tuple) as a guide.

To begin, we summarize the different kinds of modeling languages proposed for expressing models. At a high-level, modeling languages either focus on (1) expressing well-known probabilistic modeling abstractions such as Bayesian networks [8, 23] as in AugurV2 or (2) are embedded into general-purpose programming languages [11, 20, 21, 28, 29]. Within both categories, there are variations with how models are expressed. For example, in the first category, the Stan [8] language enables users to specify a model by both imperatively updating its (log) density as well as writing the generative process. Bugs [23] provides an imperative language for specifying the generative process. For comparison, AugurV2 provides a first-order functional language for specifying the generative process, which simplifies the analysis a compiler performs. The Factorie [15] language expresses Factor graphs, and hence, does not necessarily express generative processes. The second category of languages provide richer modeling languages.

There are also a multitude of approaches to what queries and inference algorithms are supported. AugurV2 takes a sampling approach to posterior inference as with many other systems [20, 28, 29]. However, some systems such as Stan [8] and Infer.net [16] support multiple kinds of inference strategies, including Automatic Differentiable Variational Inference (AVDI) and Expectation Propagation respectively. Hence, these systems provide a fixed number of possible inference strategies. Hakaru [18] and Psi [10] provide capabilities for exact inference via a symbolic approach to integration.

The Blaise [5] system provides a domain-specific graphical language for expressing both models and MCMC algorithms. Hence, such a system provides a more expressive language for encoding inference algorithms beyond choosing from a preselected and fixed number of strategies. As we mentioned in the introduction, the Kernel IL used by AugurV2 is closely related to the Blaise language. In particular, it should be possible to translate the Kernel IL in-

stantiated with the Density IL to a Blaise graph. We use the Kernel IL for the purposes of compilation, whereas in Blaise the language is interpreted. It would be an interesting direction of future work to see if the subset of the Blaise language related to inference can also be used for the purposes of compilation.

Edward [26] and Venture [14] also explore increasing the expressivity of different query and inference languages. Edward is built on top of TensorFlow [1], and hence, provides gradient-based inference strategies such as HMC and ADVI. Notably, the TensorFlow system efficiently supports the computation of gradients for an input computational graph (*e.g.*, the computational graph of the log-likelihood calculation for a probabilistic model). Consequently, Edward leverages the benefits of TensorFlow. It would be interesting direction for future work to see how much of TensorFlow’s infrastructure for AD can be used in AugurV2 as done in Edward. However, note that AugurV2 provides higher-level functionality than TensorFlow. For example, AugurV2 can generate non-gradient-based MCMC algorithms (*e.g.*, Gibbs samplers) as well as compose gradient-based MCMC with non-gradient-based MCMC. Venture [14] provides an inference language that is closer to a general-purpose programming language, and hence, explores the more expressive regime of the design space for inference algorithms.

In addition to exploring various points of the (*model, query, inference*) tuple space, many PPLs have also proposed interesting implementation decisions that are related to AugurV2’s. For example, the Hierarchical Bayes Compiler (HBC) [9] explores the compilation of Gibbs samplers to C code. In this work and our previous work on Augur, we support compilation of Gibbs samplers to the GPU. Swift [29] uses a compiler pass to optimize how conditional independence relationships are tracked at runtime for a modeling language that can express dynamic relationships. In contrast, AugurV2 provides a simpler modeling language that expresses static relationships, and hence, statically approximates these relationships. Bhat *et al.* [3] give a proof of correctness for a compiler that transforms a term in a first-order modeling language into a density. This language subsumes AugurV2’s (*e.g.*, AugurV2 does not provide branching constructs) and can be seen as justifying the implementation of the AugurV2 frontend, which translates a term in the modeling language into its density factorization.

## 9. Conclusion

In summary, we present a sequence of ILs and the steps of compilation that can be used to convert a description of a fixed-structure, parametric model and a query for posterior samples into a composable MCMC inference algorithm. These ILs were designed to support (1) the static analysis of models to derive facts useful for inference such as the model decomposition, (2) generating multiple kinds of inference algorithms and their composition, and (3) CPU and

GPU compilation. We show that this compilation scheme can scale automatic inference to probabilistic models with many latent variables (*e.g.*, LDA). We follow the traditional strategy of using ILs to cleanly separate out the requirements of each phase of compilation. It would be an interesting to see what aspects of AugurV2’s ILs and compilation strategy could be reused in the context of a larger infrastructure for compiling PPLs.

## Acknowledgments

We would like to thank Vikash Mansinghka for providing valuable feedback during the shepherding process. We would also like to thank our anonymous reviewers, our reviewers from the artifact evaluation committee, and Owen Arden. Finally, we would like to thank all those who contributed to version 1 of Augur: Adam C. Pocock, Joseph Tassarotti, Guy L. Steele Jr., and Stephen X. Green. The first author is supported by Oracle Labs.

## References

- [1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., KUDLUR, M., LEVENBERG, J., MONGA, R., MOORE, S., MURRAY, D. G., STEINER, B., TUCKER, P., VASUDEVAN, V., WARDEN, P., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).
- [2] BARTHOLOMEW-BIGGS, M., BROWN, S., CHRISTIANSON, B., AND DIXON, L. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics* 124, 1 (2000), 171–190.
- [3] BHAT, S., BORGSTRÖM, J., GORDON, A. D., AND RUSSO, C. Deriving Probability Density Functions from Probabilistic Functional Programs. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2013), pp. 508–522.
- [4] BISHOP, C. M., Ed. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [5] BONAWITZ, K. A. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.
- [6] BORGSTRÖM, J., GORDON, A. D., GREENBERG, M., MARGETSON, J., AND VAN GAEL, J. Measure Transformer Semantics for Bayesian Machine Learning. In *Programming Languages and Systems* (2011), pp. 77–96.
- [7] BROOKS, S., GELMAN, A., JONES, G. L., AND MENG, X.-L., Eds. *Handbook of Markov Chain Monte Carlo*, 1 ed. Chapman and Hall/CRC, 2011.
- [8] CARPENTER, B., LEE, D., BRUBAKER, M. A., RIDDELL, A., GELMAN, A., GOODRICH, B., GUO, J., HOFFMAN, M., BETANCOURT, M., AND LI, P. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* (2016), in press.

- [9] DAUME, III, H. Hierarchical Bayesian Compiler. <http://www.umiacs.umd.edu/~hal/HBC/>, 2008.
- [10] GEHR, T., MISAILOVIC, S., AND VECHEV, M. PSI: Exact Symbolic Inference for Probabilistic Programs. In *Computer Aided Verification* (2016), Springer International Publishing, pp. 62–83.
- [11] GOODMAN, N., MANSINGHKA, V., ROY, D. M., BONAWITZ, K., AND TENENBAUM, J. B. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence* (2008), AUAI, pp. 220–229.
- [12] HUANG, D., AND MORRISETT, G. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. In *Programming Languages and Systems* (2016), pp. 337–363.
- [13] LICHMAN, M. UCI Machine Learning Repository, 2013.
- [14] MANSINGHKA, V. K., SELSAM, D., AND PEROV, Y. N. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR abs/1404.0099* (2014).
- [15] MCCALLUM, A., SCHULTZ, K., AND SINGH, S. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 23* (2009), Neural Information Processing Systems Foundation, pp. 1249–1257.
- [16] MINKA, T., WINN, J., GUIVER, J., WEBSTER, S., ZAYKOV, Y., YANGEL, B., SPENGLER, A., AND BRONSKILL, J. Infer.NET 2.6, 2014. Microsoft Research Cambridge. <http://research.microsoft.com/infernet>.
- [17] MURRAY, I., ADAMS, R. P., AND MACKAY, D. J. Elliptical Slice Sampling. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010), SAIS, pp. 541–548.
- [18] NARAYANAN, P., CARETTE, J., ROMANO, W., SHAN, C.-C., AND ZINKOV, R. Probabilistic inference by program transformation in Hakaru (System Description). In *International Symposium on Functional and Logic Programming* (2016), Springer, pp. 62–79.
- [19] NEAPOLITAN, R. E. *Learning Bayesian Networks*. Prentice Hall, 2004.
- [20] NORI, A. V., HUR, C.-K., RAJAMANI, S. K., AND SAMUEL, S. R2: An efficient mcmc sampler for probabilistic programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence* (2010), AAAI, pp. 2476–2482.
- [21] PAIGE, B., AND WOOD, F. A Compilation Target for Probabilistic Programming Languages. In *Proceedings of the 31st International Conference on Machine Learning* (2014), JMLR.
- [22] STAN DEVELOPMENT TEAM. Stan Modeling Language: User’s Guide and Reference Manual. <http://mc-stan.org/documentation/>, 2015.
- [23] THOMAS, A., SPIEGELHALTER, D. J., AND GILKS, W. R. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian Statistics 4*, 9 (1992), 837–842.
- [24] TIERNEY, L. Markov Chains for Exploring Posterior Distributions. *Ann. Statist.* 22, 4 (12 1994), 1701–1728.
- [25] TORONTO, N., MCCARTHY, J., AND VAN HORN, D. Running probabilistic programs backwards. In *Programming Languages and Systems* (2015), Springer, pp. 53–79.
- [26] TRAN, D., HOFFMAN, M. D., SAUROUS, R. A., BREVDO, E., MURPHY, K., AND BLEI, D. M. Deep Probabilistic Programming. *arXiv preprint arXiv:1701.03757* (2017).
- [27] TRISTAN, J.-B., HUANG, D., TASSAROTTI, J., POCOCK, A. C., GREEN, S., AND STEELE, G. L. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems 28* (2014), Neural Information Processing Systems Foundation, pp. 2600–2608.
- [28] WOOD, F., VAN DE MEENT, J. W., AND MANSINGHKA, V. A new approach to probabilistic programming inference. In *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics* (2014), SAIS, pp. 2–46.
- [29] WU, Y., LI, L., RUSSELL, S., AND BODÍK, R. Swift: Compiled Inference for Probabilistic Programming Languages. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (2016), pp. 3637–3645.

## Adding Approximate Counters

GUY L. STEELE JR. and JEAN-BAPTISTE TRISTAN, Oracle Labs

We describe a general framework for adding the values of two approximate counters to produce a new approximate counter value whose expected estimated value is equal to the sum of the expected estimated values of the given approximate counters. (To the best of our knowledge, this is the first published description of any algorithm for adding two approximate counters.) We then work out implementation details for five different kinds of approximate counter and provide optimized pseudocode. For three of them, we present proofs that the variance of a counter value produced by adding two counter values in this way is bounded, and in fact is no worse, or not much worse, than the variance of the value of a single counter to which the same total number of increment operations have been applied. Addition of approximate counters is useful in massively parallel divide-and-conquer algorithms that use a distributed representation for large arrays of counters. We describe two machine-learning algorithms for topic modeling that use millions of integer counters, and confirm that replacing the integer counters with approximate counters is effective, speeding up a GPU-based implementation by over 65% and a CPU-based implementation by nearly 50%, as well as reducing memory requirements, without degrading their statistical effectiveness.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent programming; E.2 [Data Storage Representations]; G.3 [Probability and Statistics]: Probabilistic algorithms

Additional Key Words and Phrases: approximate counters, distributed computing, divide and conquer, multi-threading, parallel computing, statistical counters

**ACM Reference Format:**

Guy L. Steele Jr. and Jean-Baptiste Tristan. 2017. Adding approximate counters. *ACM Trans. Parallel Comput.* V, N, Article A (January YYYY), 45 pages.

DOI: 0000001.0000001

### 1. INTRODUCTION, BACKGROUND, AND RELATED WORK

We will say that a *counter* is an integer-valued variable whose value is initially 0 and to which two operations can be freely applied: *increment*, which replaces the current value  $k$  with  $k+1$ , and *read*, which returns the current value of the variable. We assume that in a concurrent environment all the operations on a given counter are observed by all threads to have been performed as if in some specific sequential order, and so each operation may be regarded as atomic. It is then easy to see that the result of any *read* operation on a given counter will be the number of *increment* operations performed on that counter prior to the *read* operation.

Morris [1978] introduced the notion of a *probabilistic counter*. His idea was to be able to use  $w$  bits to count more than  $2^w$  things by actually increasing the counter value in response to only some *increment* operations rather than all of them; in one specific case, his algorithm increases a counter value of  $k$  with probability  $2^{-k}$ , and a *read* operation that observes a counter value  $k$  returns  $2^k - 1$  as a statistical estimate of the actual number of times the *increment* operation has been performed.

---

Authors' addresses: Guy L. Steele Jr. and Jean-Baptiste Tristan, Oracle Labs, 35 Network Drive UBUR02-313, Burlington, MA 01803.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© YYYY ACM. 1539-9087/YYYY/01-ARTA \$15.00  
DOI: 0000001.0000001

Morris furthermore provided a generalization of this algorithm as well as a statistical analysis. The probabilistic decision made by the *increment* operation can rely on the output of a random (or pseudorandom) number generator, and as Morris observes, “The random number generator can be of the simplest sort and no great demands are made on its properties.” Flajolet [1985] provides a detailed statistical analysis of the Morris algorithms.

Cvetkovski [2007] describes Sampled-Log Approximate Counting, a variant that has better accuracy over most of the counting range and a lower expected number of actual advances of the counter. Csűrös [2010] provides a framework and analysis applicable to a more general class of counter representations and algorithms than those of Morris. Mitchell and Day [2011] introduce “flexible approximate counters” in which the probability of increasing the counter value may be controlled by a user-supplied table. Dice, Lev, and Moir (hereafter “DLM”) [2013] examine the use of probabilistic counters in a concurrent environment and explore alternate representations for the counter state.

Tristan *et al.* [2015] report that replacing standard counters with approximate counters allows a machine-learning algorithm (topic modeling) using LDA Gibbs (Latent Dirichlet Allocation with Gibbs sampling) on a GPU to make more efficient use of the limited memory on a GPU and therefore process larger data sets. They also report that the statistical performance of the algorithm remains essentially unchanged, and that, surprisingly, the version using approximate counters runs *faster*, despite the fact that incrementing an approximate counter requires generation of a pseudorandom number as well as other calculations. (Their Figure 6 indicates an overall speed improvement of approximately 9%.) They suggest two possible reasons for the increase in speed: (1) probabilistic incrementation performs many fewer actual writes to memory, and (2) less memory bandwidth is needed when performing bulk reads of the counters.

We have now extended this prior work by implementing distributed versions of the same algorithm (using either normal integer counters or approximate counters), on eight GPU cards with an Ethernet interconnect. The arrays of counters are replicated so that each GPU has a complete set of counters, but each GPU otherwise has just  $\frac{1}{8}$  of the total dataset to be analyzed. On each iteration, each GPU clears its counters, processes its fraction of the dataset, and then uses a standard (hypercube-like) message-exchange pattern to sum elementwise the eight arrays of counter values, in such a way that every GPU receives all sums (as for the MPI operation MPI\_ALLREDUCE [Snir *et al.* 1998, §4.11.4]); the version that uses approximate counters adds approximate counters by using the algorithms that we present in this paper. The behavior is quite similar to that of the single-GPU version reported by Tristan *et al.*: replacing integer counters with approximate counters does not affect the statistical performance of the algorithm, but allows the same hardware to process larger datasets. In one test, either 32-bit integer counters were used, or 8-bit approximate counters; using approximate counters improved the overall speed of each iteration of the distributed application by over 65%. We have also tested a distributed multiple-CPU implementation of another algorithm, organized as a stochastic cellular automaton, that addresses the same topic-modeling problem. We believe that approximate counters will be increasingly useful in future applications that perform statistical analysis of “Big Data” (whether using “machine learning” techniques or otherwise) and that large-scale distributed implementations of these applications will require the ability to add approximate counters.

Novel contributions of this paper (an expanded version of Steele and Tristan [2016]):

- (1) a notational framework that encompasses many kinds of approximate counter representation and reconciles some conflicting notation in the literature;
- (2) the presentation, using this framework, of a general algorithm for adding approximate counters so that expected estimated value of the sum is equal to the sum of

- the expected estimated values of the given approximate counters (we believe that this is the first published algorithm for adding approximate counters);
- (3) the derivation of optimized implementations of this general addition algorithm for five specific kinds of approximate counter;
  - (4) the presentation of “cookbook” versions of the algorithms (highlighted by framing boxes) that also check for overflow and therefore are suitable for transcription into actual programs;
  - (5) proofs for three kinds of counters that the variance remains bounded when counter values are added;
  - (6) speed measurements of multiple algorithms for adding approximate counters; and
  - (7) speed and quality measurements of distributed implementations of two machine-learning applications whose performance is greatly improved through the use of approximate counters and the ability to add approximate counters.

## 2. A GENERAL FRAMEWORK

Following Csűrös [2010], but generalizing his framework slightly to accommodate the alternate representations of DLM, we characterize a probabilistic counter that uses representation  $T$ , transition function  $\tau : T \rightarrow T$ , and transition probability function  $Q : T \rightarrow [0, 1]$  as a variable that can contain values of type  $T$  whose value is initially  $S_0$  (a specific value of type  $T$ ). Two operations may be freely applied to such a counter: *increment*, which with probability  $Q(s)$  replaces the current value  $s$  with  $\tau(s)$  (and with probability  $1 - Q(s)$  does not change the value of the variable), and *read*, which returns a value  $f(s)$  that may be regarded (that is, modeled) as a random variable whose expected value is the number of *increment* operations performed on that counter prior to the *read* operation. The function  $f$  is called the *unbiased estimator function*; Csűrös shows that  $f$  may be uniquely derived from  $Q$ .<sup>1</sup> We require the transition function  $\tau$  never to cycle; that is, it satisfies the property that  $\tau^i(S_0) \neq \tau^j(S_0)$  if  $i \neq j$ .<sup>2</sup> As a result, the possible states of a probabilistic counter produced by *increment* operations from the starting value  $S_0$  are in one-to-one correspondence with the natural numbers, and the only reason to choose a representation  $T$  other than the natural numbers  $\mathbb{N}$  is for “engineering purposes.” For notational convenience, we will define  $q_k = Q(\tau^k(S_0))$ ; this matches the meaning of “ $q_k$ ” as used by Csűrös. We will also define  $f_k = f(\tau^k(S_0))$ , which corresponds to “ $f(k)$ ” as used by Csűrös.

If we assume that the pseudo-function *random()* chooses a real number uniformly randomly (or uniformly pseudorandomly) from the half-open real interval  $[0, 1)$ , then the *increment* and *read* operations may be implemented as follows:

```

1: procedure increment(var  $X : T$ )
2:   if random() <  $Q(X)$  then  $X \leftarrow \tau(X)$ 

1: procedure read( $X : T$ )
2:   return  $f(X)$ 
```

That is the essence of it; all the rest is engineering and the statistical analysis behind it. We may choose  $T$  and  $\tau$  and  $Q$  so that  $\tau$  is easy to calculate, or  $Q$  is easy to calculate,

---

<sup>1</sup>Mitchell and Day [2011] take the opposite approach: their method represents the estimator function  $f$  (which they call “ $\phi$ ”) as a monotonically increasing table and uses a solver to derive the transition probability function  $Q$  (which they call “ $p$ ”), also represented as a table.

<sup>2</sup>In practice, it may be acceptable for an implementation of  $\tau$  to signal an “overflow error” if it would otherwise be compelled to repeat a counter state. Some of the pseudocode we present explicitly checks for and signals overflow errors. One way to handle such a signal is to allow the counter to “saturate” once it reaches its highest possible value; our pseudocode shows the appropriate assignment statement (if needed) for this purpose.

or  $f$  is easy to calculate, or all three. It may be desirable to choose  $Q$  so as to guarantee certain statistical properties, for example so that the variance of values returned by  $f$  can be bounded (and this can be done in various ways), and perhaps also to choose  $Q$  so that the dynamic range of the counter will be appropriate for a specific application.

In addition to the pseudofunction  $random()$  already described, which returns a real value (or floating-point approximation) in the range  $[0, 1]$ , we will also find it convenient to assume the availability of two additional pseudofunctions. First,  $randomBits(j)$  takes a nonnegative integer  $j$  and returns an integer value chosen uniformly randomly from the  $2^j$  integers in the range  $[0, 2^j]$  (which is equivalent to independently and uniformly choosing  $j$  bits at random and using them as binary digits to represent an integer value). Second,  $allZeroRandomBits(j)$  takes a nonnegative integer  $j$  and returns *true* with probability  $2^{-j}$  and *false* with probability  $1 - 2^{-j}$  (which is equivalent to independently and uniformly choosing  $j$  bits at random and testing whether they all happen to be 0). Note that  $randomBits(0)$  always returns 0 and  $allZeroRandomBits(0)$  always returns *true*.

### 3. ADDING APPROXIMATE COUNTERS

Generalizing the observation of Csűrös to our framework,  $f$  may be derived from  $Q$  as follows:

$$f_k = f(\tau^k(S_0)) = \sum_{0 \leq i < k} \frac{1}{Q(\tau^i(S_0))} = \sum_{0 \leq i < k} \frac{1}{q_i}$$

Note that  $f_0 = 0$ . Because each probability  $q_i$  lies in the range  $[0, 1]$ , we have  $\frac{1}{q_i} \geq 1$ , and therefore the values  $f_k = f(\tau^k(S_0))$  are strictly monotonic in  $k$ , and then some:  $f_k + 1 \leq f_{k+1}$ . Therefore we can uniquely define a representation-finding function  $\varphi : [0, +\infty) \rightarrow T$  that given any nonnegative real number  $v$  returns  $\tau^K(S_0)$  where  $K$  is the unique integer such that  $f_K \leq v < f_{K+1}$ . Because the function  $\varphi$  produces a “quantized” result, it is a left inverse to  $f$  but not a right inverse:  $\varphi(f(\tau^K(S_0))) = \tau^K(S_0)$  for all  $k \geq 0$ , but in general it is not true that  $f(\varphi(v)) = v$ ; the best we can say is that  $f(\varphi(v)) \leq v < f(\varphi(v))$ . A general implementation of  $\varphi$  is as follows:

```

1: procedure  $\varphi(v)$ 
2:   let  $S \leftarrow S_0$ 
3:   loop
4:     let  $S' \leftarrow \tau(S)$ 
5:     if  $v < f(S')$  then return  $S$ 
6:      $S \leftarrow S'$ 
7:   end loop

```

However, specific counter representations typically permit a much faster (non-iterative) implementation. Just as an implementation of  $\tau$  may in practice signal an overflow error, a specialized implementation of  $\varphi$  may likewise signal an overflow error if its argument is too large.

Given an implementation of  $\varphi$  for a specific sort of counter, we can “add” two counter representation values  $x$  and  $z$  so that the expected estimated value of the sum  $x \oplus z$  equals the sum of their individual expected estimated values (provided that the processes that produced  $x$  and  $z$  are statistically independent):

$$x \oplus z = \begin{cases} \tau(\varphi(f(x) + f(z))) & \text{with probability } \Delta \\ \varphi(f(x) + f(z)) & \text{with probability } (1 - \Delta) \end{cases}$$

where  $\Delta = \frac{(f(x) + f(z)) - f(\varphi(f(x) + f(z)))}{f(\tau(\varphi(f(x) + f(z)))) - f(\varphi(f(x) + f(z)))}$

The expected value of  $f(x \oplus z)$  will then be

$$\begin{aligned}
& (1 - \Delta)f(\varphi(f(x) + f(z))) + \Delta f(\tau(\varphi(f(x) + f(z)))) \\
&= f(\varphi(f(x) + f(z))) + \\
&\quad \Delta(f(\tau(\varphi(f(x) + f(z)))) - f(\varphi(f(x) + f(z)))) \\
&= f(\varphi(f(x) + f(z))) + \\
&\quad (f(x) + f(z)) - f(\varphi(f(x) + f(z))) \\
&= f(x) + f(z)
\end{aligned}$$

as desired.

We can express this addition operation as an algorithm that modifies a counter  $X$  by adding into it the estimated value of a statistically independent counter  $Z$  (we do this so that it will be similar in form to the *increment* operation):

```

1: procedure add(var  $X: T, Z: T$ )
2:   let  $v \leftarrow f(X)$ 
3:   let  $w \leftarrow f(Z)$ 
4:   let  $S \leftarrow v + w$ 
5:   let  $K \leftarrow \varphi(S)$ 
6:   let  $V \leftarrow f(K)$ 
7:   let  $W \leftarrow f(\tau(K))$ 
8:   let  $\Delta \leftarrow \frac{S-V}{W-V}$ 
9:   if  $\text{random}() < \Delta$  then
10:     $X \leftarrow \tau(K)$ 
11:   else
12:     $X \leftarrow K$ 

```

This algorithm is not terribly mysterious: it adds two counters  $X$  and  $Z$  by computing their expected values  $v$  and  $w$ , adding those values to get a sum  $S$ , then mapping that sum back to one of the two counter states whose expected values straddle  $S$ , in such a way that the expected value of the result is  $S$ . The real point is that we will use this general algorithm as a template for addition algorithms specialized to particular counter representations by inlining specific definitions of  $f$  and  $\tau$  and  $\varphi$ , and then optimizing. This strategy guarantees that appropriate expected values are maintained. However, it is necessary to provide a separate proof for each kind of counter that the variance is bounded.

#### 4. GENERAL MORRIS COUNTERS

For the general probabilistic counter defined by Morris,

$$\text{type } T \text{ is } \mathbb{N} \quad S_0 = 0 \quad \tau(x) = x + 1 \quad Q(x) = q^{-x}$$

(where  $q$  is a fixed constant equal to  $1 + \frac{1}{a}$ , where  $a$  is the parameter used by Morris); it follows that

$$f(x) = \frac{q^x - 1}{q - 1}$$

(which is equivalent to the formula  $a((1 + \frac{1}{a})^x - 1)$  given by Morris) and

$$\varphi(v) = \lfloor \log_q((q - 1)v + 1) \rfloor$$

Then *increment* and *read* operations for such probabilistic counters may be described in the following manner:

```

1: procedure increment(var  $X: \mathbb{N}$ )
2:   if  $\text{random}() < q^{-X}$  then  $X \leftarrow X + 1$ 

1: procedure read(var  $X: \mathbb{N}$ )
2:   return  $\frac{q^X - 1}{\langle q - 1 \rangle}$ 

```

We use angle brackets  $\langle \dots \rangle$  to indicate a constant ( $q - 1$  in this case) whose value can be computed at compile time.

If instead we let the counter representation type  $T$  be the set of values  $\mathbb{Z}_{2^b} = \{0, 1, 2, \dots, 2^b - 1\}$  representable in a  $b$ -bit word as unsigned binary integers, the *increment* operation may perform overflow checking:

```

1: procedure increment(var  $X: \mathbb{Z}_{2^b}$ )
2:   if  $\text{random}() < q^{-X}$  then
3:     if  $X \neq \langle 2^b - 1 \rangle$  then  $X \leftarrow X + 1$ 
4:     else overflow error

```

In some implementation contexts, when  $b$  is relatively small it may be worthwhile to tabulate the functions  $Q$  and  $f$ —that is, to preconstruct two arrays  $Q'$  and  $f'$  containing the values of  $Q(X)$  and  $f(X)$  for all  $0 \leq X < 2^b$ , but let  $Q'[2^b - 1] = 0$  so as to serve as a sentinel that will effectively enforce saturation on overflow. Such use of tabulations is certainly worth considering when  $b = 8$ , and perhaps even when  $b$  is as large as 16. Then the *increment* and *read* operations are simply:

```

1: procedure increment(var  $X: \mathbb{Z}_{2^b}$ )
2:   if  $\text{random}() < Q'[X]$  then  $X \leftarrow X + 1$ 
1: procedure read(var  $X: \mathbb{Z}_{2^b}$ )
2:   return  $f'[X]$ 

```

Counter  $Z$  may be added into counter  $X$  as follows:

```

1: procedure add(var  $X: \mathbb{N}, Z: \mathbb{N}$ )
2:   let  $v \leftarrow \frac{q^X - 1}{\langle q - 1 \rangle}$ 
3:   let  $w \leftarrow \frac{q^Z - 1}{\langle q - 1 \rangle}$ 
4:   let  $S \leftarrow v + w$ 
5:   let  $K \leftarrow \lfloor \log_q(\langle q - 1 \rangle S + 1) \rfloor$ 
6:   let  $V \leftarrow \frac{q^K - 1}{\langle q - 1 \rangle}$ 
7:   let  $W \leftarrow \frac{q^{K+1} - 1}{\langle q - 1 \rangle}$ 
8:   let  $\Delta \leftarrow \frac{S - V}{W - V}$ 
9:   if  $\text{random}() < \Delta$  then  $X \leftarrow K + 1$  else  $X \leftarrow K$ 

```

If the computation of the base- $q$  logarithm is not entirely accurate in the computation of  $K$ , it may fall just under an integer value that it should have equaled or exceeded. If so,  $K$  will be 1 smaller than it should have been. But this error is benign in this context, because then  $\Delta > 1$ , and so  $K$  will be incremented. There will be no further chance to increment  $K$  again, but that would have occurred only with probability commensurate with other floating-point errors.

However, in many practical situations, it is faster to avoid the computation of a logarithm entirely. If the implementation pretabulates the values of  $f(x)$  in an array  $f'$  (as already described above for use by the *read* operation) the array can be searched for the correct position. For this purpose it is best to make  $f'$  have length  $2^b + 2$ , with

$f'[X] = f(X)$  for  $0 \leq X \leq 2^b$ , and place a sentinel value—either  $+\infty$  or a very large finite value—in the last position at index  $2^b + 1$ . Also, note that  $W - V$  can be simplified to  $q^k$ , so it is helpful to pretabulate  $q^{-k}$  in a length- $2^b$  array  $p$ . Then for  $b$ -bit words with overflow checking, we have:

```

1: procedure add(var  $X: \mathbb{Z}_{2^b}, Z: \mathbb{Z}_{2^b}$ )
2:   let  $S \leftarrow f'[X] + f'[Z]$ 
3:   let  $K \leftarrow \max(X, Z)$ 
4:   while  $S \geq f'[K + 1]$  do  $K \leftarrow K + 1$ 
5:   if  $K \leq \langle 2^b - 1 \rangle$  then
6:     let  $V \leftarrow f'[K]$ 
7:     if  $\text{random}() < (p[K])(S - V)$  then
8:       if  $K \neq \langle 2^b - 1 \rangle$  then  $X \leftarrow K + 1$ 
9:       else overflow error:  $X \leftarrow \langle 2^b - 1 \rangle$ 
10:    else  $X \leftarrow K$ 
11:   else overflow error:  $X \leftarrow \langle 2^b - 1 \rangle$ 
```

If  $q = 1.08$ , for example, the **while** loop should iterate no more than 8 times, so this may be much faster than the calculation of a logarithm in software. (In Section 13 we confirm this general intuition by presenting a C implementation of this technique and measurements of its use within one specific computational environment.)

Note that the explicit overflow checking only ensures that the value of  $K$ , once computed, will fit in a  $b$ -bit word. It is assumed that the computation of  $S$  will use an arithmetic with sufficient range to avoid overflow. This is typically not difficult in practice: if  $b = 8$ , and  $S$  is computed using standard IEEE 754 double-precision floating-point operations, then the computation of  $S$  will never suffer overflow.

Morris asserts (without proof) that the variance in the estimated value of a counter after  $n$  *increment* operations have been performed is  $n(n - 1)/2a = \frac{q-1}{2}n(n - 1)$ . In Section 11 of this paper, our Lemma 11.1 provides an explicit proof of a generalization of this proposition, and Theorem 11.4 shows that when *add* operations are also used, the variance is bounded by  $\frac{q-1}{2}n(n - 1) + \rho$  where  $\rho = \frac{1}{-2(q^2 - 4q + 1)}$ . Note that  $\rho$  lies in the range  $[\frac{1}{6}, \frac{1}{4})$  for  $1 < q \leq 2$ , so this is a reasonably tight bound on the variance even for small  $n$ .

## 5. BINARY MORRIS COUNTERS

For the simplest sort of probabilistic counter defined by Morris, we choose  $q = 2$  (which corresponds to  $a = 1$  as used by Morris):

$$\text{type } T \text{ is } \mathbb{N} \quad S_0 = 0 \quad \tau(x) = x + 1 \quad Q(x) = 2^{-x}$$

It follows that

$$\begin{aligned} f(x) &= 2^x - 1 \\ \varphi(v) &= \lfloor \log_2(v + 1) \rfloor \end{aligned}$$

This makes the *increment* and *read* operations especially simple:

```

1: procedure increment(var  $X: \mathbb{N}$ )
2:   if  $\text{allZeroRandomBits}(X)$  then  $X \leftarrow X + 1$ 

1: procedure read(var  $X: \mathbb{N}$ )
2:   return  $2^X - 1$ 
```

Skilled programmers know various clever ways to compute  $2^X - 1$ , depending on whether the result is to be represented as an integer (perhaps by using a left-shift operation) or as a floating-point value (using a scaling operation to adjust the exponent field). For an integer result, the *read* operation might be simply:

```
1: procedure read(var X:  $\mathbb{N}$ )
2:   return  $(1 \ll X) - 1$ 
```

Note that in C, C++, the Java<sup>TM</sup> programming language, or other languages that have similar rules for the shift operator  $\ll$ , it may be important to write the literal 1 in the expression  $1 \ll X$  as, say,  $1L$  or  $1ull$  in order to ensure that the result has a specific (sufficiently large) type, such as (respectively) `long` or `unsigned long long`; alternatively, it may be appropriate to use an explicit cast, as in this example:

```
uint64_t read(uint8_t x) {
    return (((uint64_t) 1) << x) - 1;
}
```

It is especially easy to add counter  $Z$  into counter  $X$ : because  $f(k+1) = 2^{k+1} - 1 > 2(2^k - 1) = 2f(k)$ , it follows that  $V$  is always equal to  $f(\max(X, Z))$ , so  $S - V = f(\min(X, Z))$ , and therefore  $\Delta = \frac{2^{\min(X, Z)} - 1}{2^{\max(X, Z)}}$ . This leads to a very simple procedure that does not need to implement the  $\varphi$  function explicitly:

```
1: procedure add(var X:  $\mathbb{N}$ , Z:  $\mathbb{N}$ )
2:   let K  $\leftarrow \max(X, Z)$ 
3:   let L  $\leftarrow \min(X, Z)$ 
4:   if allZeroRandomBits(K - L) then
5:     if  $\neg$ allZeroRandomBits(L) then X  $\leftarrow K + 1$ 
6:     else X  $\leftarrow K$ 
7:   else X  $\leftarrow K$ 
```

With overflow checking, *increment* and *add* look like this:

```
1: procedure increment(var X:  $\mathbb{Z}_{2^b}$ )
2:   if allZeroRandomBits(X) then
3:     if  $X \neq \langle 2^b - 1 \rangle$  then X  $\leftarrow X + 1$ 
4:   else overflow error
```

```
1: procedure add(var X:  $\mathbb{Z}_{2^b}$ , Z:  $\mathbb{Z}_{2^b}$ )
2:   let K  $\leftarrow \max(X, Z)$ 
3:   let L  $\leftarrow \min(X, Z)$ 
4:   if allZeroRandomBits(K - L) then
5:     if  $\neg$ allZeroRandomBits(L) then
6:       if K  $\neq \langle 2^b - 1 \rangle$  then X  $\leftarrow K + 1$ 
7:       else overflow error: X  $\leftarrow K$ 
8:     else X  $\leftarrow K$ 
9:   else X  $\leftarrow K$ 
```

While the binary Morris approximate counter is in principle a special case of the general Morris approximate counter with  $q = 2$ , our addition algorithm is rather different from the one presented for the general case. Therefore in Section 11 we also present Theorem 11.6, a separate proof of bounded variance for binary Morris approximate counters when *add* operations are used, showing that the variance is bounded by  $\frac{n(n-1)}{2}$  (no additive constant  $\rho$  is needed).

## 6. CSÚRÖS FLOATING-POINT COUNTERS

For the general (that is, scaled) floating-point counters defined by Csúrös [2010], which are parametrized not only by a base  $q$  ( $1 < q \leq 2$ ) but also by  $M = 2^s$  for some integer  $s \geq 0$ , we have

$$\text{type } T \text{ is } \mathbb{N} \quad S_0 = 0 \quad \tau(x) = x + 1 \quad Q(x) = q^{-\lfloor x/M \rfloor}$$

For convenience, let  $\mu = \frac{M}{q-1}$ ; it follows that

$$f(x) = (\mu + (x \bmod M))q^{\lfloor x/M \rfloor} - \mu$$

$$\varphi(v) = d \cdot M + \left\lfloor \frac{v + \mu}{q^d} - \mu \right\rfloor \text{ where } d = \left\lfloor \log_q \frac{v + \mu}{\mu} \right\rfloor$$

In effect, all bits of an integer  $x$  *except* the  $s$  lower order bits are treated as a binary<sup>3</sup> exponent  $e = \lfloor x/M \rfloor$ , and the  $s$  low-order bits of  $x$  (that is,  $x \bmod M$ ) are treated as a floating-point significand with an implicit leading 1-bit (that's why you have to add  $\mu$  before multiplying by  $q^{-\lfloor x/M \rfloor}$ ); finally,  $\mu$  is subtracted so that the integer counter representation 0 will map to the estimated value 0 (and, most pleasantly, every integer representation  $x$  less than  $M$  maps to the estimated value  $x$ ). Again the *increment* operation is very simple, and *read* is reasonably simple:

```
1: procedure increment(var X:  $\mathbb{N}$ )
2:   if random() <  $q^{-\lfloor X/M \rfloor}$  then  $X \leftarrow X + 1$ 

1: procedure read(var X:  $\mathbb{N}$ )
2:   return  $(\mu + (X \bmod M))q^{\lfloor X/M \rfloor} - \mu$ 
```

And again, in some implementation contexts, if the range of values for  $X$  is relatively small, it may be worthwhile to tabulate the functions  $Q$  and  $f$  as arrays  $Q'$  and  $f'$  (but letting the last element of  $Q'$  be 0) so that the *increment* and *read* operations are simply:

```
1: procedure increment(var X:  $\mathbb{Z}_{2^b}$ )
2:   if random() <  $Q'[X]$  then  $X \leftarrow X + 1$ 

1: procedure read(var X:  $\mathbb{Z}_{2^b}$ )
2:   return  $f'[X]$ 
```

But for  $q = 2$  and  $M = 2^s$ , shifting and bitwise operations can be useful (we show a version of *increment* with overflow checking):

```
1: procedure increment(var X:  $\mathbb{Z}_{2^b}$ )
2:   if allZeroRandomBits(X  $\gg s$ ) then
3:     if  $X \neq \langle 2^b - 1 \rangle$  then  $X \leftarrow X + 1$  else overflow error

1: procedure read(var X:  $\mathbb{N}$ )
2:   return  $((M + (X \& \langle M - 1 \rangle)) \ll (X \gg s)) - M$ 
```

<sup>3</sup>Csúrös [2010] primarily assumes binary counters ( $q = 2$ ), but also briefly discusses “scaled floating-point counters,” relating them to earlier work by Stanojević [2007]. These bear the same relationship to binary Csúrös floating-point counters that general Morris counters have to binary Morris counters. We wish to add the observation that the primary motivation for requiring  $M$  to be a power of 2 is to make it easy to compute  $\lfloor x/M \rfloor$  and  $x \bmod M$  using bit shifting and masking operations. For some applications, using base-2 Csúrös floating-point counters with  $M$  an integer that is not a power of 2 may well be preferable to limiting  $M$  to be some power of 2 and then choosing a value of  $q$  other than 2. For other applications, it may be convenient to choose some  $q < 2$  but then choose  $M$  so that  $\mu$  is an integer. Our pseudocode assumes only that  $M$  is an integer unless otherwise indicated, and the proofs in Section 11 assume only that  $M$  is an integer.

In general (for any  $q$  and  $M$ ), addition goes like this:

```

1: procedure add(var  $X: \mathbb{N}$ ,  $Z: \mathbb{N}$ )
2:   let  $v \leftarrow (\mu + (X \bmod M))q^{\lfloor X/M \rfloor} - \mu$ 
3:   let  $w \leftarrow (\mu + (Z \bmod M))q^{\lfloor Z/M \rfloor} - \mu$ 
4:   let  $S \leftarrow v + w$ 
5:   let  $d \leftarrow \left\lfloor \log_q \frac{S+\mu}{\mu} \right\rfloor$ 
6:   let  $K \leftarrow d \cdot M + \left\lfloor \frac{S+\mu}{q^d} - \mu \right\rfloor$ 
7:   let  $V \leftarrow (\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu$ 
8:   let  $W \leftarrow (\mu + ((K+1) \bmod M))q^{\lfloor \frac{K+1}{M} \rfloor} - \mu$ 
9:   let  $\Delta \leftarrow \frac{S-V}{W-V}$ 
10:  if  $\text{random}() < \Delta$  then  $X \leftarrow K+1$  else  $X \leftarrow K$ 
```

Note that  $d \geq 0$ ; indeed,  $d \geq \max(\lfloor X/M \rfloor, \lfloor Z/M \rfloor)$ .

Now  $1 < q \leq 2$  and  $0 \leq r < 1$  together imply  $q^r < 2$ ; therefore  $\left\lfloor \frac{S+\mu}{2^d} - \mu \right\rfloor = \left\lfloor \frac{S+\mu}{q^{\lfloor \log_q \frac{S+\mu}{\mu} \rfloor}} - \mu \right\rfloor = \left\lfloor \frac{S+\mu}{q^{\log_q \frac{S+\mu}{\mu}-r}} - \mu \right\rfloor = \lfloor q^r \mu - \mu \rfloor \leq q^r \mu - \mu = (q^r - 1)\mu < \mu \leq M$ . Therefore  $K \bmod M = \left\lfloor \frac{S+\mu}{q^d} - \mu \right\rfloor$ , and so  $\lfloor K/M \rfloor = d$  and we have

$$\begin{aligned}
S - V &= S - ((\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu) \\
&= S - \left( \left( \mu + \left\lfloor \frac{S+\mu}{q^d} \right\rfloor - \mu \right) q^{\lfloor K/M \rfloor} - \mu \right) \\
&= S - \left( \left\lfloor \frac{S+\mu}{q^d} \right\rfloor q^d - \mu \right) \\
&= S - (((S+\mu) - ((S+\mu) \bmod q^d)) - \mu) \\
&= (S+\mu) \bmod q^d
\end{aligned}$$

Next, it's worth simplifying  $W - V$  by case analysis as follows:

(a) If  $(K \bmod M) < M - 1$ , then  $\lfloor (K+1)/M \rfloor = \lfloor K/M \rfloor$ , and

$$\begin{aligned}
W - V &= ((\mu + ((K+1) \bmod M))q^{\lfloor (K+1)/M \rfloor} - \mu) - ((\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu) \\
&= (\mu + ((K+1) \bmod M))q^{\lfloor K/M \rfloor} - (\mu + (K \bmod M))q^{\lfloor K/M \rfloor} \\
&= (((K+1) \bmod M) - (K \bmod M))q^{\lfloor K/M \rfloor} \\
&= (((K+1) - K) \bmod M)q^{\lfloor K/M \rfloor} = q^{\lfloor K/M \rfloor}
\end{aligned}$$

(b) If  $(K \bmod M) = M - 1$ , then  $\lfloor (K+1)/M \rfloor = \lfloor K/M \rfloor + 1$ :

$$\begin{aligned}
W - V &= ((\mu + ((K+1) \bmod M))q^{\lfloor (K+1)/M \rfloor} - \mu) - ((\mu + (K \bmod M))q^{\lfloor K/M \rfloor} - \mu) \\
&= (\mu + ((K+1) \bmod M))q^{\lfloor K/M \rfloor + 1} - (\mu + (K \bmod M))q^{\lfloor K/M \rfloor} \\
&= (q\mu + q((K+1) \bmod M))q^{\lfloor K/M \rfloor} - (\mu + (K \bmod M))q^{\lfloor K/M \rfloor} \\
&= (q\mu + q \cdot 0 - \mu - (M-1))q^{\lfloor K/M \rfloor} = q^{\lfloor K/M \rfloor}
\end{aligned}$$

Therefore in all cases  $W - V = q^{\lfloor K/M \rfloor} = q^d$ .

So, after some simplification, we have:

```

1: procedure add(var X: N, Z: N)
2:   let S  $\leftarrow (\mu + (X \bmod M))q^{\lfloor X/M \rfloor} +$ 
      $(\mu + (Z \bmod M))q^{\lfloor Z/M \rfloor} - \langle 2\mu \rangle$ 
3:   let d  $\leftarrow \left\lfloor \log_q \frac{S+\mu}{\mu} \right\rfloor$ 
4:   let K  $\leftarrow d \cdot M + \left\lfloor \frac{S+\mu}{q^d} - \mu \right\rfloor$ 
5:   if random() <  $\frac{S+\mu}{q^d} \bmod 1$  then X  $\leftarrow K + 1$  else X  $\leftarrow K$ 
```

However, in practice it may be faster to search an array  $f'$  as described in Section 4.

If we restrict our attention to  $q = 2$  and  $M = 2^s$ , and assume the use of a machine word  $B$  bits wide for representing estimated values (in contrast to words  $b$  bits wide that may be used for counter representation values) and an instruction to count the number of leading zeros in a  $B$ -bit word, we can recast the algorithm (with overflow checking) as follows (letting  $S' = S + \mu > 0$ ):

```

1: procedure add(var X:  $\mathbb{Z}_{2^b}$ , Z:  $\mathbb{Z}_{2^b}$ )
2:   let S'  $\leftarrow ((\mathbb{Z}_{2^B})(M + (X \& \langle M - 1 \rangle)) \ll (X \gg s)) +$ 
      $((\mathbb{Z}_{2^B})(M + (Z \& \langle M - 1 \rangle)) \ll (Z \gg s)) - M$ 
3:   let d  $\leftarrow \langle B - (s + 1) \rangle - \text{countLeadingZeros}(S')$ 
4:   let K  $\leftarrow (d \ll s) + (S' \gg d) - M$ 
5:   if K  $\leq \langle 2^b - 1 \rangle$  then
6:     if randomBits(d) <  $(S' \& ((1 \ll d) - 1))$  then
7:       if K  $\neq \langle 2^b - 1 \rangle$  then X  $\leftarrow K + 1$ 
8:       else overflow error: X  $\leftarrow \langle 2^b - 1 \rangle$ 
9:     else X  $\leftarrow K$ 
10:   else overflow error: X  $\leftarrow \langle 2^b - 1 \rangle$ 
```

Section 11 presents Theorem 11.10, a proof of bounded variance for scaled Csűrös approximate counters when *add* operations are used.

## 7. DLM PROBABILITY COUNTERS

For the probabilistic counter defined by DLM [2013], where the value in a counter is not approximately the logarithm of the number of increment operations but rather the probability that the next increment operation should change the counter:

$$\text{type } T \text{ is the closed real interval } [0, 1] \quad S_0 = 1.0 \quad \tau(x) = \frac{x}{q} \quad Q(x) = x$$

If, as before, we let  $q = 1 + \frac{1}{a}$  (thus  $a = \frac{1}{q-1}$ ), it follows that

$$f(x) = \frac{a}{x} - a$$

(which is equivalent to the formula  $a(\frac{1}{x} - 1)$  given by DLM but has one fewer operation). Then the *increment* and *read* operations may be implemented as follows:

```

1: procedure increment(var X: [0, 1])
2:   if random() < X then X  $\leftarrow \langle q^{-1} \rangle X$ 
1: procedure read(var X: [0, 1])
2:   return  $\frac{a}{X} - a$ 
```

The counter representation is not quantized to integers, only to floating-point approximations<sup>4</sup> of real numbers, so it may not be necessary in practice when implementing the  $\varphi$  function to choose randomly between two possible values, though for complete accuracy one would indeed perform the floating-point calculations to “infinite precision” and then do the final floating-point rounding in an appropriately probabilistic manner, as described by Forsythe [1959] and later by Callahan [1976] and Parker [1997, §6.4][2000]. Assuming no need to be that finicky, we pretend that  $\varphi$  is an exact inverse of  $f$ :

$$\varphi(v) = \frac{a}{v+a}$$

and so counter  $Z$  may be added into counter  $X$  as follows:

```

1: procedure add(var  $X: [0, 1]$ ,  $Z: [0, 1]$ )
2:   let  $S \leftarrow (\frac{a}{X} - a) + (\frac{a}{Z} - a)$ 
3:   let  $K \leftarrow \frac{S}{S+a}$ 
4:    $X \leftarrow K$ 
```

After simplification, this becomes:

```

1: procedure add(var  $X: [0, 1]$ ,  $Z: [0, 1]$ )
2:    $X \leftarrow \frac{1}{\frac{1}{X} + \frac{1}{Z} - 1}$ 
```

The remarkable thing about this representation is that the parameter  $a$  is not required for computing the sum of two counters, *provided*, of course, that the two counters do use the same parameter  $a$ .

While DLM probability counters may not require fewer bits for their representation than ordinary integer counters, they do share with other kinds of approximate counters the property that the *increment* operation does not always perform a write to memory.

## 8. DLM FLOATING-POINT COUNTERS

For the floating-point counters defined by DLM [2013], which are parametrized by a constant  $M = 2^s$  for some integer  $s \geq 0$  and by a second constant  $\Theta$  (which DLM calls MantissaThreshold) that is a positive even integer not greater than  $M$ , we have

$$\text{type } T \text{ is } \mathbb{N} \quad S_0 = 0 \quad Q(x) = 2^{-\lfloor x/M \rfloor}$$

$$\tau(x) = \begin{cases} \text{if } (k \bmod M) + 1 < \Theta \text{ then } k + 1 \\ \text{else } k - (k \bmod M) + M + \frac{\Theta}{2} \end{cases}$$

and it follows that

$$f(x) = (k \bmod M)2^{\lfloor k/M \rfloor}$$

In effect, all bits of an integer  $k$  *except* the  $s$  lower order bits are treated as a binary exponent  $e = \lfloor k/M \rfloor$ , and the  $s$  low-order bits of  $k$  (that is,  $k \bmod M$ ) are treated as a floating-point significand that does *not* have an implicit leading 1-bit and whose leading bit may or may not be 1 (that is, the representation is not necessarily in normalized form). As a result, there is some redundancy in the representation: two or more

---

<sup>4</sup>Although we cannot resist pointing out that fixed-point fractions may be a practical alternate representation for probabilities, with the advantage of allowing the use of a random number generator that generates random  $b$ -bit integers, which may be faster than generating random floating-point values.

counter representation values may map to the same estimated value, and  $\varphi$  is not a function but rather a relation. One similarity to the floating-point representation used by Csűrös is that every integer representation  $k$  less than  $M$  maps to the estimated value  $k$ . Again the *increment* operation is very simple, and *read* is reasonably simple:

```

1: procedure increment(var  $X$ :  $\mathbb{N}$ )
2:   if allZeroRandomBits ( $\lfloor \frac{X}{M} \rfloor$ ) then
3:     if  $(X \bmod M) + 1 < \Theta$  then  $X \leftarrow X + 1$ 
4:     else  $X \leftarrow X - (X \bmod M) + \langle M + \frac{\Theta}{2} \rangle$ 
```

```

1: procedure read(var  $X$ :  $\mathbb{N}$ )
2:   return  $(X \bmod M)2^{\lfloor X/M \rfloor}$ 
```

Again, we cleverly use shifting and bitwise operations:

```

1: procedure increment(var  $X$ :  $\mathbb{N}$ )
2:   if allZeroRandomBits ( $X \gg s$ ) then
3:     if  $(X \& \langle M - 1 \rangle) < \langle \Theta - 1 \rangle$  then  $X \leftarrow X + 1$ 
4:     else  $X \leftarrow (X \& \langle \neg(M - 1) \rangle) + \langle M + \frac{\Theta}{2} \rangle$ 
```

```

1: procedure read(var  $X$ :  $\mathbb{N}$ )
2:   return  $(X \& \langle M - 1 \rangle) \ll (X \gg s)$ 
```

DLM [2013] points out that the purpose of using a redundant representation and allowing  $\Theta$  to be smaller than  $M$  is to allow a choice of representations that are equivalent in value but differ in their probability of changing the counter during an *increment* operation; this flexibility is used to address situations in which there appears to be high contention among multiple threads for a shared counter. We will assume that this flexibility is not needed when performing an *add* operation, and therefore we are free to choose an implementation for  $\varphi$  that does not take  $\Theta$  into account:

```

1: procedure add(var  $X$ :  $\mathbb{N}$ ,  $Z$ :  $\mathbb{N}$ )
2:   let  $S \leftarrow (X \bmod M)2^{\lfloor X/M \rfloor} + (Z \bmod M)2^{\lfloor Z/M \rfloor}$ 
3:   if  $S < M$  then  $X \leftarrow S$ 
4:   else
5:     let  $d \leftarrow \lfloor \log_2(S + 1) \rfloor - \langle s + 1 \rangle$ 
6:     let  $K \leftarrow d \cdot 2^s + \lfloor \frac{S}{2^d} \rfloor$ 
7:     let  $V \leftarrow (K \bmod M)2^{\lfloor K/M \rfloor}$ 
8:     let  $W \leftarrow ((K + 1) \bmod M)2^{\lfloor (K+1)/M \rfloor}$ 
9:     let  $\Delta \leftarrow \frac{S - V}{W - V}$ 
10:    if random() <  $\Delta$  then
11:      if  $(K \bmod M) = \langle M - 1 \rangle$  then
12:         $X \leftarrow (d + 1)2^s + \langle \frac{M}{2} \rangle$ 
13:      else  $X \leftarrow K + 1$ 
14:    else  $X \leftarrow K$ 
```

After simplification and introduction of bitwise operations this is:

```

1: procedure add(var  $X$ :  $\mathbb{N}$ ,  $Z$ :  $\mathbb{N}$ )
2:   let  $S \leftarrow ((X \& \langle M - 1 \rangle) \ll (X \gg s)) +$ 
      $((Z \& \langle M - 1 \rangle) \ll (Z \gg s))$ 
3:   if  $S < M$  then  $X \leftarrow S$ 
4:   else
```

```

5:   let  $d \leftarrow \langle B - s \rangle - countLeadingZeros(S')$ 
6:   let  $K \leftarrow (d \ll s) + (S \gg d)$ 
7:   if  $randomBits(d) < (S \& ((1 \ll d) - 1))$  then
8:     if  $(K \bmod M) = \langle M - 1 \rangle$  then
9:        $X \leftarrow K + \langle \frac{M}{2} + 1 \rangle$ 
10:    else  $X \leftarrow K + 1$ 
11:   else  $X \leftarrow K$ 

```

With overflow checking, the *increment* and *add* operations look like this (we assume that one may disobey the  $\Theta$  threshold if doing so will postpone an overflow error):

```

1: procedure increment(var  $X: \mathbb{N}$ )
2:   if  $allZeroRandomBits(X \gg s)$  then
3:     if  $(X \& \langle M - 1 \rangle) < \langle \Theta - 1 \rangle$  then  $X \leftarrow X + 1$ 
4:     else if  $(X \gg s) = \langle 2^{b-s} - 1 \rangle$  then
5:       if  $X \neq \langle 2^b - 1 \rangle$  then  $X \leftarrow X + 1$ 
6:       else overflow error
7:     else  $X \leftarrow X \& \langle \neg(M - 1) \rangle + \langle M + \frac{\Theta}{2} \rangle$ 
1: procedure add(var  $X: \mathbb{N}, Z: \mathbb{N}$ )
2:   let  $S \leftarrow ((X \& \langle M - 1 \rangle) \ll (X \gg s)) +$ 
      $((Z \& \langle M - 1 \rangle) \ll (Z \gg s))$ 
3:   if  $S < M$  then  $X \leftarrow S$ 
4:   else
5:     let  $d \leftarrow \langle B - s \rangle - countLeadingZeros(S')$ 
6:     let  $K \leftarrow (d \ll s) + (S \gg d)$ 
7:     if  $K \leq \langle 2^b - 1 \rangle$  then
8:       if  $randomBits(d) < (S \& ((1 \ll d) - 1))$  then
9:         if  $K \neq \langle 2^b - 1 \rangle$  then
10:          if  $(K \bmod M) = \langle M - 1 \rangle$  then
11:             $X \leftarrow K + \langle \frac{M}{2} + 1 \rangle$ 
12:          else  $X \leftarrow K + 1$ 
13:        else overflow error:  $X \leftarrow \langle 2^b - 1 \rangle$ 
14:      else  $X \leftarrow K$ 
15:      else overflow error:  $X \leftarrow \langle 2^b - 1 \rangle$ 

```

## 9. ADDING COUNTERS OF DIFFERENT TYPES

If counter  $X$  is of type  $T_1$  (with transition function  $\tau_1$ , estimation function  $f_1$ , and representation-finding function  $\varphi_1$ ) and we wish to add into it a counter  $Z$  of type  $T_2$  (with estimation function  $f_2$ ), our general algorithm for the *add* operation accommodates this easily:

```

1: procedure add(var  $X: T_1, Z: T_2$ )
2:   let  $v \leftarrow f_1(X)$ 
3:   let  $w \leftarrow f_2(Z)$ 
4:   let  $S \leftarrow v + w$ 
5:   let  $K \leftarrow \varphi_1(S)$ 
6:   let  $V \leftarrow f_1(K)$ 
7:   let  $W \leftarrow f_1(\tau_1(K))$ 
8:   let  $\Delta \leftarrow \frac{S - V}{W - V}$ 
9:   if  $random() < \Delta$  then  $X \leftarrow \tau_1(K)$  else  $X \leftarrow K$ 

```

Whether a specific optimized version is worthwhile will depend on the two specific choices of counter representation.

It is even possible to add two counters of different types to produce a result in a third representation  $T_3$  (with transition function  $\tau_3$ , estimation function  $f_3$ , and representation-finding function  $\varphi_3$ ):

```

1: procedure add(X:  $T_1$ , Z:  $T_2$ ):  $T_3$ 
2:   let v  $\leftarrow f_1(X)$ 
3:   let w  $\leftarrow f_2(Z)$ 
4:   let S  $\leftarrow v + w$ 
5:   let K  $\leftarrow \varphi_3(S)$ 
6:   let V  $\leftarrow f_3(K)$ 
7:   let W  $\leftarrow f_3(\tau_3(K))$ 
8:   if random()  $< \frac{S-V}{W-V}$  then return  $\tau_3(K)$  else return K
```

At this time we do not know of any specific practical application for this ability.

## 10. DISCUSSION

Using approximate counters (perhaps with addition) is easy: they are a direct replacement for increment-only integer counters. The API is trivial: *read*, *increment*, and perhaps *add* operations, plus a way to initialize them to zero. The question is under what circumstances (that is, in what sort of algorithm) it is appropriate to do such a replacement. We speak to that briefly in the last paragraph of Section 16.

Approximate counters deliver only a statistical estimate of the true number of increment operations. The standard deviation of this estimate is the square root of the variance (this is the very definition of “standard deviation”). If the statistical distribution of values taken on by an approximate counter after some number  $n$  of *increment* operations were a (continuous) normal distribution, we could apply the standard “68-95-99.7 Rule” for normal distributions and expect the statistical estimate to be less than one standard deviation away from the true value about 68% of the time, to be less than two standard deviations away from the true value about 95% of the time, and to be less than three standard deviations away from the true value “nearly always” (about 99.7% of the time). However, the distribution of an approximate counter is actually a discrete distribution—which is, however, easily computed by recurrence. We have done so for eight choices of parameters  $q$  and  $M$  and for  $0 \leq n \leq 1000$ . (As examples, Fig. 1 shows the discrete distributions for the estimated value of a general Morris approximate counter (for  $q = 1.2$ ) after 3, 4, 5, 6, 7, 8, 9, and 10 *increment* operations, and Fig. 2 shows the discrete distributions for the estimated value of a general Csúrös approximate counter (for  $q = 1.2, M = 4$ ) after 6, 7, 8, 9, 10, 11, 12, and 13 *increment* operations.) For each of the 8008 distributions we computed the mean, then the standard deviation, then the fraction of probability mass in the distribution that is within one, two, or three standard deviations from the mean. The results are graphed in Fig. 3 and 4. In each chart, the  $x$  axis is the number of *increment* operations and the  $y$ -axis is the fraction of probability mass; the lowest data line corresponds to one standard deviation, the middle data line to two standard deviations, and the uppermost data line to three standard deviations. These data lines are “choppy” because of quantization effects of the discrete distributions; when the data line for one standard deviation dips down low, it is because a substantial chunk of probability mass happens to lie just outside that boundary, rather than at some great distance. Even so, that data line stays above 0.6 for all but the smallest values of  $n$ . It will be seen from this exemplary data that, at least for larger values of  $n$ , the 68-95-99.7 Rule does approximately describe these distributions despite the fact that they are discrete.

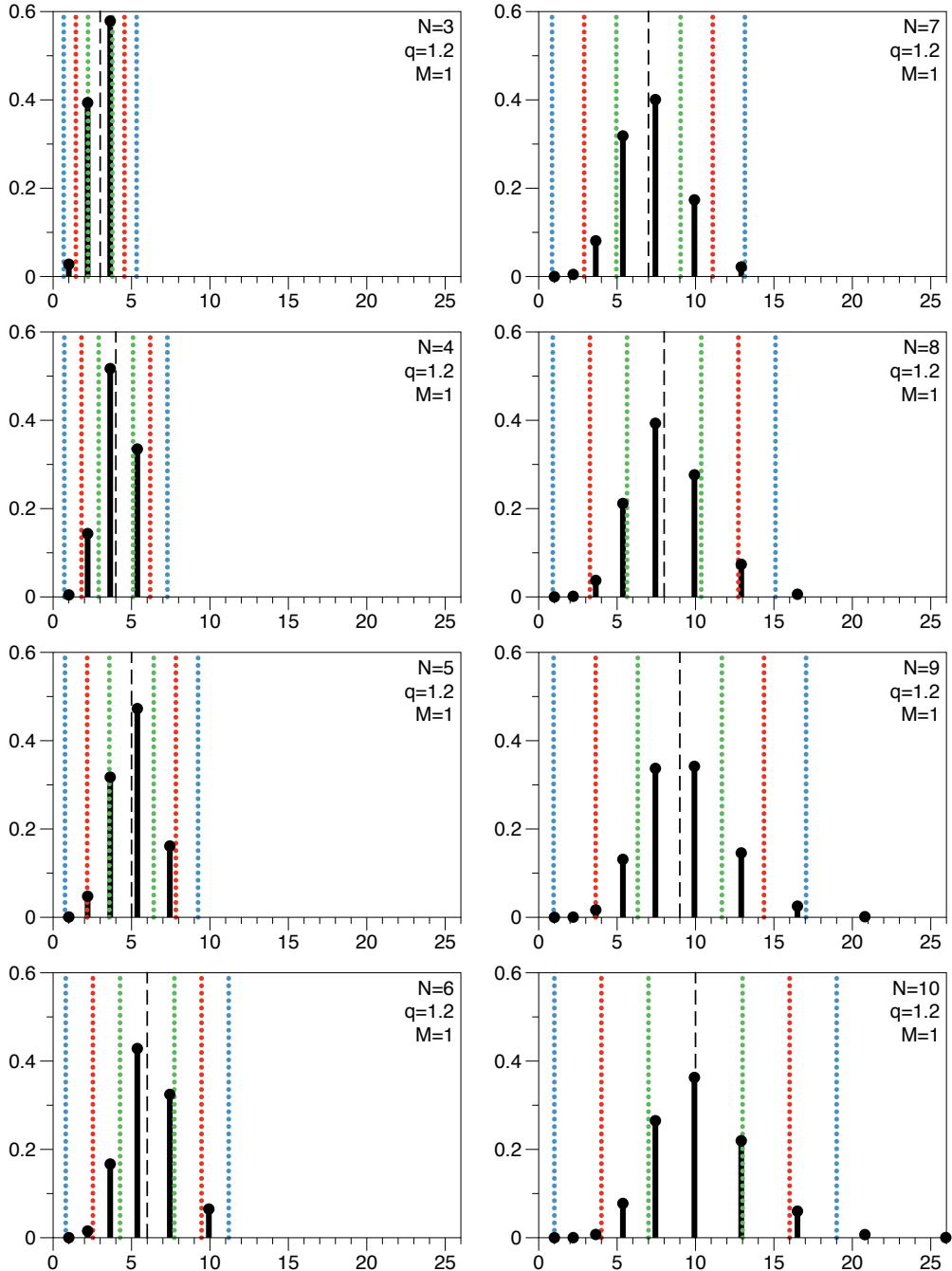


Fig. 1. The discrete statistical distributions of a general Morris approximate counter (for  $q = 1.2$ ) after  $n$  increment operations (for  $3 \leq n \leq 10$ ). The  $x$ -axis is the value of the estimator function  $f$ ; the  $y$ -axis is probability mass. The vertical dashed line indicates the mean of the distribution (equal to  $n$ ); the vertical dotted lines indicate distances of one (green), two (red), and three (blue) standard deviations from the mean.

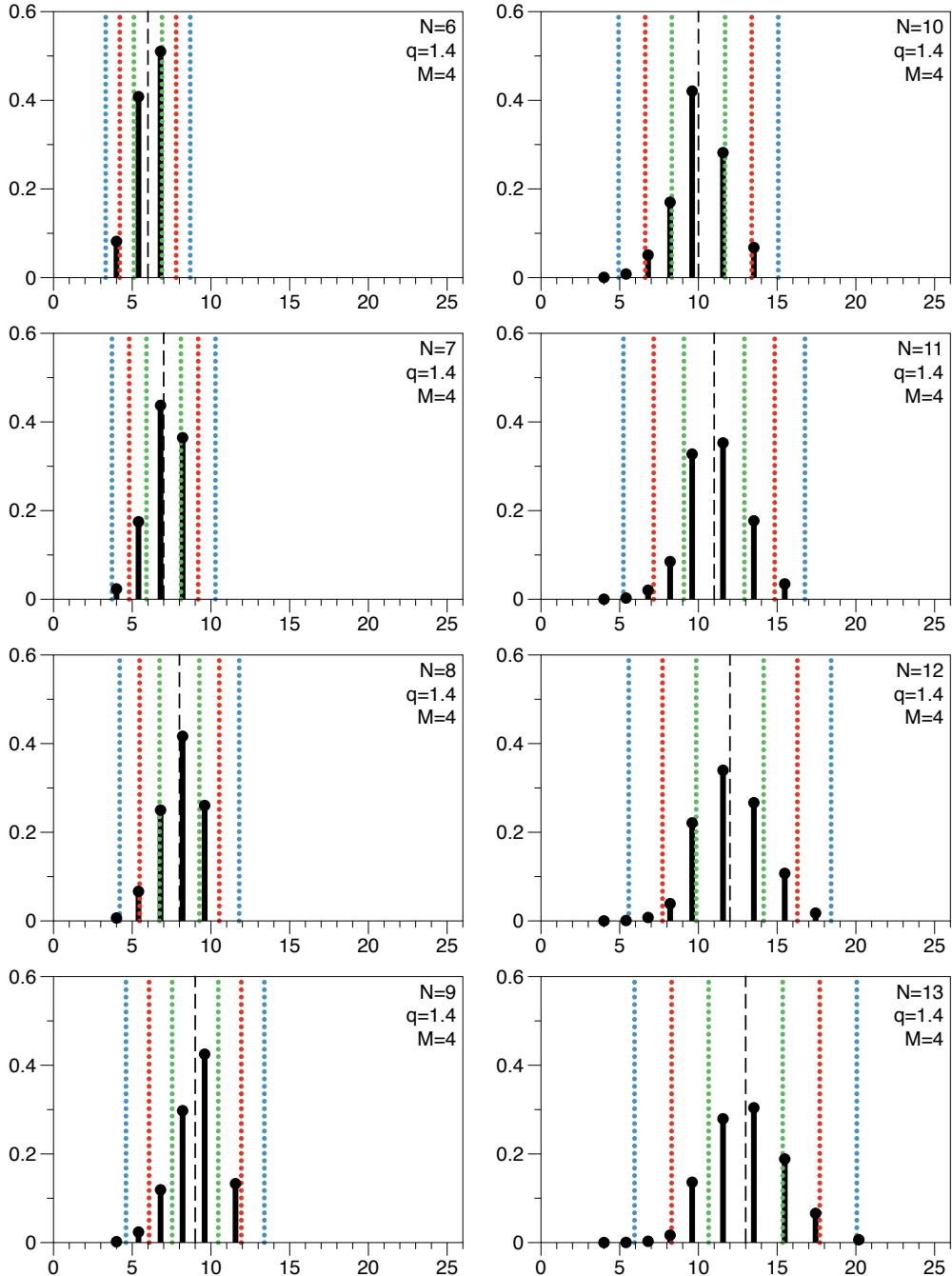


Fig. 2. The discrete statistical distributions of a general Csűrös approximate counter ( $q = 1.4$ ,  $M = 4$ ) after  $n$  increment operations (for  $6 \leq n \leq 13$ ). The  $x$ -axis is the value of the estimator function  $f$ ; the  $y$ -axis is probability mass. The vertical dashed line indicates the mean of the distribution (equal to  $n$ ); the vertical dotted lines indicate distances of one (green), two (red), and three (blue) standard deviations from the mean.

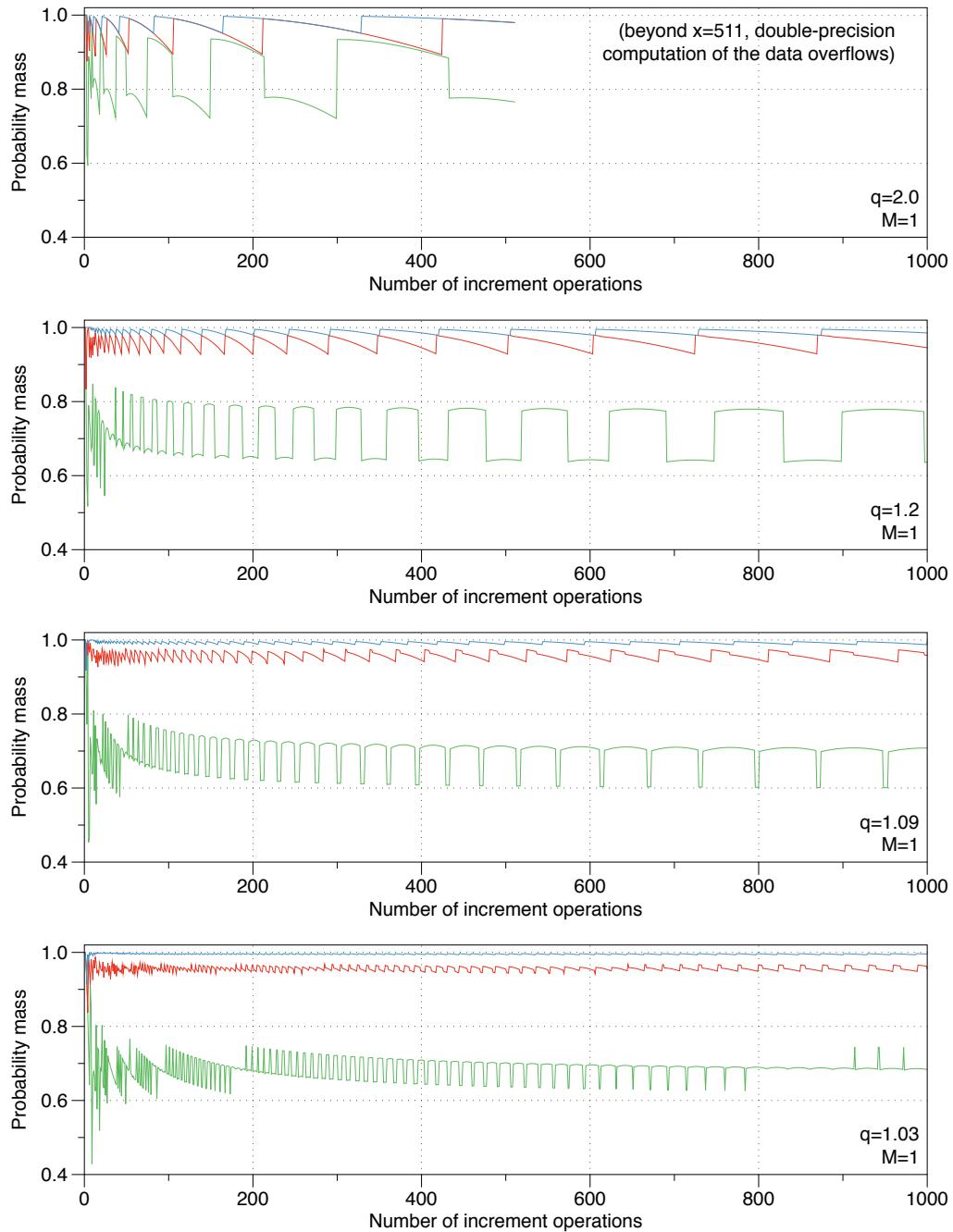


Fig. 3. The fraction of probability mass that is within one (lowest line, green), two (middle line, red), or three (highest line, blue) standard deviations from the mean after applying some number of increment operations to a zeroed Morris approximate counter (parameter  $M = 1$ ) for four different values of the parameter  $q$ . Note that in all charts the lowest value shown for the  $y$  axis is 0.4, not 0.0.

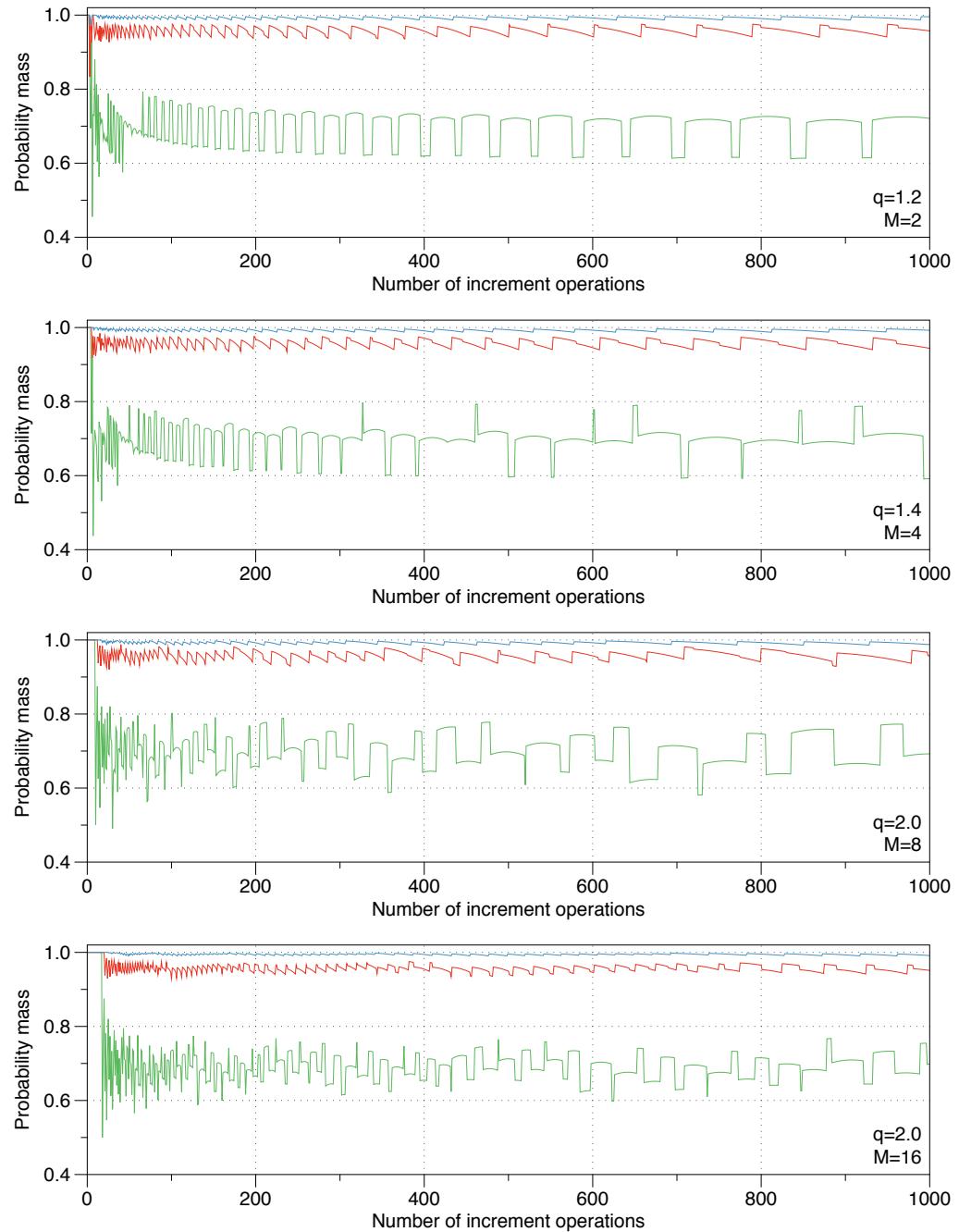


Fig. 4. The fraction of probability mass that is within one (lowest line, green), two (middle line, red), or three (highest line, blue) standard deviations from the mean after applying some number of increment operations to a zeroed Csűrös approximate counter for four different value combinations of the parameters  $q$  and  $M$ . Note that in all charts the lowest value shown for the  $y$  axis is 0.4, not 0.0.

As a specific example, by our Theorem 11.4 (which we present below in Section 11.1), the variance of a general Morris counter is no larger than  $\frac{q-1}{2}n(n-1) + \rho$  (where  $\frac{1}{6} \leq \rho < \frac{1}{4}$ ), so the standard deviation is the square root of that, and for large  $n$  this is approximately  $(\sqrt{(q-1)/2})n$ ; for  $q = 1.1$ , the standard deviation is approximately  $0.22n$ . Therefore we can expect a counter estimate to be within 22% of the true value about 2/3 of the time, to be within 44% of the true value about 95% of the time, and to be within 66% of the true value “nearly always.”

Csűrös counters and Morris counters have similar behavior when the number of increments is large (not surprising, because Morris counters are the special case of Csűrös counters with  $M = 1$ ). An advantage of Csűrös counters over Morris counters is that they are exact until the number of increments reaches  $M$ , but there is no free lunch: in exchange for achieving a variance of exactly 0 for small values of the counter, the upper bound on the variance of larger values of the counter is slightly worse. Consider an 8-bit general Csűrös counter with  $M = M_C = 8$  and  $q = q_C = 1.2$ ; the largest representable value is  $(\frac{8}{1.2-1} + (255 \bmod 8))(1.2)^{\lfloor 255/8 \rfloor} - \frac{8}{1.2-1} = (40+7)(1.2)^{31} - 40 \approx 13348.02$ . An 8-bit general Morris counter described by  $q = q_M$  whose highest representable value is the same must satisfy  $\frac{q_M^{255}-1}{q_M-1} \approx 13348.02$ ; solving for  $q_M$  gives  $q_M \approx 1.022667$ . So the coefficient of  $n(n-1)$  in the formula given by Theorem 11.10 for the upper bound on the variance of the Csűrös counter is  $\frac{1}{2\mu_C} = \frac{q_C-1}{2M_C} = \frac{0.2}{16} = 0.0125$ , but the coefficient of  $n(n-1)$  in the formula given by Theorem 11.4 for the upper bound on the variance of the Morris counter is only  $\frac{q_M-1}{2} = \frac{0.022667}{2} = 0.011333$ .

An advantage of a binary Csűrös counter over a non-binary Morris counter is that the incrementation code is cheaper. However, binary Csűrös counters support only a limited choice of maximum counter value. If your application happens to have a maximum counter value that well matches one of the possibilities for binary Csűrös counters, then that is a good choice. Otherwise, one might be better off using a general Morris counter with an appropriately calculated value for  $q$ , unless there is a specific desire to have small counter values be exact, in which case a general Csűrös counter may be a better choice, where one first chooses the necessary  $M$  and then calculates the appropriate value for  $q$ .

Table I shows the base-2 logarithm of the largest representable value of an 8-bit, 10-bit, 12-bit, or 16-bit general Csűrös counter for various values of  $q$  and  $M$ , truncated to one decimal place. (One can pack six 10-bit counters or five 12-bit counters into a 64-bit word.) This table is useful for getting an idea of what parameters might be appropriate for an approximate counter intended to replace a  $k$ -bit ordinary integer counter. For example, if an application uses 64-bit integer counters but there is reason to believe that no counter value ever exceeds  $2^{40}$ , then it is reasonable to look for values just above 40 in the table. If we wish to use an 8-bit counter, then  $M = 1$  (a general Morris counter) and  $q = 1.11$  may be about right; we can also see that no 8-bit counter works for  $M \geq 8$ , but  $M = 4$  and  $q = 1.5$  is a possibility. If we are willing to use a 16-bit counter, then a binary Csűrös counter ( $q = 2$ ) with  $M = 2048$  should easily suffice, so this may be a better choice than  $q = 1.08$  and  $M = 256$ .

## 11. PROOFS OF BOUNDED VARIANCE

In this section we prove three theorems about bounds on the variance of approximate counters: one for general Morris counters, a slightly tighter bound for the special case of binary Morris counters, and one for general Csűrös counters. In each case, the result is that the variance of a counter whose expected value is  $n$  is bounded by an expression of the form  $\xi n(n-1) + \eta$  where  $\xi$  and  $\eta$  are specific constants that depend on the counter representation parameters  $q$  and  $s$ . For each theorem the proof is inductive, following

Table I. Base-2 logarithms of largest representable values for general Csűrös counters

M	8-bit counter: $\log_2 f(2^8 - 1)$						10-bit counter: $\log_2 f(2^{10} - 1)$					
	1	2	4	8	16	32	4	8	16	32	64	128
$q = 2$	255.0	128.5	65.8	34.9	19.9	12.9	257.8	130.9	67.9	36.9	21.9	14.9
$q = 1.9$	236.2	119.2	61.2	32.6	18.9	12.5	239.0	121.5	63.3	34.7	20.9	14.5
$q = 1.8$	216.5	109.5	56.4	30.3	17.8	12.0	219.2	111.7	58.5	32.4	19.8	14.0
$q = 1.7$	195.7	99.1	51.3	27.9	16.7	11.5	198.3	101.4	53.4	29.9	18.7	13.6
$q = 1.6$	173.6	88.2	45.9	25.3	15.5	11.1	176.1	90.4	48.0	27.4	17.5	13.1
$q = 1.5$	150.1	76.6	40.3	22.6	14.3	10.6	152.6	78.8	42.4	24.7	16.3	12.6
$q = 1.4$	125.1	64.2	34.2	19.8	13.0	10.0	127.4	66.4	36.3	21.8	15.0	12.1
$q = 1.3$	98.2	51.0	27.8	16.8	11.7	9.5	100.5	53.1	29.9	18.8	13.7	11.5
$q = 1.2$	69.3	36.8	21.0	13.7	10.4	9.0	71.5	38.9	23.1	15.7	12.4	11.0
$q = 1.1$	38.3	21.8	14.0	10.6	9.1	8.5	40.4	23.9	16.1	12.6	11.1	10.5
$q = 1.09$	35.1	20.3	13.3	10.3	9.0	8.4	37.2	22.3	15.4	12.3	11.0	10.4
$q = 1.08$	31.9	18.8	12.7	10.0	8.9	8.4	34.0	20.8	14.7	12.0	10.9	10.4
$q = 1.07$	28.7	17.2	12.0	9.7	8.7	8.3	30.8	19.3	14.0	11.7	10.7	10.3
$q = 1.06$	25.4	15.7	11.3	9.4	8.6	8.3	27.5	17.8	13.3	11.5	10.6	10.3
$q = 1.05$	22.2	14.2	10.7	9.2	8.5	8.2	24.3	16.3	12.7	11.2	10.5	10.2
$q = 1.04$	19.0	12.8	10.1	8.9	8.4	8.1	21.1	14.8	12.1	10.9	10.4	10.2
$q = 1.03$	15.9	11.4	9.5	8.7	8.3	8.1	17.9	13.4	11.5	10.7	10.3	10.1
$q = 1.02$	12.9	10.1	8.9	8.4	8.2	8.0	14.9	12.1	10.9	10.4	10.2	10.0
$q = 1.01$	10.1	8.9	8.4	8.2	8.1	8.0	12.1	11.0	10.4	10.2	10.1	10.0

M	12-bit counter: $\log_2 f(2^{12} - 1)$						16-bit counter: $\log_2 f(2^{16} - 1)$					
	16	32	64	128	256	512	256	512	1024	2048	4096	8192
$q = 2$	259.9	132.9	69.9	38.9	23.9	16.9	263.9	136.9	73.9	42.9	27.9	20.9
$q = 1.9$	241.1	123.6	65.4	36.7	22.9	16.5	245.2	127.7	69.4	40.8	27.0	20.6
$q = 1.8$	221.3	113.8	60.5	34.4	21.8	16.0	225.4	117.9	64.6	38.5	25.9	20.1
$q = 1.7$	200.4	103.4	55.4	32.0	20.7	15.6	204.5	107.5	59.5	36.0	24.8	19.6
$q = 1.6$	178.2	92.5	50.1	29.4	19.5	15.1	182.3	96.5	54.1	33.4	23.6	19.1
$q = 1.5$	154.7	80.8	44.4	26.7	18.3	14.6	158.7	84.9	48.4	30.7	22.4	18.6
$q = 1.4$	129.5	68.4	38.3	23.8	17.0	14.1	133.6	72.5	42.4	27.9	21.1	18.1
$q = 1.3$	102.6	55.1	31.9	20.8	15.7	13.5	106.6	59.2	36.0	24.8	19.8	17.6
$q = 1.2$	73.6	40.9	25.1	17.7	14.4	13.0	77.7	45.0	29.2	21.7	18.5	17.0
$q = 1.1$	42.5	25.9	18.1	14.6	13.1	12.5	46.5	29.9	22.1	18.7	17.2	16.5
$q = 1.09$	39.2	24.3	17.4	14.3	13.0	12.4	43.3	28.4	21.4	18.4	17.0	16.5
$q = 1.08$	36.0	22.8	16.7	14.0	12.9	12.4	40.1	26.9	20.7	18.1	16.9	16.4
$q = 1.07$	32.8	21.3	16.0	13.7	12.8	12.3	36.8	25.3	20.1	17.8	16.8	16.4
$q = 1.06$	29.5	19.8	15.4	13.5	12.6	12.3	33.6	23.8	19.4	17.5	16.7	16.3
$q = 1.05$	26.3	18.3	14.7	13.2	12.5	12.2	30.3	22.3	18.8	17.2	16.6	16.3
$q = 1.04$	23.1	16.8	14.1	12.9	12.4	12.2	27.1	20.9	18.1	17.0	16.4	16.2
$q = 1.03$	19.9	15.4	13.5	12.7	12.3	12.1	24.0	19.5	17.6	16.7	16.3	16.2
$q = 1.02$	16.9	14.1	12.9	12.4	12.2	12.1	20.9	18.2	17.0	16.5	16.2	16.1
$q = 1.01$	14.2	13.0	12.4	12.2	12.1	12.0	18.2	17.0	16.5	16.2	16.1	16.1

Each entry is the base-2 logarithm of the largest representable value of an 8-bit, 10-bit, 12-bit, or 16-bit general Csűrös counter for the specified value of  $q$  and  $M$ , truncated to one decimal place. A counter for which this value is  $k$  is comparable in range to an ordinary  $k$ -bit integer counter. Note that a Morris counter is the same as a Csűrös counter with  $M = 1$ , and a binary counter is simply a general counter with  $q = 2$ .

the structure of the computation that produced the counter by using *increment* and *add* operations. Each of the the subsections below presents one theorem, preceded by relevant lemmas and their proofs.

### 11.1. Proof of Bounded Variance for General Morris Counters

LEMMA 11.1. [generalized from Morris [1978]] Let  $X$  be a general Morris approximate counter whose expected value is  $n$ , and let  $X'$  be the result of applying an increment operation to that counter. Let  $\kappa$  be an arbitrary constant.

Assume  $\text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) + \kappa$ ; then  $\text{Var}(f(X')) \leq \frac{q-1}{2}(n+1)n + \kappa$ . Alternatively, assume  $\text{Var}(f(X)) = \frac{q-1}{2}n(n-1) + \kappa$ ; in that case,  $\text{Var}(f(X')) = \frac{q-1}{2}(n+1)n + \kappa$ .

PROOF. Let  $\Delta = q^{-X} = \frac{q-1}{(q^{X+1}-1)-(q^X-1)} = \frac{1}{f(X+1)-f(X)}$ . Then:

$$\begin{aligned} & \text{Var}(f(X')) \\ &= \mathbb{E}[(f(X'))^2] - (\mathbb{E}[f(X')])^2 \\ &= \mathbb{E}[(1-\Delta)(f(X))^2 + \Delta(f(X+1))^2] - (n+1)^2 \\ &= \mathbb{E}[(f(X))^2 + \Delta((f(X+1))^2 - (f(X))^2)] - (n+1)^2 \\ &= \mathbb{E}[(f(X))^2] + \mathbb{E}[f(X+1) + f(X)] - (n+1)^2 \\ &= \mathbb{E}[(f(X))^2] + \mathbb{E}\left[\frac{q((q-1)f(X)+1)-1}{q-1} + f(X)\right] - (n+1)^2 \\ &= (\text{Var}(f(X)) + (\mathbb{E}[f(X)])^2) + \mathbb{E}[(q+1)f(X)+1] - (n+1)^2 \\ &= (\text{Var}(f(X)) + n^2) + (q+1)n + 1 - (n+1)^2 \\ &\leq \frac{q-1}{2}n(n-1) + \kappa + n^2 + (q+1)n + 1 - (n+1)^2 \\ &= \frac{q-1}{2}n(n+1) + \kappa \end{aligned}$$

with equality holding when the equality  $\text{Var}(f(X)) = \frac{q-1}{2}n(n-1) + \kappa$  holds.  $\square$

LEMMA 11.2. Given values  $q$  and  $S$  such that  $1 < q \leq 2$  and  $S \geq 0$ , let  $K = \lfloor \log_q((q-1)S+1) \rfloor$ , let  $V = \frac{q^K-1}{q-1}$ , let  $W = \frac{q^{K+1}-1}{q-1}$ , and let  $A = (S-V)(W-S)$ . Then  $A \leq \frac{((q-1)S+1)^2}{4q}$ .

PROOF. From the definition of the floor operation  $\lfloor \cdot \rfloor$ ,  $K = \log_q((q-1)S+1) - r$  for some  $0 \leq r < 1$ . If we fix  $q$  and  $S$  and view  $V$  as a function of  $r$ , it is easy to see that it is monotonic; if we then compute  $L = V_{r=1} = \frac{q^{-1}((q-1)S+1)-1}{q-1}$  it follows that  $L < V \leq S$ .

We will now recharacterize  $V$  as an element of the range  $(L, S]$  using a more convenient parameter  $\delta = \frac{V-L}{S-L}$  such that  $0 < \delta \leq 1$ . We write  $V$  in terms of  $S$ ,  $q$ , and  $\delta$  as  $V = L + \delta(S-L) = q^{-1}(\delta((q-1)S+1) + S - 1)$  and  $W = \frac{q((q-1)V+1)-1}{q-1} = \delta((q-1)S+1) + S$ . Then we have  $A = (S-V)(W-S) = \delta(1-\delta)q^{-1}((q-1)S+1)^2$ .

Regard  $A$  as a function of  $\delta$  and ask what value of  $\delta$  maximizes it; the answer, easily derived by solving  $\frac{d}{d\delta}(\delta(1-\delta)) = 0$ , is  $\delta = \frac{1}{2}$ , and the second derivative there is negative, so the maximum possible value for  $A$  is  $\frac{1}{2}(1 - \frac{1}{2})q^{-1}((q-1)S+1)^2$ , and therefore for any value of  $\delta$  in  $(0, 1]$  we have  $A \leq \frac{((q-1)S+1)^2}{4q}$ .  $\square$

LEMMA 11.3. Let  $X$  be a general Morris approximate counter with parameter  $q$  ( $1 < q \leq 2$ ) whose expected value is  $n$ , let  $Z$  be a general Morris approximate counter with parameter  $q$  whose expected value is  $m$ , and assume that  $X$  and  $Z$  are statistically independent. Let  $\rho = \frac{1}{-2(q^2-4q+1)}$ , and also assume that  $\text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$  and that  $\text{Var}(f(Z)) \leq \frac{q-1}{2}m(m-1) + \rho$ ; then  $\text{Var}(f(X \oplus Z)) \leq \frac{q-1}{2}(n+m)((n+m)-1) + \rho$ .

PROOF. If  $Z = 0$ , then  $m = \mathbb{E}[f(Z)] = 0$ ,  $w = 0$ ,  $S = v$ ,  $V = v$ , and  $\Delta = 0$ ; therefore  $f(X \oplus Z) = f(X)$  exactly, and it follows that  $\text{Var}(f(X \oplus Z)) = \text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) \leq \frac{q-1}{2}(n+m)(n+m-1) + \rho$ , as desired. Similarly, if  $X = 0$ , then

$n = \mathbb{E}[f(Z)] = 0, v = 0, S = w, V = w$ , and  $(1 - \Delta) = 0$ ; therefore  $f(X \oplus Z) = f(Z)$  exactly, so  $\text{Var}(f(X \oplus Z)) = \text{Var}(f(Z)) \leq \frac{q-1}{2}m(m-1) \leq \frac{q-1}{2}(n+m)(n+m-1) + \rho$ .

Now suppose  $X > 0$  and  $Z > 0$  (therefore  $n = \mathbb{E}[f(X)] \geq 1$  and  $m = \mathbb{E}[f(Z)] \geq 1$ ):

$$\begin{aligned}
& \text{Var}(f(X \oplus Z)) \\
&= \mathbb{E}\left[\left(f(X \oplus Z)\right)^2\right] - (\mathbb{E}[f(X \oplus Z)])^2 \\
&= \mathbb{E}\left[(1 - \Delta)V^2 + \Delta W^2\right] - \mathbb{E}[S^2] \\
&= \mathbb{E}\left[\frac{W-S}{W-V}V^2 + \frac{S-V}{W-V}W^2 - S^2\right] \\
&= \mathbb{E}[(S - V)(W - S)] \\
&\leq \mathbb{E}\left[\frac{((q-1)S+1)^2}{4q}\right] && [\text{by Lemma 11.2}] \\
&= \mathbb{E}\left[\frac{(q-1)^2S^2+2(q-1)S+1}{4q}\right] \\
&= \mathbb{E}\left[\frac{(q-1)^2}{4q}(f(X) + f(Z))^2 + \frac{q-1}{2q}(f(X) + f(Z)) + \frac{1}{4q}\right] \\
&= \frac{(q-1)^2}{4q}\left(\mathbb{E}\left[(f(X))^2\right] + \mathbb{E}[2f(X)f(Z)] + \mathbb{E}\left[(f(Z))^2\right]\right) + \frac{q-1}{2q}\mathbb{E}[f(X) + f(Z)] + \frac{1}{4q} \\
&= \frac{(q-1)^2}{4q}(\text{Var}(f(X)) + (\mathbb{E}[f(X)])^2 + \mathbb{E}[2f(X)f(Z)] + \text{Var}(f(Z)) + (\mathbb{E}[f(Z)])^2) \\
&\quad + \frac{q-1}{2q}(\mathbb{E}[f(X) + f(Z)]) + \frac{1}{4q} \\
&= \frac{(q-1)^2}{4q}(\text{Var}(f(X)) + n^2 + 2nm + \text{Var}(f(Z)) + m^2) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
\\
&= \frac{(q-1)^2}{4}(\text{Var}(f(X)) + \text{Var}(f(Z)) + (n+m)^2) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
&\leq \frac{(q-1)^2}{4q}\left(\frac{q-1}{2}n(n-1) + \rho + \frac{q-1}{2}m(m-1) + \rho + (n+m)^2\right) + \frac{q-1}{2q}(n+m) + \frac{1}{4q} \\
&= \frac{(q-1)(q^2-1)}{8q}(n^2 + m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \frac{(q-1)^2}{2q}\rho + \frac{1}{4q} \\
&= \frac{(q-1)(q^2-1)}{8q}(n^2 + m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \rho
\end{aligned}$$

We wish to show that this last quantity is less than or equal to  $\frac{q-1}{2}(n+m)(n+m-1) + \rho$ ; we do so by showing that  $\frac{8q}{q-1}$  times their difference is nonpositive (note that  $\frac{8q}{q-1} > 0$ ).

$$\begin{aligned}
& \frac{8q}{q-1}\left(\frac{(q-1)(q^2-1)}{8q}(n^2 + m^2) + \frac{(q-1)^2}{4q}(2nm) - \frac{(q-1)(q+1)(q-3)}{8q}(n+m) + \rho \right. \\
& \quad \left. - \left(\frac{q-1}{2}(n+m)(n+m-1) + \rho\right)\right) \\
&= (q^2 - 4q - 1)n^2 - (q^2 - 6q - 3)n - 2(q + 1)nm + \\
&\quad (q^2 - 4q - 1)m^2 - (q^2 - 6q - 3)m - 2(q + 1)nm
\end{aligned}$$

We now need only show that

$$(q^2 - 4q - 1)n^2 - (q^2 - 6q - 3)n - 2(q + 1)nm \leq 0$$

and

$$(q^2 - 4q - 1)m^2 - (q^2 - 6q - 3)m - 2(q + 1)nm \leq 0$$

The two proofs are identical in form; here we present just one of the proofs. Because  $2(q+1)n > 0$  and  $m \geq 1$ ,

$$\begin{aligned} & (q^2 - 4q - 1)n^2 - (q^2 - 6q - 3)n - 2(q+1)nm \\ & \leq (q^2 - 4q - 1)n^2 - (q^2 - 6q - 3)n - (2q+2)n \\ & = (q^2 - 4q - 1)n^2 - (q^2 - 4q - 1)n \\ & = (q^2 - 4q - 1)n(n-1) \leq 0 \end{aligned}$$

because  $n(n-1) \geq 0$  for all  $n \geq 1$  and  $q^2 - 4q - 1 < 0$  for all  $1 < q \leq 2$  (the roots of  $q^2 - 4q - 1 = 0$  are  $2 - \sqrt{5} \approx -0.236\dots$  and  $2 + \sqrt{5} \approx 4.236\dots$ ).  $\square$

**THEOREM 11.4.** *Let  $1 < q \leq 2$ , let  $\rho = \frac{1}{-2(q^2 - 4q + 1)}$ , and let  $X$  be a general Morris approximate counter with parameter  $q$  whose expected value is  $n$ . Then  $\text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$ .*

**PROOF.** We proceed by structural induction on the computation that created  $X$ .

Basis case: Suppose that  $X$  is a newly created counter. Then its value is definitely 0, and therefore its expected value  $n$  is 0 and its variance is 0, and so  $\text{Var}(f(X)) = 0 = \frac{q(0-1)}{2} \leq \frac{n(n-1)}{2} + \rho$ .

Inductive case 1 (incrementation): Suppose that  $X$  was produced by incrementing a counter  $Y$  having expected value  $n-1$ . By our inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{(n-1)(n-2)}{2} + \rho$ , and therefore by Lemma 11.1 with  $\kappa = \rho$  we have  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2} + \rho$ .

Inductive case 2 (addition): Suppose that  $X$  was produced by adding a counter  $Y$  with expected value  $y$  and a statistically independent counter  $Z$  with expected value  $z$ , such that  $y+z = n$ . By inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{q-1}{2}y(y-1) + \rho$  and  $\text{Var}(f(Z)) \leq \frac{q-1}{2}z(z-1) + \rho$ , and therefore by Lemma 11.3 we have  $\text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$ .

Therefore in all cases  $\text{Var}(f(X)) \leq \frac{q-1}{2}n(n-1) + \rho$ .  $\square$

## 11.2. Proof of Bounded Variance for Binary Morris Counters

**LEMMA 11.5.** *Let  $X$  be a binary Morris approximate counter whose expected value is  $n$ , and  $Z$  be a statistically independent binary Morris approximate counter whose expected value is  $m$ . Without loss of generality assume  $m \leq n$ . Assume also that  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2}$ ; then we have  $\text{Var}(f(X \oplus Z)) \leq \frac{(n+m)((n+m)-1)}{2}$ . As an alternative, assume also that  $m \leq 1$  and  $\text{Var}(f(X)) = \frac{n(n-1)}{2}$ ; then  $\text{Var}(f(X \oplus Z)) = \frac{(n+m)((n+m)-1)}{2}$ .*

**PROOF.** Let  $\Delta = \frac{f(Z)}{f(X+1) - f(X)}$ . Then:

$$\begin{aligned}
& \text{Var}(f(X \oplus Z)) \\
&= \mathbb{E}[(f(X \oplus Z))^2] - (\mathbb{E}[f(X \oplus Z)])^2 \\
&= \mathbb{E}[(1 - \Delta)(f(X))^2 + \Delta(f(X + 1))^2] - (n + m)^2 \\
&= \mathbb{E}[(f(X))^2 + \Delta((f(X + 1))^2 - (f(X))^2)] - (n + m)^2 \\
&= \mathbb{E}[(f(X))^2] + \mathbb{E}[f(Z)(f(X + 1) + f(X))] - (n + m)^2 \\
&= (\text{Var}(f(X)) + (\mathbb{E}[f(X)])^2) + \mathbb{E}[f(Z)(3f(X) + 1)] - (n + m)^2 \\
&\leq \frac{n(n-1)}{2} + n^2 + \mathbb{E}[f(Z)(3f(X) + 1)] - (n + m)^2 \\
&= \frac{n(n-1)}{2} + n^2 + m(3n + 1) - (n + m)^2 \\
&= \frac{n^2}{2} + nm - m^2 - \frac{n}{2} + m \\
&\leq \frac{n^2}{2} + nm - m^2 - \frac{n}{2} + m + \frac{3}{2}(m^2 - m) \\
&= \frac{(n+m)((n+m)-1)}{2}
\end{aligned}$$

(because  $\frac{3}{2}(m^2 - m) \geq 0$  for all integer  $m \geq 0$ , with equality holding when  $m \leq 1$  and  $\text{Var}(f(X)) = \frac{n(n-1)}{2}$  (because we have  $\frac{3}{2}(m^2 - m) = 0$  when  $m = 0$  or  $m = 1$ ).  $\square$

**THEOREM 11.6.** *Let  $X$  be a binary Morris approximate counter whose expected value is  $n$ . Then  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2}$ .*

**PROOF.** We proceed by structural induction on the computation that created  $X$ .

Basis case: Suppose that  $X$  a newly created counter. Then its value is definitely 0, and therefore its expected value is 0 and its variance is 0, and so  $\text{Var}(f(X)) = 0 = \frac{0(0-1)}{2} = \frac{n(n-1)}{2}$ .

Inductive case 1 (incrementation): Let us suppose that  $X$  was produced by incrementing counter  $Y$  having expected value  $n - 1$ . By our inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{(n-1)(n-2)}{2}$ , and so by Lemma 11.1 with  $\kappa = 0$  we have  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2}$ .

Inductive case 2 (addition): Suppose that  $X$  was produced by adding a counter  $Y$  with expected value  $y$  and a statistically independent counter  $Z$  with expected value  $z$ , such that  $z \leq y$  and  $y + z = n$ . By inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{y(y-1)}{2}$ , and therefore by Lemma 11.5 we have  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2}$ .

Therefore in all cases  $\text{Var}(f(X)) \leq \frac{n(n-1)}{2}$ .  $\square$

### 11.3. Proof of Bounded Variance for General Csűrös Counters

**LEMMA 11.7.** *Given real  $S \geq 0$ , real  $1 < q \leq 2$ , and integer  $M \geq 1$ , let  $\mu = \frac{M}{q-1}$ ,  $d = \left\lfloor \log_q \frac{S+\mu}{\mu} \right\rfloor$ ,  $\mathcal{V} = M \frac{q^d - 1}{q-1}$ ,  $\mathcal{W} = M \frac{q^{d+1} - 1}{q-1}$ ,  $\omega = \frac{\mathcal{W} - \mathcal{V}}{M}$ ,  $j = \left\lfloor \frac{S-\mathcal{V}}{\omega} \right\rfloor$ ,  $V = \mathcal{V} + j\omega$ ,  $W = V + \omega$ , and  $A = (S - V)(W - S)$ . Then  $A \leq \frac{((q-1)S+M)^2}{4M(M+(q-1))} = \frac{(S+\mu)^2}{4\mu(\mu+1)}$ .*

PROOF. The interval  $[\mathcal{V}, \mathcal{W}]$  is divided into  $M$  subintervals of width  $\omega$ ; subinterval  $j$  is the one that contains  $S$ . (If  $S$  falls right on the boundary between two subintervals, we can regard it as being in either subinterval; then either  $S = V$  or  $S = W$ , and so  $(S - V)(W - S) = 0$ .)

If  $S$  is in subinterval  $j$  ( $0 \leq j < M$ ), then the smallest value  $\mathcal{V}_{\min}$  that  $\mathcal{V}$  could have (holding  $q$ ,  $M$ , and  $j$  fixed while letting  $S$  vary) occurs when  $S$  is at the upper end of subinterval  $j$ , and similarly  $\mathcal{W}_{\min} = q\mathcal{V}_{\min} + M$  and  $\omega_{\min} = \frac{(q-1)\mathcal{V}_{\min} + M}{M}$ . But where is  $\mathcal{V}_{\min}$  with respect to  $S$ ? If  $S$  is at the upper end of subinterval  $j$ , then  $S - \mathcal{V}_{\min} = (j+1)\omega_{\min}$  and  $\mathcal{W}_{\min} - S = (q\mathcal{V}_{\min} + M) - S = (M - (j+1))\omega_{\min}$  similarly. (Note that  $\mathcal{W}$  and  $\omega$  are minimized exactly when  $\mathcal{V}$  is minimized, which is what justifies using the “min” subscripts on  $\mathcal{W}_{\min}$  and  $\omega_{\min}$ .) Eliminating  $\omega_{\min}$  gives us  $(M - (j+1))(S - \mathcal{V}_{\min}) = (j+1)((q\mathcal{V}_{\min} + M) - S)$ . Solving this for  $\mathcal{V}_{\min}$  gives  $\mathcal{V}_{\min} = \frac{M(S-(j+1))}{M+(q-1)(j+1)} = \frac{\mu(S-(j+1))}{\mu+(j+1)}$ .

Now  $V_{\min}$ , the lowest possible value for  $V$  (which is the lower end of the subinterval containing  $S$ ), is  $j$  subinterval-widths above  $\mathcal{V}_{\min}$  and so  $V_{\min} = (\mathcal{V}_{\min} + j\omega_{\min}) = \frac{((q-1)jS + M(S-1))}{M+(q-1)(j+1)} = \frac{(jS + \mu(S-1))}{\mu+(j+1)}$ . (Note that  $V_{\min}$  corresponds to  $L$  as used in the proof of Lemma 11.2.) We characterize  $V$  using a parameter  $\delta$  as

$$V = ((1 - \delta)V_{\min} + \delta S) = \frac{\delta(M + (q-1)S) + (q-1)jS + M(S-1)}{M + (q-1)(j+1)} = \frac{\delta(\mu + S) + jS + \mu(S-1)}{\mu + (j+1)}$$

Now we have a formula for  $V$  in terms of  $S$  and  $\delta$  (and fixed  $q$ ,  $M$ , and  $j$ ); in exchange for introducing the parameter  $\delta$ , we have gotten rid of those pesky floor functions.

Once  $V$  has been defined in this way and regarded as the actual lower end of subinterval  $j$ , we can calculate a corresponding  $\mathcal{V}_V$  that is exactly  $j$  subinterval-widths below  $V$  by solving  $(V - \mathcal{V}_V)(M - j) = ((q\mathcal{V}_V + M) - V)j$  to get the result

$$\mathcal{V}_V = \frac{M(\delta(M + (q-1)S) - (q-1)j^2 + j(q-1)(S-1) + MS - (j+1)M)}{(M + (q-1)j)(M + (q-1)(j+1))} = \frac{\mu(\delta(\mu + S) - j^2 + j(S-1) + \mu S - (j+1)\mu)}{(\mu + j)(\mu + (j+1))}$$

The width of each subinterval is  $\omega_V = \frac{(q-1)\mathcal{V}_V + M}{M}$  and therefore we have  $W = V + \omega_V = S + \delta \frac{M + (q-1)S}{M + (q-1)j} = S + \delta \frac{\mu + S}{\mu + j}$ .

Then  $(S - V)(W - S) = \delta(1 - \delta) \frac{((q-1)S + M)^2}{(M + (q-1)j)(M + (q-1)(j+1))}$ . We now have  $(S - V)(W - S)$  in terms of  $S$  and  $\delta$  and  $q$  and  $M$  and  $j$ . Now switch gears: hold  $S$  and  $q$  and  $M$  fixed, and allow  $\delta$  and  $j$  to vary. Then  $(S - V)(W - S)$  is maximal when  $\delta = \frac{1}{2}$  and  $(\mu + j)(\mu + (j+1))$  is minimal. Now  $(2\mu + 1)$  is positive for  $1 < q \leq 2$  and  $M \geq 1$ ; therefore for  $j \geq 0$ ,  $(\mu + j)(\mu + (j+1)) = j^2 + (2\mu + 1)j + \mu(\mu + 1)$  is minimal when  $j = 0$ , and so  $A = (S - V)(W - S) \leq \frac{((q-1)S + M)^2}{4M(M + (q-1))} = \frac{(S + \mu)^2}{4\mu(\mu + 1)}$ .  $\square$

**LEMMA 11.8.** *Let  $X$  be a Csűrös approximate counter with integer parameter  $M$  ( $M \geq 1$ ) and real parameter  $q$  ( $1 < q \leq 2$ ) whose expected value is  $n$ , let  $Z$  be a Csűrös approximate counter with the same parameters  $M$  and  $q$  whose expected value is  $m$ , and assume that  $X$  and  $Z$  are statistically independent. Let  $\mu = \frac{M}{q-1}$  and  $\rho = \frac{\mu^2}{4\mu^2 + 4\mu - 2}$ , and assume that  $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$  and that  $\text{Var}(f(Z)) \leq \frac{1}{2\mu}m(m-1) + \rho$ ; then  $\text{Var}(f(X \oplus Z)) \leq \frac{1}{2\mu}(n+m)((n+m)-1) + \rho$ .*

PROOF. As in the proof of Lemma 11.3, if  $Z = 0$ , then  $m = \mathbb{E}[f(Z)] = 0$ ,  $w = 0$ ,  $S = v$ ,  $V = v$ , and  $\Delta = 0$ ; therefore  $f(X \oplus Z) = f(X)$ , so  $\text{Var}(f(X \oplus Z)) = \text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) \leq \frac{1}{2\mu}(n+m)(n+m-1) + \rho$ , as desired. In the same way, if  $X = 0$ , then  $n = \mathbb{E}[f(Z)] = 0$ ,  $v = 0$ ,  $S = w$ ,  $V = w$ , and  $(1 - \Delta) = 0$ ; therefore  $f(X \oplus Z) = f(Z)$ , so  $\text{Var}(f(X \oplus Z)) = \text{Var}(f(Z)) \leq \frac{1}{2\mu}m(m-1) \leq \frac{1}{2\mu}(n+m)(n+m-1) + \rho$ .

Now suppose  $X > 0$  and  $Z > 0$  (therefore  $n = \mathbb{E}[f(X)] \geq 1$  and  $m = \mathbb{E}[f(Z)] \geq 1$ ):

$$\begin{aligned}
& \text{Var}(f(X \oplus Z)) \\
&= \mathbb{E}[(S - V)(W - S)] && [\text{as in proof of Lemma 11.3}] \\
&\leq \mathbb{E}\left[\frac{(S+\mu)^2}{4\mu(\mu+1)}\right] && [\text{by Lemma 11.7}] \\
&= \mathbb{E}\left[\frac{S^2+2\mu S+\mu^2}{4\mu(\mu+1)}\right] \\
&= \frac{1}{4\mu(\mu+1)} \left( \mathbb{E}\left[(f(X))^2\right] + \mathbb{E}[2f(X)f(Z)] + \mathbb{E}\left[(f(Z))^2\right] \right) \\
&\quad + \frac{1}{2(\mu+1)} \mathbb{E}[f(X) + f(Z)] + \frac{\mu}{4(\mu+1)} \\
&= \frac{1}{4\mu(\mu+1)} (\text{Var}(f(X)) + (\mathbb{E}[f(X)])^2 + \mathbb{E}[2f(X)f(Z)] + \text{Var}(f(Z)) + (\mathbb{E}[f(Z)])^2) \\
&\quad + \frac{1}{2(\mu+1)} (\mathbb{E}[f(X) + f(Z)]) + \frac{\mu}{4(\mu+1)} \\
&= \frac{1}{4\mu(\mu+1)} (\text{Var}(f(X)) + n^2 + 2nm + \text{Var}(f(Z)) + m^2) + \frac{1}{2(\mu+1)} (n + m) + \frac{\mu}{4(\mu+1)} \\
&= \frac{1}{4\mu(\mu+1)} (\text{Var}(f(X)) + \text{Var}(f(Z)) + (n + m)^2) + \frac{1}{2(\mu+1)} (n + m) + \frac{\mu}{4(\mu+1)} \\
&\leq \frac{1}{4\mu(\mu+1)} \left( \frac{n(n-1)}{2\mu} + \rho + \frac{m(m-1)}{2\mu} + \rho + (n + m)^2 \right) + \frac{1}{2(\mu+1)} (n + m) + \frac{\mu}{4(\mu+1)} \\
&= \frac{1}{4\mu(\mu+1)} \left( \frac{1}{2\mu} + 1 \right) (n^2 + m^2) + \frac{1}{4\mu(\mu+1)} (2nm) + \\
&\quad \left( \frac{1}{2(\mu+1)} - \frac{1}{4\mu^2(\mu+1)^2} \right) (n + m) + \frac{1}{2\mu(\mu+1)} \rho + \frac{\mu}{4(\mu+1)} \\
&= \frac{2\mu+1}{8\mu^2(\mu+1)^2} (n^2 + m^2) + \frac{1}{4\mu(\mu+1)} (2nm) + \frac{4\mu^2-1}{8\mu^2(\mu+1)^2} (n + m) + \rho
\end{aligned}$$

We wish to show that this last quantity is less than or equal to  $\frac{1}{2\mu}(n+m)(n+m-1) + \rho$ , by showing that  $8\mu^2(\mu+1)^2$  times their difference is nonpositive (note that for  $1 < q \leq 2$ ,  $\mu = \frac{M}{q-1} \geq M \geq 1$  and that  $8\mu^2(\mu+1)^2 > 0$ ).

$$\begin{aligned}
& 8\mu^2(\mu+1)^2 \left( \frac{2\mu+1}{8\mu^2(\mu+1)^2} (n^2 + m^2) + \frac{1}{4\mu(\mu+1)} (2nm) + \frac{4\mu^2-1}{8\mu^2(\mu+1)^2} (n + m) + \rho \right. \\
&\quad \left. - \left( \frac{1}{2\mu} (n + m) (n + m - 1) + \rho \right) \right) \\
&= (-4\mu^2 - 2\mu + 1)(n^2 + m^2) + (8\mu^2 + 4\mu - 1)(n + m) + (-4\mu^2 - 2\mu)(2nm)
\end{aligned}$$

As in the proof of Lemma 11.3, we will exhibit only a proof that

$$(-4\mu^2 - 2\mu + 1)n^2 + (8\mu^2 + 4\mu - 1)n + (-4\mu^2 - 2\mu)nm \leq 0$$

Because  $(-4\mu^2 - 2\mu) < 0$  and  $m \geq 1$ ,

$$\begin{aligned}
& (-4\mu^2 - 2\mu + 1)n^2 + (8\mu^2 + 4\mu - 1)n + (-4\mu^2 - 2\mu)nm \\
&\leq (-4\mu^2 - 2\mu + 1)n^2 + (8\mu^2 + 4\mu - 1)n + (-4\mu^2 - 2\mu)n \\
&= (-4\mu^2 - 2\mu + 1)n^2 + (4\mu^2 + 2\mu - 1)n \\
&= (-4\mu^2 - 2\mu + 1)n(n - 1) \leq 0
\end{aligned}$$

because  $n(n - 1) \geq 0$  for all  $n \geq 1$  and  $(-4\mu^2 - 2\mu + 1) < 0$  for all  $\mu \geq 1$ .  $\square$

**LEMMA 11.9.** *Let  $X$  be a Csűrös approximate counter with integer parameter  $M$  ( $M \geq 1$ ) and real parameter  $q$  ( $1 < q \leq 2$ ) whose expected value is  $n$ , let  $X'$  be the result of applying an increment operation to that counter, and let  $\mu = \frac{M}{q-1}$  and  $\rho = \frac{\mu^2}{4\mu^2+4\mu-2}$ . Assume  $\text{Var}(f(X)) \leq \frac{1}{M(M+1)}n(n-1) + \rho$ ; then  $\text{Var}(f(X')) \leq \frac{1}{M(M+1)}(n+1)n + \rho$ .*

PROOF. Analysis of the *increment* and *add* algorithms presented in Section 6 shows that  $\text{increment}(X)$  is equivalent in its behavior to  $\text{add}(X, 1)$ . Moreover, the variance of a Csűrös counter with expected value 1 is 0. Therefore we can apply Lemma 11.8.  $\square$

**THEOREM 11.10.** *Let  $X$  be a Csűrös approximate counter with integer parameter  $M$  ( $M \geq 1$ ) and real parameter  $q$  ( $1 < q \leq 2$ ) whose expected value is  $n$ , and let  $\mu = \frac{M}{q-1}$  and  $\rho = \frac{\mu^2}{4\mu^2+4\mu-2}$ . Then  $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$ .*

PROOF. We proceed by structural induction on the computation that created  $X$ .

Basis case: Suppose that  $X$  a newly created counter. Then its value is definitely 0, and therefore its expected value is 0 and its variance is 0, and so  $\text{Var}(f(X)) = 0 \leq \frac{1}{2\mu}0(0-1) + \rho = \frac{1}{2\mu}n(n-1) + \rho$ .

Inductive case 1 (incrementation): Suppose that  $X$  was produced by incrementing a counter  $Y$  having expected value  $n-1$ . By inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{1}{2\mu}y(y-1) + \rho$ , and therefore by Lemma 11.9 we have  $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$ .

Inductive case 2 (addition): Suppose that  $X$  was produced by adding a counter  $Y$  with expected value  $y$  and a counter  $Z$  with expected value  $z$ , such that  $z \leq y$  and  $y+z = n$ . By inductive hypothesis we have  $\text{Var}(f(Y)) \leq \frac{1}{2\mu}y(y-1) + \rho$  and  $\text{Var}(f(Z)) \leq \frac{1}{2\mu}z(z-1) + \rho$ , and therefore by Lemma 11.8 we have  $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$ .

Therefore in all cases  $\text{Var}(f(X)) \leq \frac{1}{2\mu}n(n-1) + \rho$ .  $\square$

Note that  $\frac{1}{6} \leq \rho < \frac{1}{4}$ , so the bound on the variance of Csűrös counters with addition, no matter what the values of  $M$  and  $q$ , is just as reasonably tight as the bound on the variance of Morris counters.

## 12. MEASUREMENTS OF TWO APPLICATIONS

We implemented a distributed version of the LDA Gibbs topic-modeling algorithm described by Tristan *et al.* [2015]. The algorithm is presented in figure 5.

The Gibbs sampler for LDA has the following parameters:  $M$  is the number of documents,  $V$  is the size of the vocabulary,  $K$  is the number of topics,  $N[M]$  is an integer array of size  $M$  that describes the shape of  $w$ ,  $\alpha$  is a parameter that controls how concentrated the distributions of topics per documents should be,  $\beta$  is a parameter that controls how concentrated the distributions of words per topics should be,  $w[M][N]$  is ragged array containing the document data (where subarray  $w[m]$  has length  $N[m]$ ),  $\theta[M][K]$  is an  $M \times K$  matrix where  $\theta[m][k]$  is the probability of topic  $k$  in document  $m$ , and  $\phi[V][K]$  is a  $V \times K$  matrix where  $\phi[v][k]$  is the probability of word  $v$  in topic  $k$ . Each element  $w[m][n]$  is a nonnegative integer less than  $V$ , indicating which word in the vocabulary is at position  $n$  in document  $m$ . The matrices  $\theta$  and  $\phi$  are typically initialized by the caller to randomly chosen distributions of topics for each document and words for each topic; these same arrays serve to deliver “improved” distributions back to the caller.

The algorithm uses three local data structures to store various statistics about the model (lines 2–4):  $tpd[M][K]$  is an  $M \times K$  matrix where  $tpd[m][k]$  is the number of times topic  $k$  is used in document  $m$ ,  $wpt[V][K]$  is an  $V \times K$  matrix where  $wpt[v][k]$  is the number of times word  $v$  is assigned to topic  $k$ , and  $wt[K]$  is an array of size  $K$  where  $wt[k]$  is the total number of time topic  $k$  is in use.

The algorithm works by iterating over the dataset  $I$  times (loop starting on line 5 and ending on line 33). An iteration starts by clearing the local data structures to zero (lines 6–8). Each iteration is composed of two phases. In the first phase (lines 10–21), we assume that the values of  $\theta$  and  $\phi$  are fixed, and to every word occurrence  $w[m][n]$  in

```

1: procedure LDA_Gibbs(int  $I$ , int  $M$ , int  $V$ , int  $K$ , int  $N[M]$ , float  $\alpha$ , float  $\beta$ ,
   int  $w[M][N]$ , var float  $\theta[M][K]$ , var float  $\phi[V][K]$ )
2:   local array int  $tpd[M][K]$ 
3:   local array int  $wpt[V][K]$ 
4:   local array int  $wt[K]$ 
5:   for all  $0 \leq i < I$  do
6:     clear array  $tpd$                                  $\triangleright$  Set every element to 0
7:     clear array  $wpt$                              $\triangleright$  Set every element to 0
8:     clear array  $wt$                              $\triangleright$  Set every element to 0
9:    $\triangleright$  Phase 1: tally statistics by sampling distributions computed from  $\theta$  and  $\phi$ 
10:  for all  $0 \leq m < M$  do
11:    for all  $0 \leq n < N$  do
12:      local array float  $p[K]$ 
13:      for all  $0 \leq k < K$  do
14:         $p[k] \leftarrow \theta[m][k] \times \phi[w[m][n]][k]$ 
15:      end for
16:      let  $z \leftarrow sample(p)$                        $\triangleright$  Now  $0 \leq z < K$ 
17:      increment  $tpd[m][z]$                           $\triangleright$  Increment counters
18:      increment  $wpt[w[m][n]][z]$                     $\triangleright$  (which may be integer
19:      increment  $wt[z]$                             $\triangleright$  or approximate)
20:    end for
21:  end for
22:  $\triangleright$  Phase 2: Compute new  $\theta$  and  $\phi$  arrays from the tallied statistics
23:  for all  $0 \leq m < M$  do
24:    for all  $0 \leq k < K$  do
25:       $\theta[m][k] \leftarrow (tpd[m][k] + \alpha) / (N[m] + K \times \alpha)$ 
26:    end for
27:  end for
28:  for all  $0 \leq v < V$  do
29:    for all  $0 \leq k < K$  do
30:       $\phi[v][k] \leftarrow (wpt[v][k] + \beta) / (wt[k] + V \times \beta)$ 
31:    end for
32:  end for
33: end for

```

Fig. 5. Pseudocode for the LDA Gibbs topic-modeling algorithm

the document data we assign a new topic, randomly chosen according to a distribution computed from  $\theta$  and  $\phi$ , and tally these choices in  $wpt$ ,  $tpd$ , and  $wt$ . In the second phase (lines 23–32), we assume that the statistics  $wpt$ ,  $tpd$ , and  $wt$  are fixed, and we compute new values for  $\theta$  and  $\phi$ ; each new value is the mean of a Dirichlet distribution induced by the statistics. We now review these two phases in detail.

In phase 1, we iterate over all the documents  $m$  (line 10) and all the words  $n$  in each document (line 11). To each word, we need to associate a topic. A topic is chosen randomly according to the following formula: the probability of associating a word  $w[m][n]$  to topic  $k$  is proportional to the probability  $\theta[m][k]$  that topic  $k$  is associated to document  $m$  times the probability  $\phi[w[m][n]][k]$  that vocabulary word word  $w[n][m]$  is associated to topic  $k$  (lines 12–15). From the resulting array  $p$  of relative (unnormalized) probabilities, we sample a new topic  $z$  (line 16) and update the statistics  $wpt$ ,  $tpd$ , and  $wt$  accordingly (lines 17–19).

In phase 2, for every document  $m$  we compute the mean of the Dirichlet distribution induced by the per-document statistics  $tpd[m]$  (lines 23–27). Then, for every vocabulary

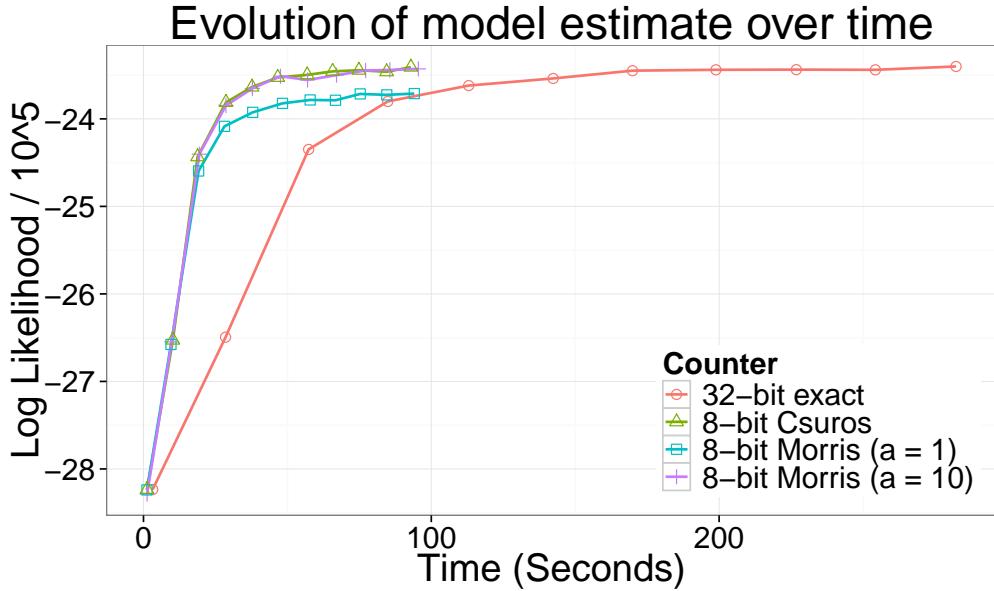


Fig. 6. 32-bit integer counters vs. 8-bit approximate counters used in a distributed (multiple-GPU) application (LDA Gibbs topic modeling) coded in CUDA using MPI, compared by log likelihood (*higher is better*)

word  $v$ , we compute the mean of the Dirichlet distribution induced by the per-word statistics  $wpt[v]$  (lines 28–32).

The algorithm was implemented on a cluster of four nodes (connected by 1Gb/s Ethernet), each with an Intel Core-i7 4820k CPU and two NVIDIA Titan Black GPU cards, for a total of 8 GPUs. The code is written in C++ and CUDA. We use MPI for internode communication. The behavior is quite similar to that of the single-GPU version reported by Tristan *et al.*: replacing integer counters with approximate counters does not affect the statistical performance of the algorithm, but allows the same hardware to process larger datasets. Moreover, the speed of the algorithm is markedly increased, largely because of a third effect: less data needs to be pushed through the network connecting the nodes. In one test, either 32-bit integer counters were used, or 8-bit approximate counters; using approximate counters improved the overall speed of each iteration of the distributed application by over 65% (see Figure 6, which shows measurements of time and log likelihood). Using 8-bit general Morris counters with  $a = 10$  (that is,  $q = \frac{11}{10} = 1.1$ ) worked well, in that topics were produced using the same number of iterations as with 32-bit integer counters, without decreasing the log-likelihood measure of statistical performance. (Because the maximum counts for this particular dataset are not large, we could have used an even smaller value of  $q$ , say 1.03, to achieve smaller variance, but we found that  $q = 1.1$  provided sufficient statistical quality and moreover was sufficient to handle the much larger counts, up to  $2^{32}$  and beyond, of larger datasets.) Also effective were 8-bit binary Csűros counters with  $s = 5$ . However, using 8-bit binary Morris counters ( $a = 1$ ) caused the algorithm to converge to a smaller (less desirable) value of log likelihood; we conjecture that this occurs because an 8-bit binary Morris counter has huge dynamic range, much of which is wasted, so the part of the range that is actually used is too coarse to be fully effective (put another way, the variance is too large).

```

1: procedure SCA(int  $I$ , int  $M$ , int  $V$ , int  $K$ , int  $N[M]$ , float  $\alpha$ , float  $\beta$ , int  $w[M][N]$ )
2:   local array int  $tpd[M][K]$ 
3:   local array int  $wpt[V][K]$ 
4:   local array int  $wt[K]$ 
5:   initialize array  $tpd[0]$                                  $\triangleright$  Randomly chosen distributions
6:   initialize array  $wpt[0]$                                  $\triangleright$  Randomly chosen distributions
7:   initialize array  $wt[0]$                                  $\triangleright$  Randomly chosen distributions
8:   for all  $0 \leq i < (I \div 2)$  do
9:     for all  $0 \leq r < 2$  do
10:      clear array  $tpd[1 - r]$                              $\triangleright$  Set every element to 0
11:      clear array  $wpt[1 - r]$                              $\triangleright$  Set every element to 0
12:      clear array  $wt[1 - r]$                              $\triangleright$  Set every element to 0
13:      for all  $0 \leq m < M$  do
14:        for all  $0 \leq n < N$  do
15:          local array float  $p[K]$ 
16:          for all  $0 \leq k < K$  do
17:            let  $\theta \leftarrow (tpd[r][m][k] + \alpha) / (N[m] + K \times \alpha)$ 
18:            let  $\phi \leftarrow (wpt[r][v][k] + \beta) / (wt[r][k] + V \times \beta)$ 
19:             $p[k] \leftarrow \theta \times \phi$ 
20:          end for
21:          let  $z \leftarrow sample(p)$                                  $\triangleright$  Now  $0 \leq z < K$ 
22:          increment  $tpd[1 - r][m][z]$                              $\triangleright$  Increment counters
23:          increment  $wpt[1 - r][w[m][n]][z]$                              $\triangleright$  (which may be integer
24:          increment  $wt[1 - r][z]$                                  $\triangleright$  or approximate)
25:        end for
26:      end for
27:    end for
28:  end for
29: end for
30: end for
31:  $\triangleright$  Final output pass
32: for all  $0 \leq m < M$  do
33:   for all  $0 \leq k < K$  do
34:     write  $\theta[m][k]$  as  $(tpd[0][m][k] + \alpha) / (N[m] + K \times \alpha)$ 
35:   end for
36: end for
37: for all  $0 \leq v < V$  do
38:   for all  $0 \leq k < K$  do
39:     write  $\phi[v][k]$  as  $(wpt[0][v][k] + \beta) / (wt[0][k] + V \times \beta)$ 
40:   end for
41: end for

```

Fig. 7. Pseudocode for the Stochastic Cellular Automaton algorithm

We have also tested a distributed implementation of a stochastic cellular automaton (SCA) for topic modeling as described by Zaheer *et al.* [2015; 2016]. The algorithm is presented in figure 7.

The SCA algorithm is best understood in comparison with the Gibbs sampler we just detailed. Like the Gibbs algorithm, the SCA algorithm iterates over the data to compute statistics for the topics (loop starting on line 9 and ending on line 30). However, SCA does not explicitly represent the entire probability matrices  $\theta$  and  $\phi$ . Since these

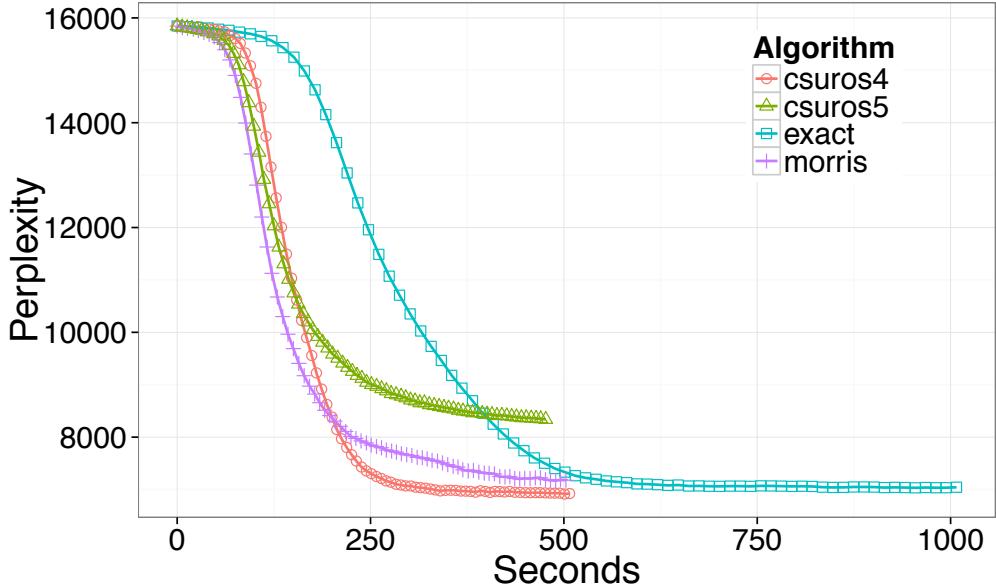


Fig. 8. 32-bit integer counters vs. 8-bit approximate counters used in a distributed (multiple-CPU) application (SCA topic modeling) coded in Java using MPI, compared by perplexity (*lower is better*)

matrices are the output of the algorithm, they need to be computed in a post-processing phase that follows the iterative phase (lines 32–41). In this post-processing phase, we compute the  $\theta$  and  $\phi$  distributions as the means of Dirichlet distributions induced by the statistics.

In the iterative phase of SCA, the values of  $\theta$  and  $\phi$ , which are necessary to compute the topic proportions, are computed on the fly (lines 19 and 20). Unlike the Gibbs algorithm, where on each iteration we have a back-and-forth between two phases, where one reads  $\theta$  and  $\phi$  in order to update the statistics and the other reads the statistics in order to update  $\theta$  and  $\phi$ , SCA performs the back-and-forth between two copies of the statistics. Therefore, the number of iterations is halved (line 9), and each iteration has two subiterations (line 10), one that reads  $tpd[0]$ ,  $wpt[0]$ , and  $wt[0]$  in order to write  $tpd[1]$ ,  $wpt[1]$ , and  $wt[1]$ , then one that reads  $tpd[1]$ ,  $wpt[1]$ , and  $wt[1]$  in order to write  $tpd[0]$ ,  $wpt[0]$ , and  $wt[0]$ . One key advantage of this algorithm over the Gibbs algorithm is that *all* the in-memory arrays are counters, so using approximate counters leads to an even smaller memory footprint and even better memory bandwidth usage than for LDA Gibbs.

We tested a version of the SCA algorithm implemented in the Java programming language. To achieve good performance, we use only arrays of primitive types and pre-allocate all arrays before learning starts. We implement multithreaded parallelization within a node using the work-stealing Fork/Join framework (tasks are recursively subdivided until the number of data points to be processed is less than  $10^5$ ). Internode communication uses the Java binding to a version of OpenMPI 1.8.7 that we modified to support the use of approximate-counter addition operations as arguments to the `allReduce` method (see Section 13). We use a sparse array representation for the counts of topics per document  $\theta$  and use Walker's alias method [Walker 1974] to draw from discrete distributions. We run our experiments on a small cluster of 16 nodes

(connected by 10Gb/s Ethernet), each with two 8-core Intel Xeon E5 processors (some nodes have Ivy Bridge processors while others have Sandy bridge processors) for a total of 32 hardware threads per node and 256GB of memory. We run 32 JVM instances (one per Xeon socket), assigning each one 20 gigabytes of memory. The number of topics ( $K$ ) is 100, the number of training iterations is 75, and  $\alpha = \beta = 0.1$ . We use two datasets, both of which are cleaned by removing stop words and rare words: we use the English Wikipedia dataset for training and the Reuters RCV1 dataset for testing. Our Wikipedia dataset has 6,749,797 documents comprising 6,749,797 tokens with a vocabulary of 291,561 words. Our Reuters dataset has 806,791 documents comprising 105,989,213 tokens with a vocabulary of 43,962 words.

Time and perplexity measurements of the SCA algorithm are shown in Figure 8. Versions that use 8-bit approximate counters (Morris with  $q = 1.08$ , and Csűrös with either  $s = 4$  or  $s = 5$ ) are approximately twice as fast as the baseline that uses 32-bit integer counters. Csűrös counters with  $s = 4$  achieved the best (lowest) perplexity in these tests. Csűrös counters with  $s = 5$  do not have enough range for the application, and we observe that its asymptotic perplexity score is markedly worse.

### 13. MODIFYING OPENMPI TO SUPPORT ADDITION OF APPROXIMATE COUNTERS

In an effort to get the best possible speed on the stochastic cellular automaton application described in Section 12, we modified the source code of OpenMPI 1.8.7 (obtained from <https://www.open-mpi.org>) to provide operations that could be used with the `allReduce` method to combine arrays of approximate counters by adding them element-wise using approximate-counter addition. In order to test the relative speeds of various implementation strategies, we used C macros to systematically produce 264 distinct implementations of approximate-counter addition. Most of the modifications were made in the files `ompi/op/op.c` and `ompi/mca/op/base/op_base_functions.c`; we also found it necessary to extend certain other tables (especially in the Java code that provides the Java binding to OpenMPI) and to arrange for the MPI initialization code to precompute additional tables.

We implemented addition operations for seven distinct 8-bit representations of approximate counters: binary Csűrös counters for  $s = 3, s = 4, s = 5, s = 6, s = 7$ , and  $s = 8$ , and general Morris counters with  $q = 1.08$ . For each of the Csűrös counters we chose the smallest possible integer type for representing estimated values in the intermediate calculations, or the smallest possible floating-point type if no integer type would suffice. For the general Morris counters we implemented two versions, one using type `float` and one using type `double` for estimated values. We also needed to decide whether to use random numbers of type `float` or type `double`. The resulting eight combinations that we used may be seen in the header lines of Table II (see page 40). Each of these combinations was then provided multiple implementations.

Each of the 264 implementations was generated by the C macro `APPROX_ADD` shown in Figure 9. The basic idea is that addition of two approximate counters, as described in Section 3, given two counter representation values  $x$  and  $y$ , consists of five steps: computing  $S$ , computing  $K$ , computing  $V$ , computing  $G = W - V$ , and finally deciding whether to add 1 to  $K$  before returning it. We always implement the last step in the same way, but each of the first four steps may be carried out in more than one way; therefore `APPROX_ADD` has four parameters `COMPUTE_S`, `COMPUTE_K`, `COMPUTE_V`, and `COMPUTE_G`. For each of these, the actual argument text will be the name of another C macro. The parameter `name` will be replaced by one of eight identifiers, indicating which of the eight operations is being implemented; the parameter `kind` is another identifier that (redundantly) encodes which actual C macros are to be used for `COMPUTE_S`, `COMPUTE_K`, `COMPUTE_V`, and `COMPUTE_G`. The parameter `s` is the  $s$  value for a Csűrös counter and is not used for a Morris counter. Each of the three parame-

---

```
#define APPROX_ADD(name,kind,s,counter,value,random,
                  COMPUTE_S,COMPUTE_K,COMPUTE_V,COMPUTE_G) \
    counter##_t name##kind(counter##_t x, counter##_t y) { \
        COMPUTE_S(name,s,counter,value); \
        COMPUTE_K(name,s,counter,value); \
        COMPUTE_V(name,s,counter,value); \
        COMPUTE_G(name,s,counter,value,random); \
        if (((next_random##random() * G) < (S - V)) \
            && (K < (counter##_t)(-1))) { ++K; } \
        return K; \
    }
```

---

Fig. 9. C macro template code for defining addition functions on approximate counters

---

```
APPROX_ADD(approx8add3,CBACC,3,uint8,uint16,float, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
APPROX_ADD(approx8add4,CBACC,4,uint8,uint32,double, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
APPROX_ADD(approx8add5,CBACC,5,uint8,uint64,double, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
APPROX_ADD(approx8add6,CBACC,6,uint8,float,float, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
APPROX_ADD(approx8add7,CBACC,7,uint8,double,double, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
APPROX_ADD(approx8add8,CBACC,8,uint8,double,double, \
           S_CSUROS_SUM,K_BINARY_SEARCH_CSUROS,V_CSUROS,G_FROM_W_CSUROS) \
           : \
APPROX_ADD(approx8add3,TMTW,3,uint8,uint16,float, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8add4,TMTW,4,uint8,uint32,double, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8add5,TMTW,5,uint8,uint64,double, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8add6,TMTW,6,uint8,float,float, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8add7,TMTW,7,uint8,double,double, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8add8,TMTW,8,uint8,double,double, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8adg,TMTW,(%unused%),uint8,double,double, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE) \
APPROX_ADD(approx8addf,TMTW,(%unused%),uint8,float,float, \
           S_TABLE_SUM,K_MATRIX_LOOKUP,V_TABLE,G_FROM_W_TABLE)
```

---

Fig. 10. C macro invocations for defining addition functions on approximate counters (14 out of 264 shown)

ters `counter`, `value`, and `random` indicates a type (one of `uint8`, `uint16`, `uint32`, `uint64`, `float`, `double`) to be used for the representation of respectively the approximate counters, their estimated values, and generated pseudorandom numbers. Figure 10 shows

---

```
#define APPROX_ADD_SENTINEL_float ((float)(ldexp(1.0, 70)))

#define APPROX_ADD_SENTINEL_double (ldexp(1.0, 500))

#define ONE16 ((uint16_t) 1)

#define ONE32 ((uint32_t) 1)

#define ONE64 ((uint64_t) 1)
```

---

Fig. 11. C macros for certain useful constants

---

```
#define INTERPRET_BINARY_CSUROS_uint16(x,s) \
  ((uint16_t)(INTERPRET_BINARY_CSUROS_uint32(x,s))) \
  \

#define INTERPRET_BINARY_CSUROS_uint32(x,s) \
  (((ONE32 << (s)) + ((x) & ((ONE32 << (s)) - ONE32))) << ((x) >> (s))) \
  - (ONE32 << (s)))

#define INTERPRET_BINARY_CSUROS_uint64(x,s) \
  (((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64))) << ((x) >> (s))) \
  - (ONE64 << (s)))

#define INTERPRET_BINARY_CSUROS_double(x,s) \
  (ldexp((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64)), (x) >> (s)) \
  - (ONE64 << (s)))

#define INTERPRET_BINARY_CSUROS_float(x,s) \
  ((float_t)(ldexp((ONE64 << (s)) + ((x) & ((ONE64 << (s)) - ONE64)), \
  (x) >> (s)) \
  - (ONE64 << (s)))) \
  \
```

---

Fig. 12. C macros for calculating the estimated value from a Csűrös counter representation x

the first six and the last eight of the 264 specific invocations of the APPROX\_ADD macro that we used in our experiments. (In all 264 cases, the counter type was uint8; the APPROX\_ADD macro was designed also to support implementation of 16-bit approximate counters, but we have not yet investigated such cases.)

Figure 11 shows C macros that define names for some useful constants, namely sentinel values (of type float and double) that are much larger than any actual table entries, and the constant 1 of various data types (especially useful as left-hand operand of the C left-shift operator <<).

Figure 12 shows C macros that, given a representation value for a Csűrös counter and the value of s for that counter representation, computes the estimated value of the counter. Five versions are provided, one for each of the possible types uint16, uint32, uint64, float, double that might be used for representing the estimated value.

Figure 13 shows C macros that, given a representation value x for a Csűrös counter and its corresponding estimated value v and the value of s for that counter representation, computes the estimated value of the counter representation value x+1. This can be done more cheaply than computing the estimated value for x+1 from scratch. Again

---

```
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint16(x,v,s) \
    ((v) + (ONE16 << ((x) >> (s)))) \
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint32(x,v,s) \
    ((v) + (ONE32 << ((x) >> (s)))) \
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_uint64(x,v,s) \
    ((v) + (ONE64 << ((x) >> (s)))) \
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_double(x,v,s) \
    ((v) + ldexp(1.0, ((x) >> (s)))) \
#define INCREMENTALLY_UPDATE_BINARY_CSUROS_float(x,v,s) \
    ((v) + (float_t)ldexp(1.0, ((x) >> (s))))
```

---

Fig. 13. C macros for calculating the estimated value for  $x+1$  given  $x$  and its estimated value  $v$ 


---

```
#define S_CSUROS_SUM(name,s,counter,value) \
    value##_t S = (INTERPRET_BINARY_CSUROS_##value(x,s) + \
                    INTERPRET_BINARY_CSUROS_##value(y,s));
```

(C) Compute  $S$  by adding Csűrös values calculated from  $x$  and  $y$ 


---

```
#define S_TABLE_SUM(name,s,counter,value) \
    value##_t S = (name##_interpret##value[x] + \
                    name##_interpret##value[y]);
```

(T) Compute  $S$  by adding value-table entries indexed by  $x$  and  $y$ Fig. 14. C macros for calculating  $S$  from  $x$  and  $y$ 

five versions are provided, one for each of the possible types uint16, uint32, uint64, float, double that might be used for representing the estimated value.

Figure 14 shows two different ways of computing  $S$ . The C macro S\_CSUROS\_SUM (also identified in abbreviated contexts, such as in Table II, by the letter C) uses the relevant estimate-calculation macro from Figure 12 twice (once on  $x$  and once on  $y$ ) and adds the results. The C macro S\_TABLE\_SUM (also identified by the letter T) instead assumes there is a precomputed table, whose name is constructed by the phrase name##\_interpret##value, which can be indexed by  $x$  and then by  $y$  to fetch two values to be added. The choice between these two macros for computing  $S$  reflects a potential space-time tradeoff that we wished to measure.

Figures 15 and 16 together show seven different ways of computing  $K$ . The C macro K\_CSUROS\_CALCULATE (also identified by C) in Figure 15 uses the relevant type-specific C macro (also shown in that same figure) to calculate  $K$  from  $S$  by using logarithms (computed using \_\_builtin\_clz if  $S$  has an integer representation, or using frexp or frexp if  $S$  has a floating-point representation). Five of the six alternative C macros in Figure 16 use another approach, starting from the larger of  $x$  and  $y$  and searching upward to find the appropriate value of  $K$ ; the last alternative instead uses a large two-dimensional table. The choices among the seven macros for computing  $K$  reflect potential space-time tradeoffs that we wished to measure.

---

```
#define K_CSUROS_CALCULATE(name,s,counter,value)
    K_CSUROS_CALCULATE_##value(s)
```

(C) Calculate  $K$  from  $S$  (by dispatching to a type-specific macro below)

```
#define K_CSUROS_CALCULATE_uint16(s)
    K_CSUROS_CALCULATE_uint32(s)
```

How to calculate  $K$  from an  $S$  value of type uint16

```
#define K_CSUROS_CALCULATE_uint32(s)
    uint32_t Sprime = S + (1 << s);
    /* Note: __builtin_clz operates on a 32-bit value */
    uint32_t d = (31 - (1 << s)) - __builtin_clz(Sprime);
    uint32_t K = (d << s) + (Sprime >> d) - (1 << s);
```

How to calculate  $K$  from an  $S$  value of type uint32

```
#define K_CSUROS_CALCULATE_uint64(s)
    uint64_t Sprime = S + (1 << s);
    /* Note: __builtin_clz operates on a 32-bit value */
    uint32_t d = (Sprime >= (((uint64_t)1)<<32)) ?
        (63 - (1 << s)) - __builtin_clz((uint32_t)(Sprime >> 32)) :
        (31 - (1 << s)) - __builtin_clz((uint32_t)Sprime);
    uint32_t K = (d << s) + ((uint32_t)(Sprime >> d)) - (1 << s);
```

How to calculate  $K$  from an  $S$  value of type uint64

```
#define K_CSUROS_CALCULATE_float(s)
    float_t Sprime = S + (1 << s);
    int d;
    (void) frexpf(Sprime, &d);
    d -= ((1 << s) + 1);
    uint32_t K = (d << s) + (uint32_t)ldexpf(Sprime, -d) - (1 << s);
```

How to calculate  $K$  from an  $S$  value of type float

```
#define K_CSUROS_CALCULATE_double(s)
    double_t Sprime = S + (1 << s);
    int d;
    (void) frexp(Sprime, &d);
    d -= ((1 << s) + 1);
    uint32_t K = (d << s) + (uint32_t)ldexp(Sprime, -d) - (1 << s);
```

How to calculate  $K$  from an  $S$  value of type double

Note: `__builtin_clz` takes a 32-bit integer and returns the number of leading zero bits in the binary representation of the operand. The definition of `K_CSUROS_CALCULATE_uint64` could be greatly simplified if a 64-bit version of this count-leading-zeroes operation were available.

---

Fig. 15. C macros for calculating  $K$  from either  $S$  or  $x$  and  $y$  (part 1 of 2)

---

```
#define K_LINEAR_SEARCH_CSUROS(name,s,counter,value) \
    uint32_t K = (x > y) ? x : y; \
    while (S >= INTERPRET_BINARY_CSUROS##value(K+1,s)) ++K;
```

(LA) Linear search of Absolutely calculated Csűrös values

```
#define K_LINEAR_INCR_CSUROS(name,s,counter,value) \
    uint32_t K = (x > y) ? x : y; \
    value##_t Z = INTERPRET_BINARY_CSUROS##value(K+1,s); \
    while ((S >= Z) && (K != (counter##_t)(-1))) { \
        ++K; \
        Z = INCREMENTALLY_UPDATE_BINARY_CSUROS##value(K,Z,s) \
    }
```

(LI) Linear search of Incrementally calculated Csűrös values

```
#define K_BINARY_SEARCH_CSUROS(name,s,counter,value) \
    uint32_t K = (x > y) ? x : y; \
    /* This loop and binary search cannot blow past the 8 sentinels */ \
    while (S >= INTERPRET_BINARY_CSUROS##value(K+8,s)) K += 8; \
    if (S >= INTERPRET_BINARY_CSUROS##value(K+4,s)) K += 4; \
    if (S >= INTERPRET_BINARY_CSUROS##value(K+2,s)) K += 2; \
    if (S >= INTERPRET_BINARY_CSUROS##value(K+1,s)) K += 1;
```

(BA) Binary search after linear search of every eighth Absolutely calculated value

```
#define K_LINEAR_SEARCH_TABLE(name,s,counter,value) \
    uint32_t K = (x > y) ? x : y; \
    /* This loop cannot blow past the sentinel */ \
    while (S >= name##_interpret##value[K+1]) ++K;
```

(LT) Linear search of entries in value Table

```
#define K_BINARY_SEARCH_TABLE(name,s,counter,value) \
    uint32_t K = (x > y) ? x : y; \
    /* This loop and binary search cannot blow past the 8 sentinels */ \
    while (S >= name##_interpret##value[K+8]) K += 8; \
    if (S >= name##_interpret##value[K+4]) K += 4; \
    if (S >= name##_interpret##value[K+2]) K += 2; \
    if (S >= name##_interpret##value[K+1]) K += 1;
```

(BT) Binary search after linear search of every eighth entry in value Table

```
#define K_MATRIX_LOOKUP(name,s,counter,value) \
    uint32_t K = name##_lookup[x][y];
```

(M) lookup in a 2-D Matrix using x and y (and ignoring S)

---

Fig. 16. C macros for calculating  $K$  from either  $S$  or  $x$  and  $y$  (part 2 of 2)

The C macro K\_LINEAR\_SEARCH\_CSUROS (also identified by LA) sets  $K$  equal to the larger of  $x$  and  $y$  and then repeatedly increments  $K$  until the calculated estimated value for  $K + 1$  is not larger than  $S$ . (This technique depends on the fact that  $K$  is represented using a sufficiently large data type that the calculation of  $K$  cannot overflow.)

The C macro K\_LINEAR\_INCR\_CSUROS (also identified by LI) likewise sets  $K$  equal to the larger of  $x$  and  $y$  and then repeatedly increments  $K$  until the estimated value for  $K + 1$  is not larger than  $S$ , but it calculates estimated values using the incremental technique of the macros in Figure 13.

The C macro K\_BINARY\_SEARCH\_CSUROS (also identified by BA) sets  $K$  equal to the larger of  $x$  and  $y$  and then repeatedly adds 8 to  $K$  until the calculated estimated value for  $K + 8$  is not larger than  $S$ ; then it successively tries increments of 4, 2, and 1. The next effect is to search linearly for an appropriate block of 15 values, then perform binary search on that block. Our intuition was that in practice the loop would nearly always perform at most one iteration, so this technique would nearly always behave as a straight binary search; the question was, would it beat a linear search? (This technique depends on the fact that  $K$  is represented using a sufficiently large data type that the calculation of  $K$  cannot overflow.)

The C macro K\_LINEAR\_SEARCH\_TABLE (also identified by LT) corresponds to the *add* pseudocode at the end of Section 4 (see page 7); it is similar to K\_LINEAR\_SEARCH\_CSUROS but uses a table of estimated values rather than calculating them. (This technique depends on the additional fact that each table named name##\_interpret\_##value is padded at the end with one copy of a sentinel value, of the appropriate type, from Figure 11.)

The C macro K\_BINARY\_SEARCH\_TABLE (also identified by BT) is similar to K\_BINARY\_SEARCH\_CSUROS but uses a table of estimated values rather than calculating them. (This technique depends on the additional fact that each table named name##\_interpret\_##value is padded at the end with *eight* copies of a sentinel value, of the appropriate type, from Figure 11.)

The last C macro in Figure 16, K\_MATRIX\_LOOKUP (also identified by M), ignores  $S$  and instead uses  $x$  and  $y$  to index into a precomputed two-dimensional table. This is expected to be quite fast, but the size of the table is fairly large (half a megabyte for the case of 8-bit approximate counters and a double representation for estimated values).

Figure 17 shows two different ways of computing  $V$ . The C macro V\_CSUROS (also identified by C) uses the relevant estimate-calculation macro from Figure 12 on  $K$ . The C macro V\_TABLE (also identified by T) instead assumes there is a precomputed table, whose name is constructed by the phrase name##\_interpret\_##value, which can be indexed by  $K$  to fetch the estimated value. The choice between these two macros for computing  $S$  reflects a potential space-time tradeoff that we wished to measure.

Figure 18 shows three different ways of computing  $G$ . The C macro G\_FROM\_W\_CSUROS (also identified by C) uses the relevant estimate-calculation macro from Figure 12 on  $K + 1$ ; then it subtracts  $V$ . The C macro G\_FROM\_W\_TABLE (also identified by W) instead assumes there is a precomputed table, whose name is constructed by the phrase name##\_interpret\_##value, which can be indexed by  $K+1$  to fetch the estimated value; then it subtracts  $V$ . The C macro G\_TABLE (also identified by T) bypasses the computation of  $W$ , and instead assumes there is a second precomputed table, whose name is constructed by the phrase name##\_gap\_##random, which can be indexed by  $K$  to fetch the precomputed quantity  $W - V$ . The choice among these three macros for computing  $S$  reflects a potential space-time tradeoff that we wished to measure.

We have now exhibited two ways to compute  $S$ , seven ways to compute  $K$ , two ways to compute  $V$ , and three ways to compute  $W$ . However, in our experiments we chose to take  $V$  from a table (using macro V\_TABLE) if and only if  $G$  was also taken from a table (using either G\_FROM\_W\_TABLE or G\_TABLE). Therefore we tried  $2 \times 7 \times 3 = 42$  different

Table II. Average benchmark execution time for 264 implementation variations of approximate-counter addition

				Csűrös $s = 3$ uint16 float	Csűrös $s = 4$ uint32 double	Csűrös $s = 5$ uint64 double	Csűrös $s = 6$ float	Csűrös $s = 7$ double	Csűrös $s = 8$ double	general $q = 1.08$ double	general $q = 1.08$ float
<i>S</i>	<i>K</i>	<i>V</i>	<i>G</i>								
C	BA	C	C	9097	9073	8715	25935	24607	26963		
C	BA	T	T	8610	8651	8453	20512	19000	19627		
C	BA	T	W	8082	8250	8056	21774	19692	21345		
C	BT	C	C	7711	7958	7515	15794	15542	13452		
C	BT	T	T	7265	7236	7305	10813	10782	9711		
C	BT	T	W	7310	6962	6892	11019	10606	9726		
C	C	C	C	7885	7663	7935	17868	17171	18223		
C	C	T	T	7382	7375	7685	13132	12544	12159		
C	C	T	W	7034	7281	7628	13093	12582	12744		
C	LA	C	C	8567	7551	7273	16979	19231	17659		
C	LA	T	T	8491	7263	7085	13002	12061	10850		
C	LA	T	W	8477	7037	6723	13105	12564	10961		
C	LI	C	C	7797	7370	6879	17303	18176	15401		
C	LI	T	T	7587	7199	6904	12842	11603	11028		
C	LI	T	W	7421	7364	6521	12781	13095	11080		
C	LT	C	C	7151	6751	6703	16545	15292	12945		
C	LT	T	T	6695	6294	6456	9965	9840	8765		
C	LT	T	W	6988	6305	6373	10056	9555	8924		
C	M	C	C	6831	6761	6347	16447	13718	13420		
C	M	T	T	5959	6166	6386	10027	9441	8680		
C	M	T	W	6227	6173	6318	9849	9625	8906		
T	BA	C	C	8692	8442	8441	22182	21189	21172		
T	BA	T	T	8204	7920	7949	18050	14820	15848		
T	BA	T	W	7822	7897	7745	18176	15022	16043		
T	BT	C	C	7554	7333	7189	13031	11290	9904		
T	BT	T	T	6946	6776	6879	6836	6345	6014	6842	6860
T	BT	T	W	7022	6514	6625	7031	6436	6295	7092	7056
T	C	C	C	8033	7636	8059	13886	13100	12398		
T	C	T	T	7160	7220	7654	8922	8595	8540		
T	C	T	W	7031	6969	7322	8767	8709	8318		
T	LA	C	C	8068	7188	6909	12696	12246	11328		
T	LA	T	T	7791	6881	6836	8342	7638	7049		
T	LA	T	W	7469	6836	6526	8417	7757	7040		
T	LI	C	C	7564	6792	6660	12727	12033	10994		
T	LI	T	T	7272	6683	6636	8618	7785	6806		
T	LI	T	W	7106	6566	6318	9248	7691	6828		
T	LT	C	C	6792	6590	6323	10079	9525	8804		
T	LT	T	T	6573	6148	6341	5594	5249	4960	6021	6075
T	LT	T	W	6643	6118	6218	5608	5265	5007	6096	6092
T	M	C	C	6445	6373	6220	9794	9494	8728		
T	M	T	T	<b>5929</b>	<b>5721</b>	<b>5894</b>	<b>5555</b>	<b>5173</b>	<b>4856</b>	<b>5839</b>	<b>5711</b>
T	M	T	W	6076	5818	5978	5606	5308	4962	5882	5849

Each entry is a time in milliseconds, computed by making nine measurements of benchmark execution time, discarding the highest and lowest, and averaging the remaining seven. The maximum relative standard deviation in any set of seven averaged measurements was less than 0.122, and the average relative standard deviation over all 264 table entries was less than 0.022. The smallest time in each column is shown in **boldface**. These times may be compared to those in Table III.

Table III. Average benchmark execution time for 10 standard MPI combining operations

MAX	MIN	SUM	PROD	LAND	BAND	LOR	BOR	LXOR	BXOR
1913	1924	1869	1875	1891	1862	1887	1854	1940	1854

Each entry is a time in milliseconds, computed by making nine measurements of benchmark execution time, discarding the highest and lowest, and averaging the remaining seven. Compare these to Table II.

---

```
#define V_CSUROS(name,s,counter,value) \
    value##_t V = INTERPRET_BINARY_CSUROS_##value(K,s);
```

---

(C) Compute  $V$  as a Csűrös value calculated from  $K$

```
#define V_TABLE(name,s,counter,value) \
    value##_t V = name##_interpret_##value[K];
```

---

(T) Compute  $V$  by using a value-table entry indexed by  $K$

---

Fig. 17. C macros for calculating  $V$  from  $K$

---

```
#define G_FROM_W_CSUROS(name,s,counter,value,random) \
    value##_t W = INTERPRET_BINARY_CSUROS_##value(K+1,s); \
    value##_t G = W - V;
```

---

(C) Compute  $G$  by subtracting  $V$  from a Csűrös value calculated from  $K + 1$

```
#define G_FROM_W_TABLE(name,s,counter,value,random) \
    value##_t W = name##_interpret_##value[K+1]; \
    value##_t G = W - V;
```

---

(W) Compute  $G$  by subtracting  $V$  from a value-table entry indexed by  $K + 1$

```
#define G_TABLE(name,s,counter,value,random) \
    value##_t G = name##_gap_##random[K];
```

---

(T) Compute  $G$  by using a gap-table entry indexed by  $K$

---

Fig. 18. C macros for calculating  $G$  from  $K$  and possibly  $V$

---

implementation combinations. For each of the six Csűrös counter representations, we implemented all 42 combinations, but for the two implementations of Morris counters, only the 6 implementation combinations that use tables (thus avoiding any use of `INTERPRET_BINARY_CSUROS_xxx` macros) are applicable. Therefore we produced a total of  $6 \times 42 + 2 \times 6 = 252 + 12 = 264$  implementations of approximate-counter addition.

To measure the relative speed of these implementations, we used an artificial benchmark that within each MPI process constructed an array of 8-bit approximate counters and then repeatedly did two things: initialize the array and then use the `allReduce` method to combine one array from each process, using one of the 264 implementations of approximate-counter addition as the combining operation. Only the execution time of the loop was measured. To provide a baseline for comparison, we also took measurements of the same benchmark using the standard MPI combining operators `MAX`, `MIN`, `SUM`, `PROD`, `LAND`, `BAND`, `LOR`, `BOR`, `LXOR`, and `BXOR` on 8-bit values.

We tried array lengths of 10000, 100000, and 1000000, adjusting the number of iterations to keep measured run times between 2 and 30 seconds (most were at least 5 seconds); we also tried various ways of initializing the arrays. These variations had some effect on the absolute measured run time, but had almost no effect at all on the relative comparisons. Therefore we present just one set of measurements here, in Table II, as being representative of all our observations. For these measurements the array length was 1000000, the number of iterations was 250, and every element of every array was initialized to 43. The measurements were taken on the same hard-

ware (16 nodes containing two Xeon processors each) used for the SCA application measurements described in Section 12, using 32 JVM instances communicating with one another as 32 MPI processes. Each instance of the benchmark was run nine times; from each set of nine measurements, the lowest and highest values were discarded and the other seven averaged to produce the data presented in Table II. For comparison, similarly processed measurements of 10 standard MPI combining operations applied to the same data are presented in Table III.

The results are perhaps unsurprising. In the specific CPU-based hardware/software environment that we measured, we generally find that:

- Approximate-counter addition is slower than any of the 10 standard MPI combining operations.
- Incremental calculations during a linear search (LI) are faster than from-scratch absolute calculations during a linear search (LA).
- Linear search turns out to be faster than binary search.
- Table-based techniques are faster than making calculations.

Our conclusion, however, is not that any one implementation technique is superior in all circumstances, but rather that it may be worthwhile to compare the speed of several such techniques in each new computational environment. In a GPU-based environment, for example, it may be infeasible to use a very large table, or it may be that some form of calculation is faster than using even small tables, or it may be that binary search is faster than linear search, either because of SIMD constraints or because the value of  $q$  is small (very close to 1).

#### 14. DON'T SUBTRACT (OR DECREMENT) APPROXIMATE COUNTERS

It is conceptually straightforward to define a subtraction operation for approximate counters: just take the pseudocode for the *add* operation in Section 3 and change the “+” to “-” in the computation of  $S$  in line 4:

```

1: procedure subtract(var  $X: T, Z: T$ )
2:   let  $v \leftarrow f(X)$ 
3:   let  $w \leftarrow f(Z)$ 
4:   let  $S \leftarrow v - w$                                  $\triangleright$  We do not recommend use of this procedure!
5:   let  $K \leftarrow \varphi(S)$ 
6:   let  $V \leftarrow f(K)$ 
7:   let  $W \leftarrow f(\tau(K))$ 
8:   let  $\Delta \leftarrow \frac{S-V}{W-V}$ 
9:   if  $\text{random}() < \Delta$  then
10:     $X \leftarrow \tau(K)$ 
11:   else
12:     $X \leftarrow K$ 
```

It is similarly easy to define a *decrement* operation (perhaps in terms of *subtract*).

But this is *not* a good idea. Subtraction increases the variance of the result in the same manner as addition, but the result itself may be smaller in magnitude (absolute value) than either of its operands, so a series of addition and subtraction operations can cause the relative standard deviation to grow without bound.

We tried to obtain empirical confirmation of this theoretical observation by taking a well-known LDA algorithm that, rather than recalculating all counts from scratch on each iteration, instead updates counters incrementally: whenever the topic assigned to a word is changed, it decrements the words-per-topic count for the old topic and then increments the words-per-topic count for the new topic. We straightforwardly modified

this algorithm to use approximate counters rather than standard integer counters. As expected, it behaved extremely poorly. It wasn't just that the log likelihood of the computed parameters failed to converge, even after an enormous number of iterations; there was gross misbehavior. We finally realized that the correctness of the algorithm depends critically on the fact that every counter always has a nonnegative value—but when incrementations and decrements are only statistical, a probabilistic counter can take on negative values even if the number of decrementation attempts is always smaller than the number of incrementation attempts. Trying to address this naïvely by saturating at the lower end (that is, refusing to decrement a counter that is already zero) introduced biases that violated other invariants of the algorithm, such as that the sum (or at least the expected sum) over an entire array should remain constant. While this particular experiment did not fully verify that the variance might grow without bound in practice, it did support the general hypothesis that pitfalls await the unwary who use subtraction on approximate counters.

### 15. MULTIPLICATION OF APPROXIMATE COUNTERS MIGHT BE WELL-BEHAVED

It *might* be reasonable to define a multiplication operation for approximate counters and to attempt to prove an appropriate bound on the variance of the product. However, we have not yet encountered or thought of a practical application for such an operation.

### 16. WHY HASN'T THIS BEEN DONE BEFORE?

When we set out to test a distributed version of the single-GPU LDA Gibbs algorithm with approximate counters, we recognized the need to replicate the counters—simple tests showed that trying to increment counters stored on a remote node would result in much slower performance—and therefore the need to add approximate counter values. We thought it would be easy to locate the necessary algorithm in the literature; approximate counters have been around for almost four decades, and it's an "obvious" operation to provide. But our best efforts uncovered no mention at all of this operation or anything like it.

We speculate that the need simply has not arisen until now, and offer a "just-so" story: If counters are stored in a central memory, then it never makes sense to use a replicated representation for the counters; if one can afford to store two copies of an 8-bit approximate counter, then one is better off using one 16-bit approximate counter, affording greater range or precision or both. So adding approximate counters makes sense only in a distributed setting. But we have found that most uses of approximate counters in the literature have had to do with counting events (such as performance counters in a hardware processor); it does make sense, for example, for every processor in a cluster to have its own counters, but typically one clears the counters, runs a computation, and then gathers and aggregates the performance data just once, after the computation has completed. For database applications, past use has typically focused on counting features while making a single (possibly distributed) pass over the database. In such cases there is nothing to be gained by reducing the aggregated values back to the approximate-counter representation; rather, one communicates the approximate counter values, expands each to a full integer, and then sums the integers—that's all there is to it.

What motivates addition of approximate counters is an *iterative* distributed application that can use replicated approximate counters within each iteration, where each iteration also includes the aggregation and redistribution of such replicated counters. We found that machine-learning algorithms (such as topic modeling) and stochastic cellular automata fit this description and benefit accordingly. We admit that one benefit of adding approximate counters in such applications, namely the reduction in network traffic, could be had in other ways, such as applying a generic data-compression

algorithm to an array of ordinary integer counters. However, the addition process is fairly quick, and there are other benefits to maintaining intermediate values in approximate-counter form, such as reduction of memory footprint.

## 17. CONCLUSIONS AND FUTURE WORK

Statistically independent approximate counters can be added, producing a result in the same representation, so that the expected value of the result is the sum of the expected values of the operands, and the variance of the result is bounded. We present specific novel algorithms for adding five kinds of approximate counters in the literature; for three of them (general Morris, binary Morris, and Csürös) we present proofs of bounded variance. We report that replacing integer counters with approximate counters maintains the statistical behavior of a distributed multiple-GPU implementation of a machine-learning application while improving its overall performance by almost a factor of 3, and of a distributed multiple-CPU implementation of another machine-learning application while improving its overall performance by almost a factor of 2.

It remains to produce proofs of bounded variance (if possible) for the other two kinds of approximate counters (DLM probability and DLM floating-point), to measure the relative speeds of various implementations of approximate-counter addition in other computational environments, and to investigate what other sorts of applications might benefit from adding approximate counters.

## ACKNOWLEDGMENTS

We thank Yossi Lev for discussion of this material and for pointing us to important related work. We also thank Daniel Ford and the anonymous referees for helpful comments.

## REFERENCES

- A. C. Callahan. 1976. Random Rounding: Some Principles and Applications. In *ICASSP '76: IEEE Intl. Conf. Acoustics, Speech, and Signal Processing*, Vol. 1. IEEE, New York, 501–504. DOI:<http://dx.doi.org/10.1109/ICASSP.1976.1169938>
- Miklós Csürös. 2010. Approximate Counting with a Floating-point Counter. In *COCOON '10: Proc. 16th Annual International Conf. Computing and Combinatorics*. Springer-Verlag, Berlin, Heidelberg, 358–367. <http://dl.acm.org/citation.cfm?id=1886811.1886857>
- Andrej Cvetkovski. 2007. An Algorithm for Approximate Counting Using Limited Memory Resources. In *SIGMETRICS '07: Proc. 2007 ACM SIGMETRICS International Conf. Measurement and Modeling of Computer Systems*. ACM, New York, 181–190. DOI:<http://dx.doi.org/10.1145/1254882.1254903>
- Dave Dice, Yossi Lev, and Mark Moir. 2013. Scalable Statistics Counters. In *PPoPP '13: Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. ACM, New York, 307–308. DOI:<http://dx.doi.org/10.1145/2442516.2442558>
- Philippe Flajolet. 1985. Approximate counting: A detailed analysis. *BIT Numerical Mathematics* 25, 1 (1985), 113–134. DOI:<http://dx.doi.org/10.1007/BF01934993>
- George E. Forsythe. 1959. Reprint of a Note on Rounding-Off Errors. *SIAM Rev.* 1, 1 (1959), 66–67. DOI:<http://dx.doi.org/10.1137/1001011>
- Scott A. Mitchell and David M. Day. 2011. Flexible Approximate Counting. In *IDEAS '11: Proc. 15th Symp. International Database Engineering & Applications*. ACM, New York, 233–239. DOI:<http://dx.doi.org/10.1145/2076623.2076655>
- Robert Morris. 1978. Counting Large Numbers of Events in Small Registers. *Commun. ACM* 21, 10 (Oct. 1978), 840–842. DOI:<http://dx.doi.org/10.1145/359619.359627>
- D. Stott Parker. 1997. *Monte Carlo Arithmetic: Exploiting Randomness in Floating-Point Arithmetic*. Technical Report 970002. Computer Science Department, UCLA, Los Angeles, CA. <http://fmdb.cs.ucla.edu/Treports/970002.pdf>
- D. S. Parker, B. Pierce, and P. R. Eggert. 2000. Monte Carlo Arithmetic: How to Gamble with Floating Point and Win. *Computing in Science Engineering* 2, 4 (July 2000), 58–68. DOI:<http://dx.doi.org/10.1109/5992.852391>

- Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI—The Complete Reference: Volume 1, The MPI Core* (second ed.). MIT Press, Cambridge, Massachusetts.
- Rade Stanojevic. 2007. Small Active Counters. In *26th IEEE International Conf. Computer Communications (INFOCOM 2007)*. IEEE, Piscataway, New Jersey, 2153–2161. DOI: <http://dx.doi.org/10.1109/INFCOM.2007.249>
- Guy L. Steele Jr. and Jean-Baptiste Tristan. 2016. Adding Approximate Counters. In *PPoPP '16: Proc. 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, Article 15, 12 pages. DOI: <http://dx.doi.org/10.1145/2851141.2851147>
- Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. 2015. Efficient Training of LDA on a GPU by Mean-For-Mode Gibbs Sampling. In *ICML 2015: 32nd International Conf. Machine Learning*, Vol. 37. Journal of Machine Learning Research / Microtome Publishing, Brookline, Massachusetts, 10. <http://jmlr.org/proceedings/papers/v37/tristan15.pdf>
- A. J. Walker. 1974. Fast generation of uniformly distributed pseudorandom numbers with floating-point representation. *Electronics Letters* 10, 25 (December 1974), 533–534. DOI: <http://dx.doi.org/10.1049/el:19740423>
- Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy L. Steele Jr. 2015. Exponential Stochastic Cellular Automata for Massively Parallel Inference. In *LearningSys: Workshop on Machine Learning Systems at Neural Information Processing Systems (NIPS)*. 10. [http://learningsys.org/papers/LearningSys\\_2015\\_paper\\_11.pdf](http://learningsys.org/papers/LearningSys_2015_paper_11.pdf)
- Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy L. Steele Jr. 2016. Exponential Stochastic Cellular Automata for Massively Parallel Inference. In *AISTATS: Proc. 19th International Conf. Artificial Intelligence and Statistics*. Journal of Machine Learning Research, 966–975. <http://jmlr.org/proceedings/papers/v51/zaheer16.pdf>

---

# Unlocking Fairness: a Trade-off Revisited

---

**Michael Wick, Swetasudha Panda, Jean-Baptiste Tristan**

{michael.wick,swetasudha.panda,jean.baptiste.tristan}@oracle.com  
Oracle Labs, Burlington, MA.

## Abstract

The prevailing wisdom is that a model’s fairness and its accuracy are in tension with one another. However, there is a pernicious *modeling-evaluating dualism* bedeviling fair machine learning in which phenomena such as label bias are appropriately acknowledged as a source of unfairness when designing fair models, only to be tacitly abandoned when evaluating them. We investigate fairness and accuracy, but this time under a variety of controlled conditions in which we vary the amount and type of bias. We find, under reasonable assumptions, that the tension between fairness and accuracy is illusive, and vanishes as soon as we account for these phenomena during evaluation. Moreover, our results are consistent with an opposing conclusion: fairness and accuracy are sometimes in accord. This raises the question, *might there be a way to harness fairness to improve accuracy after all?* Since many notions of fairness are with respect to the model’s predictions and not the ground truth labels, this provides an opportunity to see if we can improve accuracy by harnessing appropriate notions of fairness over large quantities of *unlabeled* data with techniques like posterior regularization and generalized expectation. We find that semi-supervision improves both accuracy and fairness while imparting beneficial properties of the unlabeled data on the classifier.

## 1 Introduction

Torrents of ink have been spilled characterizing the relationship between a classifier’s “fairness” and its accuracy [11, 7, 3, 8, 20, 4, 14, 2, 13, 17], where fairness refers to a concrete mathematical embodiment of some rule provided by an external party such as a government and which must be imposed on a learning algorithm. The consensus, countenanced by both empirical and analytical studies, is that the relationship is a trade-off: satisfying the supplied fairness constraints is achieved only at the expense of accuracy. On the one hand, these findings are intuitive: if we think of fairness as constraints limiting the set of possible classification assignments to those that are collectively fair, then clearly accuracy suffers because in general, optimization over the subset always lower bounds optimization over the original set. As put in another paper “demanding fairness of models *always* come at a cost of reduced accuracy” [2].<sup>1</sup>

On the other hand, the belief in a simple assumption immediately calls these findings into question. In particular, it requires no stretch of credulity to imagine that various personal attributes (e.g., race, gender, religion; sometimes termed “protected attributes”) have no bearing on a person’s intelligence, capability, potential, qualifications, etc., and consequently no bearing on ground truth classification labels — such as job qualification status — that might be functions of these qualities.<sup>2</sup> It then follows that enforcing fairness across these attributes should on average *increase* accuracy. The reason is clear. If our classifier produces different label distributions depending on the values of these dimensions, then we know, under the foregoing assumption, that at least one of these distributions must be wrong, and thus there is an opportunity to improve accuracy. An opportunity to which we later return.

---

<sup>1</sup>Our emphasis.

<sup>2</sup>This assumption is consistent with the “we’re all equal” worldview [9]

But first we must understand what accounts for this antinomy. Two possible explanations involve the phenomena of label bias and selection bias. Label bias occurs when the process that produces the labels (e.g., a manual annotation process or a decision making process such as hiring) are influenced by factors that are not particularly germane to the determination of the label value, and thus might differ from the ideal labels, whatever they should have been. Accuracy measured against any such biased labels should be considered carefully with a grain of salt. Selection bias occurs when selecting a subsample of the data in such a way that happens to introduce unexpected correlations, say, between a protected attribute and the target label. Training data, which is usually derived via selection from a larger set of unlabeled data and subsequently frozen in time, is especially prone to this problem.

If pressed to couch the above discussion in a formal framework such as probably approximately correct (PAC) learning, we would say that we have a data distribution  $\mathcal{D}$  and labeling function  $f$ , either of which could be biased. For example, due to selection bias we might have a flawed data distribution  $\mathcal{D}'$  and due to label bias we might have a flawed labeling function  $f'$ . This leads to four regimes: the data distribution is biased ( $\mathcal{D}'$ ) or not ( $\mathcal{D}$ ) and the labeling function is biased ( $f'$ ) or not ( $f$ ). Many theoretical works in fair machine learning consider the regime in which neither is biased, and many empirical works—due in part to the unavailability of an unbiased  $f$ —draw conclusions assuming the regime in which neither is biased. But many forms of unfairness arise exactly because one or both of these are biased: hence the dualism in fair machine learning. In this work, we assume that some of the unfairness might arrise because we are actually in one of the other three regimes.

In this paper we account for both label and selection bias in our evaluations and show that when taken into consideration, that certain definitions of fairness and accuracy are not always in tension. Since we do not have access to the unbiased, unobserved ground truth labels in practice, we instead simulate datasets in tightly controlled ways such that, for example, it exposes the actual unbiased labels for evaluation. Encouraged by theoretical results on semi-supervised PAC learning that state that these techniques will be successful exactly when there is compatibility between some semi-supervised signal and the data distribution [1] and the success of GE [16, 10], we also introduce and study a semi-supervised method that exploits fairness constraints expressed over large quantities of unlabeled data to build better classifiers. Indeed, we find that as fairness improves, so does accuracy. Moreover, we find that like other fairness methods, the semi-supervised approach can successfully overcome label bias; but unlike other fairness methods, it can also overcome selection bias on the training set.

## 2 Related work

Somehow, the idea that fairness and accuracy are not always in tension is both obvious and inconspicuous (but nevertheless of practical significance). The idea appears obvious because we assume the unobserved unbiased ground-truth to be fair, and then limit our hypotheses to the fair region of the space, and then claim that fairness improves accuracy. At this level of generality, it even appears to beg the question, but note that not all fair hypotheses are accurate since in the case we consider a perfectly random classifier is also fair. Moreover, the noise on the observed biased labels with which we train the classifier is diametrically opposed to the unobserved label. Thus even under our assumptions, it is not a foregone conclusion that improving fairness improves accuracy. Rather, our assumption merely leaves open the possibility for this to happen. The finding is inconspicuous in the sense that, as mentioned earlier, there is a preponderance of work investigating this trade-off yet label bias appears to have gone unnoticed: very few papers (e.g., [8, 20]) mention the fact that the labels against which we evaluate are often biased (unfairly against a protected attribute) in the very same way as the unfair classifier trained on them [11, 7, 3, 8, 20, 4, 14, 2, 13, 17]. It may be the case that label-bias is so obvious to most authors that it does not even occur to them to mention it; howbeit, the conspicuous absence of label-bias from papers on fairness perniciously pervades real-world discussions underlying the decisions about how to balance the trade-off between fairness and accuracy. Thus, we believe this finding to be of practical importance and worthy of highlighting.

While uncommon, some papers do indeed mention label-bias, including recent work that considers the largely hypothetical case: if we have access to unbiased labels, then we can propose a better way of evaluating fairness with “disparate mistreatment” [20]. However, their emphasis is on new fairness metrics, not on its tradeoff with accuracy. Other work mentions the problem of label bias in passing, lamenting that it is difficult to account for in practice because we “only have biased data” and thus we “cannot evaluate our classifiers against an unbiased ground truth” and so achieving parity requires

that “one must be willing to reduce accuracy” ([8]). They overcome the lack of unbiased labels via data simulation, a strategy we also employ.

Congruent with our findings, others have noted that the fairness-accuracy tension is not as bad as it seems. Recent work correctly remarks that while there is a tradeoff between fairness and goodness of fit on the training set, that “it does not [necessarily] introduce a tension” since a reduction in model complexity via fairness constraints might act as a regularizer and improve generalization [2]. This is a very interesting remark, but it could have gone even further and addressed generalization with respect to the unbiased labels, which we study in this work.

In recent theoretical work, the authors’ propose a “construct space” in which the observed data might differ from some unobserved actual truth about the world [9]. While they investigate many different notions of fairness, they do not address accuracy. The construct space provides a promising theoretical framework for our work, but we save such analysis for another day. Other analytical work studies the trade-off between fairness and accuracy as a function of the amount of statistical dependence between the target class and protected attribute, concluding that only “in the other extreme” of perfect independence that “we can have maximum accuracy and fairness simultaneously” [17]. This “extreme” is none other than the “we’re all equal” assumption, which we believe to be perfectly reasonable in many situations. Further, note that this theoretical “maximum” may not be achievable in practice due to imperfect classifiers trained on incomplete, noisy data, or in the context of the phenomena mentioned herein, and hence there is still an opportunity to improve both.

It is worth thinking about the problems of selection and label bias with respect to an existing fairness datasets such as COMPAS, for which the labels are sometimes treated as if they are the unbiased ground truth [20]. Consider that the people in the COMPAS data had been selected from a specific county in Florida with its concomitant pattern of policing, during a specific period of time (2013-2014), meeting a specific set of criteria such as being scored during a specific stage within the judicial system. Each one of these “selections” opens the door for selection bias to introduce unintentional correlations. Indeed, recent work demonstrates that the data is skewed with respect to age, which acts as a confounding variable in existing analysis [18]. Moreover, while not exactly label-bias, the variable indicating recidivism is only partially observed since it considers only a two-year window and assumes that no crime goes uncaught.

Finally, we emphasize that our findings do not imply that the existing theories and conclusions discussed in the literature are incorrect. On the contrary, these works are in fact both sound and relevant. The different conclusions then are explained by the consideration of different types of data bias (or lack thereof) as well as the underlying assumptions, and our assumptions may not always apply [3]. If there differences between groups based on a protected attribute (e.g., due to selection bias), then enforcing fairness could indeed hurt accuracy. We do not address the degree to which one assumption applies to a particular problem or dataset in this paper. Thus, just like in statistical significance testing, it remains up to the discretion of the discerning practitioner to determine if our (or their) set assumptions reasonably apply to the situation in question, and if the assumptions do not, then our (or their) conclusions do not apply, and should be properly rejected as irrelevant to that data.

### 3 Background

**Fairness and bias types** We consider two types of biases that lead to unfair machine learning models: label bias and selection bias. Label bias is when the observed binary class labels, say, on the training and testing set, are influenced by protected attributes. For example, the labels in the dataset might be the result of yes/no hiring decisions for each job candidate. It is known that this hiring process is sometimes biased with respect to protected attributes such as race, age or gender. Since decisions might be influenced by protected attributes that on the contrary should have no bearing on the class label, this implies there is a hypothetical set of latent unobserved labels corresponding to decisions that were not influenced by these attributes. We denote these unobserved unbiased labels as  $z$ . We denote the observed biased labels as  $y$ . Typically, we only have access to the latter for training and testing our models.

Selection bias (skew) occurs when the method employed to select some subset of the overall population biases or skews the subset in unexpected ways. This can occur if selecting based on some attribute that inadvertently correlates with a protected class or the target labels. Training sets are particularly vulnerable to such bias because, for the sake of manual labeling expedience, they are

meager subsamples of the original unlabeled data points. Moreover, this problem is compounded since most available labeled datasets are statically frozen in time and are thus also selectionally biased along the axis of time. For example, in natural language processing (NLP), the particular topical subjects or the entities mentioned in newswire articles change over time: the entities discussed in political discourse today are very different from a decade ago and new topics must emerge to keep pace with the *dernier cri* [19]. And, as we continue to make progress in reducing discrimination, the discrepancy between the training data of the past and the available data of the present will increasingly differ w.r.t. to selection bias. Indeed, selection bias might manifest itself in a way such that on the relatively small training set, the data examples that were selected for labeling happen to show bias against the protected class. It is with this manifestation of selection bias that we are most concerned, and that we study in the current work.

**Illustrative example: learning fair sectors** Consider the problem of learning circular sectors of the unit disk with the following attributes: the domain set  $\mathcal{X}$  is the unit disk, the label set  $\mathcal{Y}$  is  $\{0, 1\}$ , the data generation model  $\mathcal{D}$  is an arbitrary density on  $\mathcal{X}$ , the labeling function  $f$  is an arbitrary partition of  $\mathcal{X}$  into two circular sectors, the hypothesis class  $\mathcal{H}$  is the set of all partitions of  $\mathcal{X}$  into two circular sectors. Samples from  $\mathcal{D}$  are points on the unit disk with location  $re^{i\phi}$  where  $\phi \in [0, 360)$  and  $r \in [0, 1]$ . We represent a circular sector as a pair of angles  $(\mu, \theta)$  and defined as the circular sector from angle  $(\mu - \theta)\%360$  to angle  $(\mu + \theta)\%360$  that contains the point  $e^{i\mu}$ . The labeling function  $f$  partitions the disk in two circular sectors  $f^{-1}(0)$  and  $f^{-1}(1)$  and we will refer to the former as the negative circular sector and the latter as the positive circular sector. Note that for any labeling function  $f$ , we have  $f \in \mathcal{H}$  and so the realizable assumption holds.

Due to label bias, the labeling function  $f$  might be biased ( $f'$ ) as shown in Figure 1. Here, the total positive area according to  $f$  is given by the area in green and red, but because of label bias  $f'$  only considers points in green as positive. Hence, as demonstrated in Figure 2, an empirical risk minimization (ERM) algorithm will learn a sector (dotted lines) that appears accurate with respect to  $f'$ , but is much less accurate with respect to  $f$ . If we had prior knowledge that the ratio of the positive sector and negative sector should be some constant  $k$ , perhaps we could exploit this and improve the ERM solution. We might term such an alternative empirical fairness maximization (EFM) (or fair ERM [5]), and in this paper, we present a semi-supervised EFM algorithm to exploit such fairness knowledge as a constraint on unlabeled data. This example is fully developed in appendix B.

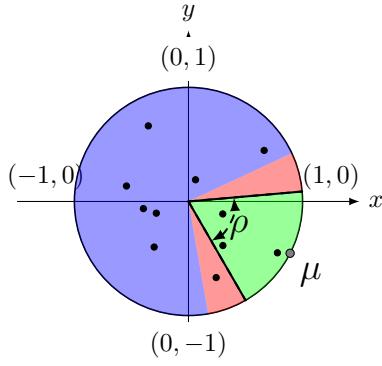


Figure 1

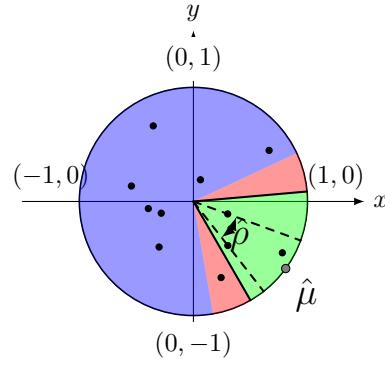


Figure 2

**Semi-supervised classification** A binary classifier<sup>3</sup>  $g_w : \mathbb{R}^k \rightarrow \{0, 1\}$  parameterized by a set of weights  $w \in \mathbb{R}^k$  is a function from a  $k$  dimensional real valued feature space, which is often in practice binary, to a binary class label. A probabilistic model  $p_w(\cdot|x)$  parameterized by (the very same)  $w$  underlies the classifier in the sense that we perform classification by selecting the class label (0 or 1) that maximizes the conditional probability of the label  $y$  given the data point  $x$

$$g_w(x) = \operatorname{argmax}_{y \in \{0, 1\}} p_w(y|x) \quad (1)$$

---

<sup>3</sup>For ease of explication, we consider the task of binary classification, though our method can easily be generalized to multiclass classification, multilabel classification, or more complex structured prediction settings.

We can then train the classifier in the usual supervised manner by training the underlying model to assign high probability to each observed label  $y_i$  in the training data  $\mathcal{D}_{\text{tr}} = \{\langle x_i, y_i \rangle \mid i = 1 \dots n\}$  given the corresponding example  $x_i$ , by minimizing the negative log likelihood:

$$\hat{w} = \operatorname{argmin}_{w \in \mathbb{R}^k} \sum_{\langle x_i, y_i \rangle \in \mathcal{D}_{\text{tr}}} -\log p_w(y_i | x_i) \quad (2)$$

We can extend the above objective function to include unlabeled data  $\mathcal{D}_{\text{un}} = \{x_i\}_{i=1}^n$  to make the classifier semi-supervised. In particular, we add a new term to the loss,  $\mathcal{C}(\mathcal{D}_{\text{un}}, w)$ , with a weight  $\eta$  to control the influence of the unlabeled data over the learned weights:

$$\hat{w} = \operatorname{argmin}_{w \in \mathbb{R}^k} \left( \sum_{\langle x_i, y_i \rangle \in \mathcal{D}_{\text{tr}}} -\log p_w(y_i | x_i) \right) + \eta \mathcal{C}(\mathcal{D}_{\text{un}}, w) \quad (3)$$

The key question is how to define the loss term  $\mathcal{C}$  over the unlabeled data in such a way that improves over our classifier.

## 4 Approach

Apropos the foregoing discussion, we propose to employ fairness in the part of the loss function that exploits the unlabeled data. There are of course many definitions of fairness proposed in the literature that we could adapt for this purpose, but for now we focus on a particular type of group fairness constraint derived from the *statistical parity* of selection rates. Although this definition has (rightfully) been criticized, it has also (rightfully) been advocated in the literature and it underlies legal definitions such as the 4/5ths rule in U.S. law [6, 8, 21]. For the purpose of this paper, we do not wish to enter the fray on this particular matter.

More formally, let  $S = \{x_i\}_{i=1}^n$  be a set of  $n$  unlabeled examples, then the selection rate of the classifier  $g_w$  is  $\bar{g}_w(S) = \frac{1}{n} \sum_{x_i \in S} g_w(x_i)$ . If we partition our data ( $\mathcal{D}_{\text{un}}$ ) into the protected ( $\mathcal{D}_{\text{un}}^P$ ) and unprotected ( $\mathcal{D}_{\text{un}}^U$ ) partitions such that  $\mathcal{D}_{\text{un}} = \mathcal{D}_{\text{un}}^P \cup \mathcal{D}_{\text{un}}^U$ , then we want the selection rate ratio

$$\frac{\bar{g}_w(\mathcal{D}_{\text{un}}^P)}{\bar{g}_w(\mathcal{D}_{\text{un}}^U)} \quad (4)$$

to be as close to one as possible. However, to make the problem more amenable to optimization via stochastic gradient descent, we relax this definition of fairness to make it differentiable with respect to  $w$ . In particular, analogous to  $\bar{g}_w(S)$ , define  $\bar{p}_w(S) = \frac{1}{n} \sum_{x_i \in S} p_w(y=1 | x_i)$  to be the average probability of the set when assigning each example  $x_i$  to the positive class  $y_i = 1$ . Then, the group fairness loss over the unlabeled data — which when plugged into Equation 3 yields an instantiation of the proposed semisupervised training technique discussed herein — is

$$\mathcal{C}(\mathcal{D}_{\text{un}}, w) = (\bar{p}_w(\mathcal{D}_{\text{un}}^P) - \bar{p}_w(\mathcal{D}_{\text{un}}^U))^2 \quad (5)$$

Parity is achieved at zero, which intuitively encodes that overall, the probability of assigning one group to the positive class should on average be the same as assigning the other group to the positive class. This loss has the important property that it is differentiable with respect to  $w$  so we can optimize it with stochastic gradient descent, along with the supervised term of the objective, making it easy to implement in existing toolkits such as Scikit-Learn, PyTorch or TensorFlow.

## 5 Experiments

In this section we investigate the relationship between fairness and accuracy under conditions in which we can account for (and vary) the amount of label bias, selection bias, and the extent to which the classifiers enforce fairness. Typically, accuracy is measured against the ground truth labels on the test set, which inconspicuously possesses the very same label bias as the training set. In this typical evaluation setting, if we train a set of classifiers that differ only in the extent to which their training objective functions enforce fairness, and then record their respective fairness and accuracy scores on a test set with such label bias, we see that increased fairness is achieved at the expense

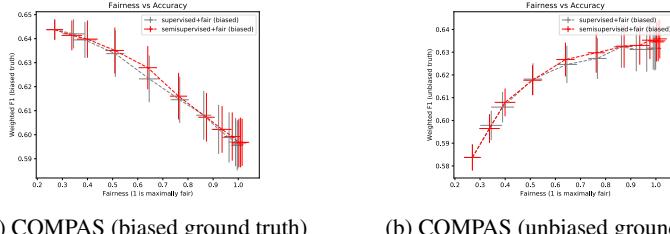


Figure 3: Accuracy vs. fairness on simulated ( $\beta=0.25$ ) COMPAS (assumption hold).

of accuracy (Figure 3a). However, because the labels are biased, we must immediately assume that the corresponding accuracy measurements are also biased. Therefore, we are crucially interested in evaluating accuracy on the *unbiased ground truth labels*, which are devoid of any such label bias. Since we do not have access to the unbiased ground truth labels of real-world datasets, we must instead rely upon data simulation. We discuss the details later, but for now, assume we could evaluate on such data. In Figure 3a, we evaluate the same set of classifiers as before, but this time measure accuracy with respect to the unbiased ground truth labels. We see the exact opposite pattern: classifiers that are more fair are also more accurate. With the gist of our results and experimental strategy in hand, we are now ready to describe the assumptions, data simulator, and systems to undertake a more comprehensive empirical investigation.

**Assumptions** We make a set of assumptions that we encode directly into the probabilistic data generator, explained in more detail below. For example, we encode the “we’re all equal assumption” by making the unbiased labels statistically independent of the protected class [9]. If these assumptions do not hold in a particular situation, then our conclusions may not apply. We describe the assumptions in more detail below and in the appendix.

**Data** Our experiments require datasets with points of the form  $\mathcal{D} = \{x, \rho, z, y\}$  in which  $x$  is the vector of unprotected attributes,  $\rho$  is the binary protected attribute,  $z$  is the (typically unobserved) label that has no label bias and  $y$  is the (typically observed) label that may have label bias. Since  $z$  is unobserved — and even if it were available, we would still want to vary the severity of label bias for experimental evaluation — we must rely upon data simulation [8]. We therefore assume that the biased labels are generated from the unbiased labels via a probabilistic model  $g$  and assume that  $y \sim g(y|z, \rho, x, \beta)$  where  $\beta$  is a parameter of the model that controls the probability of label bias occurring. Now we have two options for generating datasets of our desired form, we can either (a) simulate the dataset entirely from scratch from a probabilistic model of the joint distribution  $P(x, \rho, z, y) = g(y|z, \rho, x, \beta)P(z, \rho, x)P(\beta)$ , or we can (b) begin with an existing dataset, declare by fiat that the labels have no label bias (and are thus observed after all) and then augment the data with a set of biased labels sampled from  $g(y|z, \rho, x, \beta)$ .

For data of type (a) we generate the features and labels (both biased and unbiased) entirely from scratch with the Bayesian network in Figure 7 (Appendix A.2). For this data, we explicitly enforce the following statistical assumptions:  $z, x \perp\!\!\!\perp \rho, y \perp\!\!\!\perp \rho, z \perp\!\!\!\perp x, y \perp\!\!\!\perp z$ . A parameter  $\beta$  controls the amount of label bias;  $\sigma$  controls the amount of selection bias, which can break some assumptions. For data of type (b) we begin with the COMPAS data, treat the two-year recidivism labels as the unbiased ground-truth  $z$  and then apply our model of label bias to produce the biased labels  $y \sim g(z|y, \rho, x, \beta)$  [15]. Since the “we’re all equal” assumption does not hold for COMPAS data we also create a second type of test data in which we enforce demographic parity via subsampling so that our assumption holds (see Appendix A.3).

**Systems, baselines and evaluations** We study the behavior of the following classification systems. A traditional supervised classifier trained on biased label data, a supervised classifier trained on unbiased label data (this in some sense is an ideal model, but not possible in practice because we do not have access to the unbiased labels in practice), a random baseline in which labels are sampled according to the biased label distribution in the training data, and three fair classifiers. The first fair classification method is an in-processing classifier that employs our fairness constraint, but as a regularizer on the training data instead of the unlabeled data. The resulting classifier is similar

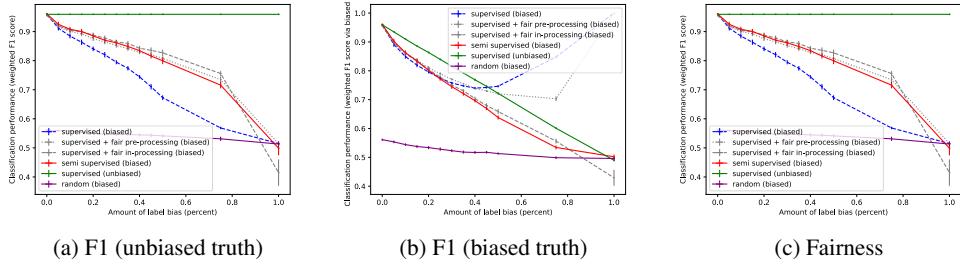


Figure 4: Classifier accuracy (F1) and fairness as a function of the amount of label bias.

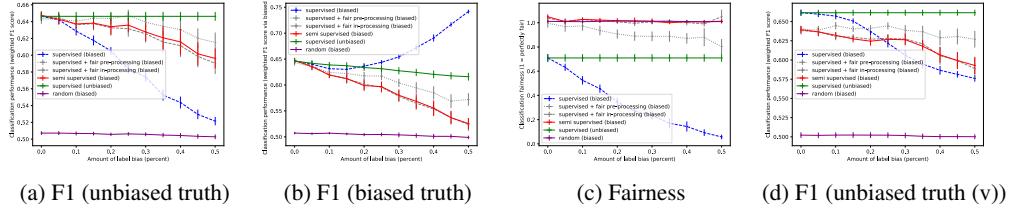


Figure 5: Varying label bias on COMPAS (assumption holds, except in 5d).

to the prejudice remover, but with a slightly different loss [12]. The second fair classifier is a supervised logistic regression trained using the “reweighing” pre-processing method [11]. The final fair classifier, which we introduce in this paper, is a semi-supervised classifier that utilizes the fairness loss (Equation 5) on the unlabeled data.

We assess fairness with a group metric that computes the ratio of the selection rates of the protected and unprotected class, as we defined in Equation 4. A score of one is considered perfectly fair. To assess ‘accuracy’ we compute the weighted macro F1, which is the macro average weighted by the relative portion of examples belonging to the positive and negative classes. We evaluate F1 with respect to both the biased labels and the unbiased labels. We always report the mean and standard error of these various metrics computed over ten experiments with ten randomly generated datasets (or in the case of COMPAS, ten random splits).

### 5.1 Experiment 1: Label Bias

In this experiment we investigate the relationship between fairness and accuracy for each classification method as we vary the amount of label bias. All classifiers except the unbiased baseline are trained on biased labels. If we evaluate the classifiers on the biased labels as in Figure 4b (data simulated from scratch) or Figure 5b (COMPAS data) we see that the classifiers that achieve high fairness (close to one, as seen in Figure 4c&5c) sometimes degrade the (biased) F1 accuracy as commonly seen in the literature. On the other hand, if we evaluate the classifiers on the unbiased labels as in Figure 4a&5a, we see that fairness and accuracy are in accord: the classifiers that achieve high fairness achieve better accuracy than the fairness-agnostic supervised baseline. The gap between the fair and unfair classifiers increases as label bias increases. We also evaluate the classifiers on COMPAS data that violates the “we’re all equal” assumption. In this case, the fairness classifiers are enforcing something untrue about the data, and thus fairness initially degrades accuracy (Figure 5d). However, as the amount of label bias increases, eventually there comes a point at which fairness once again improves accuracy (possibly because the amount of label bias exceeds the amount of other forms of bias).

### 5.2 Experiment 2: Selection Bias

We repeat the experiment from the last section, but this time fixing label bias ( $\beta = 0.2$ ) and subjecting the training data to various amounts of selection bias by lowering the probability that a data example with a positive label is assigned to the protected class. This introduces correlations in the training set between the protected class and the input features as well as correlations with both the unbiased and

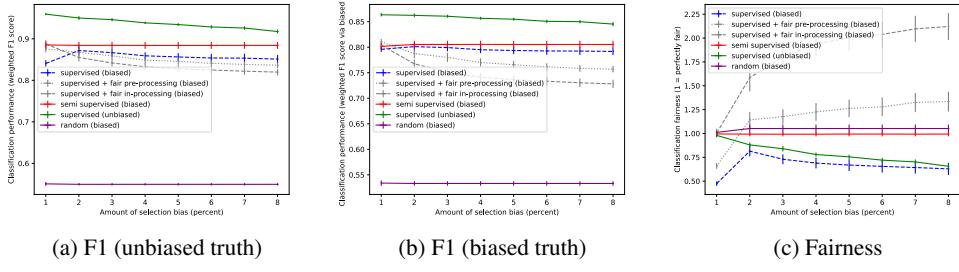


Figure 6: Classifier accuracy (F1) and fairness as a function of the amount of selection bias.

biased labels. These correlations do not exist in the test set and unlabeled set which we assume do not suffer from selection bias. We vary selection bias along the abscissa while keeping the label bias at a constant 20%, and report the same metrics as before. Results are in Figure 6. The main findings are that (a) the results are consistent with the theory that fairness and accuracy are in accord and (b) that the semi-supervised method successfully harnesses unlabeled data to correct for the selection and label bias in the training data (while the inprocessing fairness method succumbs to the difference in data distribution between training and testing). Let us now look at these findings in more detail and in the context of the other baselines.

Interestingly, the fairness-agnostic classifiers and two of the fairness-aware classifiers (in- and pre-processing) all succumb to selection bias, but in opposite ways (Figure 6c). The fairness-agnostic classifier learns the correlation between the protected attribute and the label and is unfair to the protected class. In contrast, the two supervised fair classifiers, for which fairness is enforced with statistics of the *training set* both learn to overcompensate and are unfair to the unprotected class (its fairness curve is above 1). In both cases, as selection bias increases, so does unfairness and this results in a concomitant loss in accuracy (when evaluated not only against the unbiased labels (Figure 6a), *but also against the biased labels* (Figure 6b)), indicating that fairness and accuracy are in accord. Finally, let us direct our attention to the performance of the proposed semi-supervised method by examining the same figures (Figure 6c). Now we see that regardless of the amount of selection bias, the semi-supervised method successfully harnesses the unbiased unlabeled data to rectify it, as seen by the flat fairness curve achieving a nearly perfect 1 (Figure 6c). Moreover, this improvement in fairness over the supervised baseline (biased trained) is associated with a corresponding increase in accuracy relative to that same baseline (Figures 6a & 6b), regardless of whether it is evaluated with respect to biased (20% label-bias) or unbiased labels (0% label-bias). Note that the “we’re all equal” assumption is violated as soon as we evaluate against the biased labels. Moreover, the label-bias induces a correlation between the protected class and the target label, which is a common assumption for analysis showing that fairness and accuracy are in tension [17]. Yet, the beneficial relationship between accuracy and fairness is unsullied by the incorrect assumption in this particular case.

## 6 Conclusion

We studied the relationship between fairness and accuracy while controlling for label and selection bias and found that under certain conditions the relationship is not a trade-off but rather one that is mutually beneficial: fairness and accuracy improve together. We focused on demographic parity in this paper, but the ideas emphasized in this work, especially label bias, have potentially serious implications for other notions of fairness that go beyond even their relationship with accuracy. In particular, recent ways of assessing fairness such as disparate mistreatment, equal odds and equal opportunity involve error rates as measured against labeled data. Label bias raises questions about the reliability of such measures and investigating such questions — about how label bias affects fairness and whether this causes fairness methods to undercompensate or overcompensate — is an important direction of future work. Other future directions would be to develop more complex models of label and selection bias for particular domains so we can better understand the relationship between fairness and accuracy in these domains.

## 7 Acknowledgements

We thank the anonymous reviewers for their constructive feedback and helpful suggestions on how to strengthen the paper.

## References

- [1] Maria-Florina Balcan and Avrim Blum. An augmented pac model for semi-supervised learning. In Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien, editors, *Semi-Supervised Learning*, chapter 21. MIT Press, 2006.
- [2] Richard Berk, Hoda Heidari, Shahin Jabbari, Matthew Joseph, Michael Kearns, Jamie H. Morgenstern, Seth Neel, and Aaron Roth. A convex framework for fair regression. *CoRR*, abs/1706.02409, 2017.
- [3] Alexandra Chouldechova. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *CoRR*, abs/1703.00056, 2016.
- [4] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. Algorithmic decision making and the cost of fairness. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 797–806, New York, NY, USA, 2017. ACM.
- [5] Michele Donini, Luca Oneto, Shai Ben-David, John Shawe-Taylor, and Massimiliano Pontil. Empirical risk minimization under fairness constraints. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 2796–2806, USA, 2018. Curran Associates Inc.
- [6] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 214–226, New York, NY, USA, 2012. ACM.
- [7] Michael Feldman, Sorelle A. Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 259–268, New York, NY, USA, 2015. ACM.
- [8] Benjamin Fish, Jeremy Kun, and Ádám Dániel Lelkes. A confidence-based approach for balancing fairness and accuracy. In *SDM*, 2016.
- [9] Sorelle A. Friedler, Carlos Eduardo Scheidegger, and Suresh Venkatasubramanian. On the (im)possibility of fairness. *CoRR*, abs/1609.07236, 2016.
- [10] Kuzman Ganchev, João Graça, Jennifer Gillenwater, and Ben Taskar. Posterior regularization for structured latent variable models. *J. Mach. Learn. Res.*, 11:2001–2049, August 2010.
- [11] Faisal Kamiran and Toon Calders. Data preprocessing techniques for classification without discrimination. *Knowl. Inf. Syst.*, 33(1):1–33, October 2012.
- [12] Toshihiro Kamishima, Shotaro Akaho, and Jun Sakuma. Fairness-aware learning through regularization approach. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining Workshops*, ICDMW '11, pages 643–650, Washington, DC, USA, 2011. IEEE Computer Society.
- [13] Jon Kleinberg. Inherent trade-offs in algorithmic fairness. *SIGMETRICS Perform. Eval. Rev.*, 46(1):40–40, June 2018.
- [14] Jon M. Kleinberg, Sendhil Mullainathan, and Manish Raghavan. Inherent trade-offs in the fair determination of risk scores. In *ITCS*, 2017.
- [15] Jeff Larson, Surya Mattu, Lauuren Kirchner, and Julia Angwin. ProPublica COMPAS data. <https://github.com/propublica/compas-analysis>, 2016.

- [16] Gideon S. Mann and Andrew McCallum. Generalized expectation criteria for semi-supervised learning with weakly labeled data. *J. Mach. Learn. Res.*, 11:955–984, March 2010.
- [17] Aditya Krishna Menon and Robert C Williamson. The cost of fairness in binary classification. In Sorelle A. Friedler and Christo Wilson, editors, *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, volume 81 of *Proceedings of Machine Learning Research*, pages 107–118, New York, NY, USA, 23–24 Feb 2018. PMLR.
- [18] Cynthia Rudin, Caroline Wang, and Beau Coker. The age of secrecy and unfairness in recidivism prediction. *arXiv preprint arXiv:1811.00731*, 2018.
- [19] Xuerui Wang and Andrew McCallum. Topics over time: A non-markov continuous-time model of topical trends. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’06, pages 424–433, New York, NY, USA, 2006. ACM.
- [20] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P. Gummadi. Fairness beyond disparate treatment &#38; disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th International Conference on World Wide Web*, WWW ’17, pages 1171–1180, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee.
- [21] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez-Rodriguez, and Krishna P. Gummadi. Fairness constraints: Mechanisms for fair classification. In *AISTATS*, 2017.

# Gradient-based Inference for Networks with Output Constraints

Jay Yoon Lee \* † Sanket Vaibhav Mehta † Michael Wick\* ‡ Jean-Baptiste Tristan ‡ Jaime Carbonell †

## Abstract

Practitioners apply neural networks to increasingly complex problems in natural language processing, such as syntactic parsing and semantic role labeling that have rich output structures. Many such structured-prediction problems require deterministic constraints on the output values; for example, in sequence-to-sequence syntactic parsing, we require that the sequential outputs encode valid trees. While hidden units might capture such properties, the network is not always able to learn such constraints from the training data alone, and practitioners must then resort to post-processing. In this paper, we present an inference method for neural networks that enforces deterministic constraints on outputs without performing rule-based post-processing or expensive discrete search. Instead, in the spirit of gradient-based training, we enforce constraints with gradient-based *inference* (GBI): for each input at test-time, we nudge continuous model weights until the network’s unconstrained inference procedure generates an output that satisfies the constraints. We study the efficacy of GBI on three tasks with hard constraints: semantic role labeling, syntactic parsing, and sequence transduction. In each case, the algorithm not only satisfies constraints, but improves accuracy, even when the underlying network is state-of-the-art.

## 1 Introduction

Suppose we have trained a sequence-to-sequence (seq2seq) network (Cho et al. 2014; Sutskever, Vinyals, and Le 2014; Kumar et al. 2016) to perform a structured prediction task such as syntactic constituency parsing (Vinyals et al. 2015). We would like to apply this trained network to novel, unseen examples, but still require that the network’s outputs obey an appropriate set of problem specific hard-constraints; for example, that the output sequence encodes a valid parse tree. Enforcing these constraints is important because downstream tasks, such as relation extraction or coreference resolution typically assume that the constraints hold. Moreover, the constraints impart informative hypothesis-limiting restrictions about joint assignments to multiple output units, and

thus enforcing them holistically might cause a correct prediction for one subset of the outputs to beneficially influence another.

Unfortunately, there is no guarantee that the neural network will learn these constraints from the training data alone, especially if the training data volume is limited. Although in some cases, the outputs of state-of-the-art (SOTA) systems mostly obey the constraints for the test-set of the data on which they are tuned, in other cases they do not. In practice, the quality of neural networks are much lower when run on data in the wild (e.g., because small shifts in domain or genre change the underlying data distribution). In such cases, the problem of constraint violations becomes more significant.

This raises the question: how should we enforce hard constraints on the outputs of a neural network? We could perform expensive combinatorial discrete search over a large output space, or manually construct a list of post-processing rules for the particular problem domain of interest. Though, we might do even better if we continue to “train” the neural network at test-time to learn how to satisfy the constraints on each input. Such a learning procedure is applicable at test-time because learning constraints requires no labeled data: rather, we only require a function that measures the extent to which a predicted output violates a constraint.

In this paper, we present *gradient-based inference* (GBI), an inference method for neural networks that strongly favors respecting output constraints by adjusting the network’s weights at test-time, for each input. Given an appropriate function that measures the extent of a constraint violation, we can express the hard constraints as an optimization problem over the continuous weights and apply back-propagation to tune them. That is, by iteratively adjusting the weights so that the neural network becomes increasingly likely to produce an output configuration that obeys the desired constraints. Much like scoped-learning, the algorithm customizes the weights for each example at test-time (Blei, Bagnell, and McCallum 2002), but does so in a way to satisfy the constraints.

We study GBI on three tasks: semantic role labeling (SRL), syntactic constituency parsing and a synthetic sequence transduction problem and find that the algorithm performs favorably on all three tasks. In outline, our contributions on this paper are:

1. Propose a novel Gradient-Based Inference framework.

\* corresponding authors: jaylee@cs.cmu.edu, michael.wick@oracle.com

† Carnegie Mellon University, Pittsburgh, PA

‡ Oracle Labs, Burlington, MA.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

2. Verify that GBI performs well on various applications, thus providing strong evidence for the generality of the method.
3. Examine GBI across wide range of reference model performances and report its consistency.
4. Show that GBI also perform well on out-of-domain data.

For all the tasks, we find that GBI satisfies a large percentage of the constraints (up to 98%) and that in almost every case (out-of-domain data, state-of-the art networks, and even for the lower-quality networks), enforcing the constraints improves the accuracy. On SRL, for example, the method successfully injects truth-conveying side-information via constraints, improving SOTA network<sup>1</sup> by 1.03 F1 (Peters et al. 2018). This improvement happens to surpass a A\* algorithm for incorporating constraints while also being robust, in a way that A\* is not, to cases for which the side constraints are inconsistent with the labeled ground-truth.

## 2 Constraint-aware inference in neural networks

Our goal is to design an *approximate* optimization algorithm that is similar in spirit to Lagrangian relaxation in that we replace a complex constrained decoding objective with a simpler unconstrained objective that we can optimize with gradient descent (Koo et al. 2010; Rush et al. 2010; Rush and Collins 2012), but is better suited for non-linear non-convex optimization with global constraints that do not factorize over the outputs. Although the exposition in this section revolves around Lagrangian relaxation, we emphasize that the purpose is merely to provide intuition and motivate design choices.

### 2.1 Problem definition and motivation

Typically, a neural network parameterized by weights  $W$  is a function from an input  $\mathbf{x}$  to an output  $\mathbf{y}$ . The network has an associated compatibility function  $\Psi(\mathbf{y}; \mathbf{x}, W) \rightarrow \mathbb{R}_+$  that measures how likely an output  $\mathbf{y}$  is given an input  $\mathbf{x}$  under weights  $W$ . The goal of inference is to find an output that maximizes the compatibility function and this is usually accomplished efficiently with feed-forward greedy-decoding. In this work, we want to additionally enforce that the output values belong to a feasible set or grammar  $\mathcal{L}^\mathbf{x}$  that in general depends on the input. We are thus interested in the following optimization problem:

$$\begin{aligned} \max_{\mathbf{y}} \quad & \Psi(\mathbf{x}, \mathbf{y}, W) \\ \text{s. t.} \quad & \mathbf{y} \in \mathcal{L}^\mathbf{x} \end{aligned} \quad (1)$$

Simple greedy inference are no longer sufficient since the outputs might violate the global constraints (i.e.,  $\mathbf{y} \notin \mathcal{L}^\mathbf{x}$ ). Instead, suppose we had a function  $g(\mathbf{y}, \mathcal{L}^\mathbf{x}) \rightarrow \mathbb{R}_+$  that measures a loss between output  $\mathbf{y}$  and a grammar  $\mathcal{L}^\mathbf{x}$  such that  $g(\mathbf{y}, \mathcal{L}^\mathbf{x}) = 0$  if and only if there are no grammatical errors in  $\mathbf{y}$ . That is,  $g(\mathbf{y}, \mathcal{L}^\mathbf{x}) = 0$  for the feasible region and is strictly positive everywhere else. For example, if the

<sup>1</sup>Since our submission, the previous SOTA (Peters et al. 2018) in SRL on which we apply our technique has been advanced by 1.7 F1 points (Ouchi, Shindo, and Matsumoto 2018). However, this is a training time improvement which is orthogonal to our work.

feasible region is a CFL,  $g$  could be the *least errors count* function (Lyon 1974). We could then express the constraints as an equality constraint and minimize the Lagrangian:

$$\min_{\lambda} \max_{\mathbf{y}} \Psi(\mathbf{x}, \mathbf{y}, W) + \lambda g(\mathbf{y}, \mathcal{L}^\mathbf{x}) \quad (2)$$

However, this leads to optimization difficulties because there is just a single dual variable for our global constraint, resulting intractable problem and thus leading to brute-force trial and error search.

Instead, we might circumvent these issues if we optimize over a model parameters rather than a single dual variable. Intuitively, the purpose of the dual variables is to simply penalize the score of *infeasible* outputs that otherwise have a high score in the network, but happen to violate constraints. Similarly, network’s weights can control the compatibility of the output configurations with the input. By properly adjusting the weights, we can affect the outcome of inference by removing mass from invalid outputs—in much the same way a dual variable affects the outcome of inference. Unlike a single dual variable however, the network expresses a *different* penalty weight for each output. And, because the weights are typically tied across space (e.g., CNNs) or time (e.g., RNNs) the weights are likely to generalize across related outputs. As a result, lowering the compatibility function for a single invalid output has the potential effect of lowering the compatibility for an entire family of related, invalid outputs; enabling faster search. In the next subsection, we propose a novel approach that utilizes the amount of constraint violation as part of the objective function so that we can adjust the model parameters to search for a constraint-satisfying output efficiently.

### 2.2 Algorithm

Instead of solving the aforementioned impractical optimization problem, we propose to optimize a “dual” set of model parameters  $W_\lambda$  over the constraint function while regularizing  $W_\lambda$  to stay close to the originally learned weights  $W$ . The objective function is as follows:

$$\begin{aligned} \min_{W_\lambda} \quad & \Psi(\mathbf{x}, \hat{\mathbf{y}}, W_\lambda) g(\hat{\mathbf{y}}, \mathcal{L}^\mathbf{x}) + \alpha \|W - W_\lambda\|_2 \\ \text{where} \quad & \hat{\mathbf{y}} = \underset{\mathbf{y}}{\operatorname{argmax}} \Psi(\mathbf{x}, \mathbf{y}, W_\lambda) \end{aligned} \quad (3)$$

Although this objective deviates from the original optimization problem, it is reasonable because by definition of the constraint loss  $g(\cdot)$ , the global minima must correspond to outputs that satisfy all constraints. Further, we expect to find high-probability optima if we initialize  $W_\lambda = W$ . Moreover, the objective is intuitive: if there is a constraint violation in  $\hat{\mathbf{y}}$  then  $g(\cdot) > 0$  and the gradient will lower the compatibility of  $\hat{\mathbf{y}}$  to make it less likely. Otherwise,  $g(\cdot) = 0$  and the gradient of the energy is zero and we leave the compatibility of  $\hat{\mathbf{y}}$  unchanged. Crucially, the optimization problem yields computationally efficient subroutines that we exploit in the optimization algorithm.

To optimize the objective, the algorithm alternates maximization to find  $\hat{\mathbf{y}}$  and minimization w.r.t.  $W_\lambda$  (Algorithm 1). In particular, we first approximate the maximization step by employing the neural network’s inference procedure (e.g.,

greedy decoding, beam-search, or Viterbi decoding) to find the  $\hat{\mathbf{y}}$  that approximately maximizes  $\Psi$ , which ignores the constraint loss  $g$ . Then, given a fixed  $\hat{\mathbf{y}}$ , we minimize the objective with respect to the  $W_\lambda$  by performing stochastic gradient descent (SGD). Since  $\hat{\mathbf{y}}$  is fixed, the constraint loss term becomes a constant in the gradient; thus, making it easier to employ external black-box constraint losses (such as those based on compilers) that may not be differentiable. As a remark, note the similarity to REINFORCE (Williams 1992): the decoder outputs as actions and the constraint-loss as a negative reward. However, GBI does not try to reduce *expected* reward and terminates upon discovery of an output that satisfies all constraints. Furthermore, GBI also works on sequence-tagging problem, SRL (Section 4.1), where next output does not depend on the current output, which is far from REINFORCE setting.

---

**Algorithm 1** Constrained inference for neural nets

---

```

Inputs: test instance  $\mathbf{x}$ , input specific CFL  $\mathcal{L}^\mathbf{x}$ , pretrained
weights  $W$ 
 $W_\lambda \leftarrow W$  #reset instance-specific weights
while not converged do
     $\mathbf{y} \leftarrow f(\mathbf{x}; W_\lambda)$  #perform inference using weights  $W_\lambda$ 
     $\nabla \leftarrow g(\mathbf{y}, \mathcal{L}^\mathbf{x}) \frac{\partial}{\partial W_\lambda} \Psi(\mathbf{x}, \mathbf{y}, W_\lambda) + \alpha \frac{W - W_\lambda}{\|W - W_\lambda\|_2}$  #compute constraint loss
     $W_\lambda \leftarrow W_\lambda - \eta \nabla$  #update instance-specific weights
    with SGD or a variant thereof
end while

```

---

### 3 Applications

There are multiple applications that involve hard-constraints and we provide two illustrative examples that we later employ as case-studies in our experiments: SRL and syntactic parsing. The former exemplifies a case in which external knowledge encoded as hard constraints conveys beneficial side information to the original task of interest while the latter studies a case in which hard constraints are inherent to the task of interest. Finally, we briefly mention sequence transduction as framework in which constraints may arise. Of course, constraints may in general arise for a variety of different reasons, depending on the situation. We provide an example-based case studies per application in Appendix A, B.

#### 3.1 Semantic Role Labeling

As a first illustrative example, consider SRL. SRL focuses on identifying shallow semantic information about phrases. For example, in the sentence “it is really like this, just look at the bus sign” the goal is to tag the two arguments given “is” as the verb predicate: “it” as its first argument and the prepositional phrase “like this” as its second argument. Traditionally SRL is addressed as a sequence labeling problem, in which the input is the sequence of tokens and the output are BIO-encoded class labels representing both the regimentation of tokens into contiguous segments and their semantic roles.

Note that the parse tree for the sentence might provide constraints that could assist with the SRL task. In particu-

lar, each node of the parse tree represents a contiguous segment of tokens that could be a candidate for a semantic role. Therefore, we can include as side-information constraints that force the BIO-encoded class labeling to produce segments of text that each agree with some segment of text expressed by a node in the parse tree.<sup>2</sup> To continue with our example, the original SRL sequence-labeling might incorrectly label “really like this” as the second argument rather than “like this.” Since according to the parse tree “really” is part of the verb phrase, thus while the tree contains the spans “is really like this” and “like this” it does not contain the span “really like this.” The hope is that enforcing the BIO labeling to agree with the actual parse spans would benefit SRL. Based on the experiments, this is indeed the case, and our hypothetical example is actually a real data-case from our experiments, which we describe later. The  $g(\mathbf{y}, \mathcal{L}^\mathbf{x})$  for SRL factorizes into per-span constraints  $g_i$ . For  $i$ th span  $s_i$ , if  $s_i$  is consistent with any node in the parse tree,  $g_i(s_i, \mathcal{L}^\mathbf{x}) = 0$ , otherwise  $g_i(s_i, \mathcal{L}^\mathbf{x}) = 1/n_{s_i}$  where  $n_{s_i}$  is defined as the number of tokens in  $s_i$ . Overall,  $\Psi(\mathbf{x}, \hat{\mathbf{y}}, W_\lambda)g(\hat{\mathbf{y}}, \mathcal{L}^\mathbf{x}) = \sum_{i=1}^k g(s_i, \mathcal{L}^\mathbf{x})\Psi(\mathbf{x}, s_i, W_\lambda)$  where  $k$  is number of spans on output  $\hat{\mathbf{y}}$ .

#### 3.2 Syntactic parsing

As a second illustrative example, consider a structured prediction problem of syntactic parsing in which the goal is to input a sentence comprising a sequence of tokens and output a tree describing the grammatical parse of the sentence. Syntactic parsing is a separate but complementary task to SRL. While SRL focuses on semantic information, syntactic parsing focuses on identifying relatively deep syntax tree structures. One way to model the problem with neural networks is to linearize the representation of the parse tree and then employ the familiar seq2seq model (Vinyals et al. 2015). Let us suppose we linearize the tree using a sequence of shift (s) and reduce (r, r!) commands that control an implicit shift reduce parser. Intuitively, these commands describe the exact instructions for converting the input sentence into a complete parse tree: the interpretation of the symbol s is that we shift an input token onto the stack and the interpretation of the symbol r is that we start (or continue) reducing (popping) the top elements of the stack, the interpretation of a third symbol ! is that we stop reducing and push the reduced result back onto the stack. Thus, given an input sentence and an output sequence of shift-reduce commands, we can deterministically recover the tree by simulating a shift reduce parser. For example, the sequence ssrr!ssr!rr!rr! encodes a type-free version of the parse tree (S (NP the ball) (VP is (NP red))) for the input sentence “the ball is red”. It is easy to recover the tree structure from the input sentence and the output commands by simulating the shift reduce parser. Of course in practice, reduce commands include the standard parts of speech as types (NP, VP, etc).

Note that for output sequences to form a valid tree over the input, the sequence must satisfy a number of constraints. First, the number of shifts must equal the number of input tokens

---

<sup>2</sup>The ground-truth parse spans do not always agree with the SRL spans, leading to imperfect side information.

$m_x$ , otherwise either the tree would not cover the entire input sentence or the tree would contain spurious terminal symbols. Second, the parser cannot issue a reduce command if there are no items left on the stack. Third, the number of reduces must be sufficient to leave just a single item, the root node, on the stack. The constraint loss  $g(y, \mathcal{L}^x)$  for this task simply counts the errors of each of the three types. (Appendix C.2)

As a minor remark, note that other encodings of trees, such as bracketing (of which the Penn Tree Bank’s S-expressions are an example), are more commonly used as output representations for seq2seq parsing (*ibid.*). However, the shift-reduce representation described in the foregoing paragraphs is isomorphic to the bracketing representations and as we get similar model performance to single seq2seq mode on the same data (*ibid.*), we chose the former representation to facilitate constraint analysis. Although output representations sometimes matter, for example, BIO vs BILOU encoding of sequence labelings, the difference is usually minor (Ratnayov and Roth 2009), and breakthroughs in sequence labeling have been perennially advanced under both representations. Thus, for now, we embrace the shift reduce representation as a legitimate alternative to bracketing, *pari passu*.

### 3.3 Synthetic sequence transduction

Finally, although not a specific application per se, we also consider sequence transduction as it provides a framework conducive to studying simple artificial languages with appropriately designed properties. A sequence transducer  $T : \mathcal{L}_S \rightarrow \mathcal{L}_T$  is a function from a source sequence to a target sequence. As done in previous work, we consider a known  $T$  to generate input/output training examples and train a seq2seq network to learn  $T$  on that data (Grefenstette et al. 2015). The constraint is simply that the output must belong to  $\mathcal{L}_T$  and also respect problem-specific conditions that may arise from the application of  $T$  on the input sentence. We study a simple case in Section 4.3.

## 4 Experiments

In this section we study our algorithm on three different tasks: SRL, syntactic constituency parsing and a synthetic sequence transduction task. All three tasks require hard constraints, but they play a different role in each. In the sequence transduction tasks they force the output to belong to a particular input-dependent regular expression, in SRL, constraints provide side-information about possible true-spans and in syntactic parsing, constraints ensure that the outputs encode valid trees. While the SRL task involves more traditional recurrent neural networks that have exactly one output per input token, the parsing and transduction tasks provide an opportunity to study the algorithm on various seq2seq networks.

We are interested in answering the following questions (Q1) how well does the neural network learn the constraints from data (Q2) for cases in which the network is unable to learn the constraints perfect, can GBI actually enforce the constraints (Q3) does GBI enforce constraints without compromising the quality of the network’s output. To more thoroughly investigate Q2 and Q3, we also consider: (Q4) is the behavior of the method sensitive to the reference network

performance, and (Q5) does GBI also work on out-of-domain dataset. Q3 is particularly important because we adjust the weights of the network at test-time and this may lead to unexpected behavior. Q5 deals with our original motivation of using structured prediction to enhance performance on the out-of-domain data.

To address various proposed questions, we define some terminologies to measure how well the model is doing in terms of constraints. To address (Q1) we measure the *failure-rate* (i.e., the ratio of test sentences for which the network infers an output that fails to fully satisfy the constraints). To address (Q2) we evaluate our method on the *failure-set* (i.e., the set of output sentences for which the original network produces constraint-violating outputs) and measure our method’s *conversion rate*; that is, the percentage of failures for which our method is able to completely satisfy the constraints (or “convert”). Finally, to address (Q3), we evaluate the quality (e.g., accuracy or F1) of the output predictions on the network’s *failure-set* both before and after applying our method.

### 4.1 Semantic Role Labeling

We employ the AllenNLP (Gardner et al. 2017) SRL network with ELMo embeddings, which is essentially a multi-layer highway bi-LSTM that produces BIO output predictions for each input token (Peters et al. 2018). For data we use OntoNotes v5.0, which has ground-truth for both SRL and syntactic parsing (Pradhan et al. 2013). We evaluate GBI on the test-set (25.6k examples), out of which consistent parse information is available for 81.25% examples (we only include side-information in terms of constraints for this subset).

We repeat the same experimental procedure over multiple networks, SRL-X, with varying amounts (X%) of training dataset. From Table 1, we see that GBI is able to convert 42.25 % of failure set, and this boosts the overall F1 measure by 1.23 point over the SOTA network (SRL-100) that does not incorporate the constraints (they report 84.6 F1, we obtain a similar 84.4 F1 with their network, and achieve 85.63 after enforcing constraints with our inference). Further, to address (Q1) we measure the sentence-level *failure rate* as well as span-level *disagreement rate* (i.e., the ratio of predicted spans in a sentence that disagree with the spans implied by the true syntactic parse of the sentence). To address (Q2) we evaluate our method on the *failure set* (i.e., the set of sentences for which disagreement rate is nonzero) and measure our method’s avg. disagreement rate. Finally, to address (Q3), we evaluate the quality (F1 and Exact Match) of the output predictions on the network’s *failure-set* both before and after applying our method. From Table 1, we can see that by applying GBI on SRL-100, the avg. disagreement rate on failure set goes down from 44.85% to 24.92% which results in an improvement of 11.7 F1 and 19.90% in terms of exact match on the same set. These improvements answers Q1-3 favorably.

To enforce constraints during inference, He et al. proposed to employ constrained-A\* decoding. For the sake of fair comparison with GBI, we consider A\* decoding, as used in (He et al. 2017) and report results for SRL-X networks. We see from Table 1, that GBI procedure consistently out-performs

Network	Failure rate(%)	Inference	Conv rate(%)	Failure set						Test set	
				Average (%) Disagreement		F1		Exact Match (%)		F1	
				before	after	before	after	before	after	before	after
SRL-100	9.82	GBI A*	42.25 40.40	44.85 33.91	24.92 32.32	48.00	59.70 (+11.7) 48.83 (+0.83)	0.0 13.79	19.90 16.12	84.40 83.55	85.63 (+1.23) 84.51 (+0.11)
SRL-70	10.54	GBI A*	46.22 44.42	45.54 32.32	23.02 32.17	47.81	59.37 (+11.56) 50.49 (+2.68)	0.0 19.45	19.57 15.15	83.55 82.57	84.83 (+1.28) 83.90 (+0.35)
SRL-40	11.06	GBI A*	47.89 44.74	45.71 32.17	22.42 32.17	46.53	58.83 (+12.3) 46.53 (+2.88)	0.0 12.28	19.45 12.28	78.56 78.56	84.03 (+1.46) 82.98 (+0.41)
SRL-10	14.15	GBI A*	44.28 43.66	47.14 32.80	24.88 32.80	44.19	54.78 (+10.59) 45.93 (+1.74)	0.0 15.28	15.28 12.28	78.56 78.56	80.18 (+1.62) 78.87 (+0.31)
SRL-1	21.90	GBI A*	52.85 48.96	50.38 30.28	21.45 30.28	37.90	49.00 (+11.10) 41.59 (+3.69)	0.0 11.25	12.83 11.25	67.28 67.97	69.97 (+2.69) 67.97 (+0.69)

Table 1: Comparison of the GBI vs. A\* inference procedure for SRL. We report avg. disagreement rate, F1-scores and exact match for *failure set* (columns 5-10) and F1-score for whole test set (last 2 columns). Also we report performances on wide range of reference models SRL-X, where X denotes % of dataset used for training. We employ Viterbi decoding as a base inference strategy (before) and apply GBI (after) in combination with Viterbi.

name	F1		hyper-parameters			data (%)
	BS-9	greedy	hidden	layers	dropout	
Net1	87.58	87.31	128	3	0.5	100%
Net2	86.63	86.54	128	3	0.2	100%
Net3	81.26	78.32	172	3	no	100%
Net4	78.14	74.53	128	3	no	75%
Net5	71.54	67.80	128	3	no	25%

Table 2: Parsing Networks with various performances (BS-9 means beam size 9). Net1, 2 are trained on GNMT seq2seq model whereas Net3-5 are trained on lower-resource and simpler seq2seq model to test GBI on wide range of model performances.

Net	Infer method	Failure (n/2415)	Conv rate	F1 (Failure set)	
				before	after
Net3	Greedy	317	79.81	65.62	68.79 (+3.14)
	Beam 2	206	87.38	66.61	71.15 (+4.54)
	Beam 5	160	87.50	67.5	71.38 (+3.88)
	Beam 9	153	91.50	68.66	71.69 (+3.03)
Net4	Greedy	611	88.05	62.17	64.49 (+2.32)
	Beam 2	419	94.27	65.40	66.65 (+1.25)
	Beam 5	368	92.66	67.18	69.4 (+2.22)
	Beam 9	360	93.89	67.83	70.64 (+2.81)
Net5	Greedy	886	69.86	58.47	60.41 (+1.94)
	Beam 2	602	82.89	60.45	61.35 (+0.90)
	Beam 5	546	81.50	61.43	63.25 (+1.82)
	Beam 9	552	80.62	61.64	62.98 (+1.34)

Table 4: Evaluation of the GBI on simpler, low-resource seq2seq networks. In here, we also evaluate whether GBI can be used in combination with different inference techniques: greedy and beam search of various width.

Table 3: Evaluation of the GBI on syntactic parsing using GNMT seq2seq. Note that GBI without beam search performs higher than BS-9 on Table 2.

A\* decoding on all evaluation metrics, thus, proving superiority of our approach.

## 4.2 Syntactic parsing

We now turn to a different task and network: syntactic constituency parsing. We investigate the behavior of the constraint inference algorithm on the shift-reduce parsing task described in Section 3. We transform the Wall Street Journal (WSJ) portion of the Penn Tree Bank (PTB) into shift-reduce commands in which each reduce command has a phrase-type (e.g., noun-phrase or verb-phrase) (Marcus et al. 1999). We employ the traditional split of the data with section 22 for dev, section 23 for test, and remaining sections 01-21 for training. We evaluate on the test set with evalb<sup>3</sup> F1.

<sup>3</sup><http://nlp.cs.nyu.edu/evalb/>

In each experiment, we learn a seq2seq network on a training set and then evaluate the network directly on the test set using a traditional inference algorithm to perform the decoding (either greedy decoding or beam-search).

In order to study our algorithm on a wide range of accuracy regimes (section 4.4), we train many networks with different hyper-parameters producing models of various quality, from high to low, using the standard split of the WSJ portion of the PTB. In total, we train five networks Net1-5 for this study, that we describe below.

We train our two best baseline models (Net1,2) using a highly competitive seq2seq architecture for machine translation, GNMT (Wu et al. 2016) with F1 scores, 86.78 and 87.33, respectively. And, to study wider range of accuracies, we train a simpler architecture with different hyper parameters and obtain nets ( Net3-5). For all models, we employ Glorot initialization, and basic attention (Bahdanau, Cho, and Bengio 2014). See Table 2 for a summary of the networks, hyper-parameters, and their performance.

We report the behavior of the constraint-satisfaction

method on Table 3 for Net1-2, and on Table 4 for Net3-5. Across all the experimental conditions (Table 3, 4), the conversion rates are high, often above 80 and sometimes above 90 supporting Q2. Note that beam search alone can also increase constraint satisfaction with conversion rate going as high as 51.74% (164/317) in case of Net3 with beam size 9. However, as the quality of the model increases, the conversion rate became minuscule; in case of Net1,2 the conversion rate was less than 14% with beam 9; Net1 converted 26 out of 187 and Net2 converted 1 out of 287 instances from failure set.

In order to address question Q3—the ability of our approach to satisfy constraints without negatively affecting output quality—we measure the F1 scores on the failure-sets both before and after applying the constraint satisfaction algorithm. Since F1 is only defined on valid trees, we employ heuristic post-processing to ensure all outputs are valid.

Note that an improvement on the failure-set guarantees an improvement on the entire test-set since our method produces the exact same outputs as the baseline for examples that do not initially violate any constraints. Consequently, for example, the GNMT network improves (Net2) on the failure-set from 73.54 to 79.68 F1, resulting in an overall improvement from 86.54 to 87.57 F1 (entire test-set). These improvements are similar to those we observe in the SRL task, and provide additional evidence for answering Q1-3 favorably.

We also measure how many iterations of our algorithm it takes to convert the examples that have constraint-violations. Across all conditions, it takes 5–7 steps to convert 25% of the outputs, 6–20 steps to convert 50%, 15–57 steps to convert 80%, and 55–84 steps to convert 95%.

### 4.3 Simple Transduction Experiment

In our final experiment we focus on a simple sequence transduction task in which we find that despite learning the training data perfectly, the network fails to learn the constraint in a way that generalizes to the test set.

For our task, we choose a simple transducer, similar to those studied in recent work (Grefenstette et al. 2015). The source language  $\mathcal{L}_S$  is  $(az|bz)^*$  and the target language  $\mathcal{L}_T$  is  $(aaa|zb)^*$ . The transducer is defined to map occurrences of  $az$  in the source string to  $aaa$  in the target string, and occurrences of  $bz$  in the source string to  $zb$  in the target string. For example,  $T(bzazbz) \mapsto zbaaabz$ . The training set comprises 1934 sequences of length 2–20 and the test set contain sentences of lengths 21–24. We employ shorter sentences for training to require generalization to longer sentences at test time.

We employ a 32 hidden unit single-layered, attention-less, seq2seq LSTM in which the decoder LSTM inputs the final encoder state at each decoder time-step. The network achieves perfect train accuracy while learning the rules of the target grammar  $\mathcal{L}_T$  perfectly, even on the test-set. However, the network fails to learn the input-specific constraint that the number of  $a$ 's in the output should be three times the number of  $a$ 's in the input. This illustrates how a network might rote-memorize constraints rather than learn the rule in a way that generalizes. Thus, enforcing constraints at test-time is important. To satisfy constraints, we employ GBI with a constraint

loss  $g$ , a length-normalized quadratic  $(3x_a - y_a)^2 / (m + n)$  that is zero when the number of  $a$ 's in the output ( $y_a$ ) is exactly three times the number in the input ( $x_a$ ) with  $m, n$  denoting input, output, respectively. GBI achieves a conversion rate of 65.2% after 100 iterations, while also improving the accuracy on the failure-set from 75.2% to 82.4%. This synthetic experiment provides additional evidence in support of Q2 and Q3, on a simpler small-capacity network.

### 4.4 GBI on wide range of reference models

The foregoing experimental results provide evidence that GBI is a viable method for enforcing constraints. However, we hitherto study GBI on high quality reference networks such as SRL-100. To further bolster our conclusions, we now direct our investigation towards lower quality networks to understand GBI's viability under a broader quality spectrum. We ask, how sensitive is GBI to the reference network's performance (Q4)? To this end, we train poorer quality networks by restricting the amount of available training data or employing simpler architectures.

For *SRL*, we simulate low-resource models by limiting the training data portion to 1%, 10%, 40%, and 70% resulting in F1 score range of 67.28–83.55. Similarly, for *syntactic parsing*, we train additional low-quality models Net3-5 with a simpler uni-directional encoders/decoders, and on different training data portion of 25%, 75%, and 100%. (Table 2). We evaluate GBI on each of them in Table 1, 4 and find further evidence in support of favorable answers to Q2 (satisfying constraints) and Q3 (improving F1 accuracy) by favorably answering Q4. Moreover, while not reported fully due to page limits, we examined both tasks over 20 experiments with different baseline networks in combination to different inference strategies, and we found GBI favorable in all but one case (but by just 0.04 compared to employing post-processing).

We also study whether GBI is compatible with better underlying discrete search algorithms, in particular beam search for seq2seq. As we seen in column 2 of Table 4, that although beam-search improves the F1 score and reduces the percentage of violating constraints, GBI further improves over beam-search when using the latter in the inner-loop as the decoding procedure. In conclusion, improving the underlying inference procedure has the effect of decreasing the number of violating outputs, but GBI is still very much effective on this increasingly small set, despite it intuitively representing more difficult cases that even eludes constraint satisfaction on beam search inference.

### 4.5 Experiments on out-of-domain data

Previously, we saw how GBI performs well even when the underlying network is of lower quality. We now investigate GBI on actual out-of-domain data for which the model quality can suffer. For *SRL*, we train a SOTA network with ELMo embedding on the NewsWire (NW) section of the OntoNotes v5.0 English PropBank corpus and then test on the other genres provided in the corpus: BC, BN, PT, TC, WB. The failure rate on the within genre data (test set of NW) is 18.10%. We can see from Table 5, the failure rate for the NW trained *SRL* network in general is higher for out-of-genre data with highest being 26.86% for BC (vs. 18.10% NW). Further, by

Genre	Syntactic Parsing				SRL			
	Failure rate (%)	Conversion rate (%)	F1 on failure set		Failure rate (%)	Conversion rate (%)	F1 on failure set	
			before	after			before	after
Broadcast Conversation (BC)	19.3	98.8	56.4	59.0 (+2.6)	26.86	53.88	39.72	52.4 (+12.68)
Broadcast News (BN)	11.7	98.1	63.2	68.8 (+5.6)	18.51	55.19	39.28	50.58 (+11.3)
Pivot Corpus (PT)	9.8	97.8	71.4	75.8 (+4.4)	10.01	62.34	47.19	63.69 (+16.5)
Telephone Conversation (TC)	10.1	86.2	56.9	57.6. (+0.7)	19.09	54.62	47.7	58.04 (+10.34)
Weblogs (WB)	17.6	95.3	62.0	63.2 (+1.2)	20.32	44.13	47.6	57.39 (+9.39)

Table 5: Evaluation of syntactic parser and SRL system on out-of-domain data. F1 scores are reported on the *failure set*. SRL model was trained on NW and the syntactic parser was trained on WSJ Section on OntoNote v5.0. Except PT, which is new and old Testament, all failure rate on out-domain data is higher than that of in-domain (11.9% for parsing and 18.1% for SRL) as suspected. The table shows that GBI can be successfully applied to resolve performance degradation on out-of-domain data.

enforcing constraints, we see significant gains on failure set in terms of F1 score across all genres (ranging from 9.39–16.5 F1), thus, providing additional evidences for answering Q5.

As we did for SRL, we train a GMNT seq2seq model on the WSJ NW section in OntoNotes v5.0 Treebank<sup>4</sup> which shares the same genre classification with PropBank. The F1 on the within-genre data (test set of WSJ) is 85.03, but the F1 on these genres is much lower, ranging from the mid-forties on BC (46.2–78.5 depending on the subcategory) to the low-eighties on BN (68.3–81.3. depending on the subcategory). Indeed, we find that overall the F1 is lower and in some cases, like WB, the failure rate is much higher (17.6% for WB vs. 11.9% for WSJ). Following the same experimental protocol as on the PTB data, we report the results in Table 5 (aggregating over all subcategories in each genre). We see that across all genres, the algorithm has high conversion rates (sometimes close to 100%), and that in each case, enforcing the constraints improves the F1. Again, we find support for Q2, Q3 and Q5.

#### 4.6 Robustness and Runtime analysis

We perform extra analysis on robustness and runtime for SRL in comparison with competing constrained-A\* decoding method. In Appendix D, we introduce case study of noisy constraints where A\* performs worse than the baseline whereas GBI still improves it significantly, thus, showcasing its robustness to noisy constraints.

For runtime, GBI is slightly faster than A\* while the difference in terms of the runtime is unclear on smaller evaluation sets (see Appendix E). In the case study of noisy constraints, GBI is similar in runtime to A\*, but the runtime comparison is not important in the noisy constraint setting as A\* with noisy constraints simply hurts whereas GBI shows comparable gains with the noise-free constraint setting.

#### 5 Related work

Recent work has considered applying neural networks to structured prediction; for example, structured prediction energy networks (SPENs) (Belanger and McCallum 2016). SPENs incorporate soft-constraints via back-propagating an energy function into “relaxed” output variables. In contrast, we focus on hard-constraints and back-propagate into the

<sup>4</sup>The PTB (40k instances) and OntoNotes (30k instances) coverage of WSJ are slightly different.

weights that subsequently control the original non-relaxed output variables via inference. Separately, there has been interest in employing hard constraints to harness unlabeled data in training-time for simple classifications (Hu et al. 2016). Our work instead focuses on enforcing constraints at inference-time. More specifically, for SRL, previous work for enforcing syntactic and SRL specific constraints have focused on constrained A\* decoding (He et al. 2017) or integer linear programming (Punyakanok, Roth, and Yih 2008). For parsing, previous work in enforcing hard constraints has focused on post-processing (Vinyals et al. 2015) or building them into the decoder via sampling (Dyer et al. 2016) or search constraints (Wiseman and Rush 2016).

Finally, as previously mentioned, our method highly resembles dual decomposition and more generally Lagrangian relaxation for structured prediction (Koo et al. 2010; Rush et al. 2010; Rush and Collins 2012). In such techniques, it is assumed that a computationally efficient inference algorithm can maximize over a superset of the feasible region (this assumption parallels our case because unconstrained inference in the neural network is efficient, but might violate constraints). Then, the method employs gradient descent to concentrate this superset onto the feasible region. However, these techniques are not directly applicable to our non-linear problem with global constraints.

#### 6 Conclusion

We presented an algorithm for satisfying constraints in neural networks that avoids combinatorial search, but employs the network’s efficient unconstrained procedure as a black box to coax weights towards well-formed outputs. We evaluated the algorithm on three tasks including SOTA SRL, seq2seq parsing and found that GBI can successfully convert failure sets while also boosting the task performance. Accuracy in each of the three tasks was improved by respecting constraints. Additionally, for SRL, we employed GBI on top of a model trained with similar constraint enforcing loss as GBI’s (Mehta\*, Lee\*, and Carbonell 2018), and observe that the test-time optimization of GBI still significantly improves the model output whereas A\* does not. We believe this is because GBI enables to search proximity of the provided model weights while theoretical analysis on this hypothesis is left as a future work.

## References

- Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *CoRR, arXiv preprint arXiv:1409.0473*.
- Belanger, D., and McCallum, A. 2016. Structured prediction energy networks. In *International Conference on Machine Learning*.
- Blei, D. M.; Bagnell, A.; and McCallum, A. K. 2002. Learning with scope, with application to information extraction and classification. In *Uncertainty in Artificial Intelligence (UAI)*.
- Cho, K.; Van Merriënboer, B.; Gülcabay, Ç.; Bahdanau, D.; Bougares, F.; Schwenk, H.; and Bengio, Y. 2014. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734. Association for Computational Linguistics.
- Dyer, C.; Kuncoro, A.; Ballesteros, M.; and Smith, N. A. 2016. Recurrent neural network grammars. In *NAACL-HLT*, 199–209.
- Gardner, M.; Grus, J.; Neumann, M.; Tafjord, O.; Dasigi, P.; Liu, N. F.; Peters, M.; Schmitz, M.; and Zettlemoyer, L. S. 2017. AllenNLP: A deep semantic natural language processing platform.
- Grefenstette, E.; Hermann, K. M.; Suleyman, M.; and Blunsom, P. 2015. Learning to transduce with unbounded memory. In *Neural Information Processing Systems (NIPS)*.
- He, L.; Lee, K.; Lewis, M.; and Zettlemoyer, L. S. 2017. Deep semantic role labeling: What works and what’s next. In *ACL*.
- Hu, Z.; Ma, X.; Liu, Z.; Hovy, E.; and Xing, E. P. 2016. Harnessing deep neural networks with logical rules. In *Association for Computational Linguistics (ACL)*.
- Koo, T.; Rush, A. M.; Collins, M.; Jaakkola, T.; and Sontag, D. 2010. Dual decomposition for parsing with non-projective head automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 1288–1298. Association for Computational Linguistics.
- Kumar, A.; Irsoy, O.; Ondruska, P.; Iyyer, M.; Bradbury, J.; Gulrajani, I.; Zhong, V.; Paulus, R.; and Socher, R. 2016. Ask me anything: Dynamic memory networks for natural language processing. *Machine Learning* 1378–1387.
- Lyon, G. 1974. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Programming Languages* 17(1).
- Marcus, M. P.; Santorini, B.; Marcinkiewicz, M. A.; and Taylor, A. 1999. Treebank-3 ldc99t42 web download. In *Philadelphia: Linguistic Data Consortium*.
- Mehta\*, S. V.; Lee\*, J. Y.; and Carbonell, J. 2018. Towards semi-supervised learning for deep semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 4958–4963.
- Ouchi, H.; Shindo, H.; and Matsumoto, Y. 2018. A span selection model for semantic role labeling. In *EMNLP*, 1630–1642. Association for Computational Linguistics.
- Peters, M. E.; Neumann, M.; Iyyer, M.; Gardner, M.; Clark, C.; Lee, K.; and Zettlemoyer, L. 2018. Deep contextualized word representations. In *Proc. of NAACL*.
- Pradhan, S.; Moschitti, A.; Xue, N.; Ng, H. T.; Björkelund, A.; Uryupina, O.; Zhang, Y.; and Zhong, Z. 2013. Towards robust linguistic analysis using ontonotes. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, 143–152.
- Punyakanok, V.; Roth, D.; and Yih, W.-t. 2008. The importance of syntactic parsing and inference in semantic role labeling. *Computational Linguistics* 34(2):257–287.
- Ratinov, L., and Roth, D. 2009. Design challenges and misconceptions in named entity recognition. In *Computational Natural Language Learning (CoNLL)*.
- Rush, A. M., and Collins, M. 2012. A tutorial on dual decomposition and lagrangian relaxation for inference in natural language processing. *Journal of Artificial Intelligence Research* 45:305–362.
- Rush, A. M.; Sontag, D.; Collins, M.; and Jaakkola, T. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, 1–11. Association for Computational Linguistics.
- Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence-to-sequence learning with neural networks. In *Neural Information Processing Systems (NIPS)*.
- Vinyals, O.; Kaiser, L.; Koo, T.; Petrov, S.; Sutskever, I.; and Hinton, G. 2015. Grammar as a foreign language. In *NIPS*.
- Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8:229–256.
- Wiseman, S., and Rush, A. M. 2016. Sequence-to-sequence learning as beam-search optimization. In *Empirical Methods in Natural Language Processing*, 1296–1306.
- Wu, Y.; Schuster, M.; Chen, Z.; Le, Q. V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. 2016. Google’s neural machine translation system: Bridging the gap between human and machine translation. *CoRR, arXiv preprint arXiv:1609.08144*.

## Appendix

### A GBI vs. Constrained decoding

In Table 8 of Appendix B, we provide an example data-case that shows how our algorithm solves the initially violated shift-reduce parse output. For simplicity we omit the phrase-types of constituency parsing and display only on the shift (`s`), reduce (`r`) and stop reducing commands (`!`), and color them red if there is an error. The algorithm satisfies the constraint in just 12 iterations, and this results in a perfectly correct parse. What is interesting about this example is that the original network commits a parsing mistake early in the output sequence. This type of error is problematic for a naive decoder that greedily enforces constraints at each time-step. The reason is that the early mistake does not create a constraint violation until it is too late, at which point errors have already propagated to future time-steps and the greedy decoder must shift and reduce the last token into the current tree, creating additional spurious parse structures. In contrast, our method treats the constraints holistically, and uses it to correct the error made at the beginning of the parse. See Table 6 for a comparison of how the methods fix the constraints. Specifically, the constraint violation is that there were not enough shift and reduce commands to account for all the tokens in the sentence. Rather than fixing the constraint by inserting these extra commands at the end of the sequence as the greedy decoder must do, GBI inserts them at the beginning of the sequence where the initial mistake was made, thereby correcting the initial mistake. Moreover, this correction propagates to a mistake made later in the sequence (viz., the the sequence of three reduces after the four shifts) and fixes them too. This example provides evidence that GBI can indeed enforce constraints holistically and that doing so improves the output in a global sense.

$\langle \text{“So it’s a very mixed bag .”} \rangle \rightarrow \text{sssr!ssssrr!srrr!rr!ssrrrrrr!}$	
inference method	output
unconstrained-decoder	<code>ssr!sr!ssssrrr!</code> <code>rr!ssrrrrrr!</code>
constrained-decoder	<code>ssr!sr!ssssrrr!</code> <code>rr!ssrrrrrr!srr!</code>
our method	<code>sssr!ssssrr!srrr!rr!ssrrrrrr!</code>
true parse	<code>sssr!ssssrr!srrr!rr!ssrrrrrr!</code>

Table 6: A shift-reduce example for which the method successfully enforces constraints. The initial unconstrained decoder prematurely reduces “So it” into a phrase, missing the contracted verb “is.” Errors then propagate through the sequence culminating in the final token missing from the tree (a constraint violation). The constrained decoder is only able to deal with this at the end of the sequence, while our method is able to harness the constraint to correct the early errors.

### B Example-based case study

$\langle \text{“it is really like this , just look at the bus signs .”} \rangle \rightarrow \text{B-ARG1 B-V B-ARGM-ADV B-ARG2 I-ARG2 O O \dots O}$			
iteration	output	loss	accuracy
0	<code>B-ARG1 B-V B-ARG2 I-ARG2 I-ARG2 O O O O O O O</code>	0.012	50.0%
6	<code>B-ARG1 B-V B-ARG2 I-ARG2 I-ARG2 O O O O O O O</code>	0.049	50.0%
7	<code>B-ARG1 B-V B-ARGM-ADV B-ARG2 I-ARG2 O O O O O O O</code>	0.00	100.0%

Table 7: A semantic role labeling example for which the method successfully enforces syntactic constraints. The initial output has inconsistent span for token “really like this”. Enforcing the constraint not only corrects the number of agreeing spans, but also changes the semantic role “B-ARG2” to “B-ARGM-ADV” and “I-ARG2” to “B-ARG2”..

$\langle \text{“So it’s a very mixed bag .”} \rangle \rightarrow \text{sssr!ssssrr!srrr!rr!ssrrrrrr!}$			
iteration	output	loss	accuracy
0	<code>ssr!sr!ssssrrr!rr!ssrrrrrr!</code>	0.0857	33.3%
11	<code>ssr!sr!ssssrrr!rr!ssrrrrrr!</code>	0.0855	33.3%
12	<code>sssr!ssssrr!srrr!rr!ssrrrrrr!</code>	0.0000	100.0%

Table 8: A shift-reduce example for which the method successfully enforces constraints. The initial output has only nine shifts, but there are ten tokens in the input. Enforcing the constraint not only corrects the number of shifts to ten, but changes the implied tree structure to the correct tree.

azazbzazbzazbzazbzazbz → aaaaazbaaaazbzbaaaazbzazbz			
iteration	output	loss	accuracy
0	aaaaazbaaaazbzazbz <b>aaazb</b> zazbz <b>aaazb</b>	0.2472	66.7
1	aaaaazbaaaazbzazbz <b>aaazb</b> zazbz <b>aaazb</b>	0.2467	66.7
2	aaaaazbaaaazbzazbz <b>aaazb</b> zazbz <b>aaazb</b>	0.2462	66.7
3	aaaaazbaaaazbzazbz <b>aaazb</b> zazbz <b>aaazb</b>	0.0	100.0

Table 9: A sequence transduction example for which enforcing the constraints improves accuracy. Red indicates errors.

## C Constraint functions

Here we define the specific constraint loss function  $g(\mathbf{y}, \mathcal{L}^x)$  for each task. Note that a common theme is that we normalize the constraint loss by the length of the sequence so that it does not grow unbounded with sequence size. We recommend this normalization as we found that it generally improves performance.

### C.1 Semantic Role labeling

The  $g(\mathbf{y}, \mathcal{L}^x)$  for SRL factorizes into per-span constraints  $g_i$ . For  $i$ th span  $s_i$ , if  $s_i$  is consistent with any node in the parse tree,  $g_i(s_i, \mathcal{L}^x) = 0$ , otherwise  $g_i(s_i, \mathcal{L}^x) = 1/n_{s_i}$  where  $n_{s_i}$  is defined as the number of tokens in  $s_i$ . Overall,

$$\Psi(\mathbf{x}, \hat{\mathbf{y}}, W_\lambda) g(\hat{\mathbf{y}}, \mathcal{L}^x) = \sum_{i=1}^k g(s_i, \mathcal{L}^x) \Psi(\mathbf{x}, s_i, W_\lambda)$$

where  $k$  is number of spans on output  $\hat{\mathbf{y}}$ .

More precisely, for a span  $s$  to be “consistent with a parse node” we mean the following. Let  $t_i \in T$  be a node in the parse tree  $T$  and let  $s_i^t$  be the span of text implied by the descendants of the node  $t_i$ . Let  $S^T = \{s_i^t\}$  be the set of spans implied by all nodes in the parse tree  $T$ . We say that a span of text  $s$  is consistent with the parse tree  $T$  if and only if  $s \in S^T$ .

### C.2 Syntactic Parsing

Let  $m_{\mathbf{x}}, n$  be number of input, output tokens, respectively,  $\text{ct}_{i=0}^n(b(i))$  be the function that counts the number of times proposition  $b(i)$  is true for  $i = 1, \dots, n$ . Now, define the following loss

$$g(\mathbf{y}, \mathcal{L}^x) = \frac{1}{m_{\mathbf{x}} + n} \left\{ |m_{\mathbf{x}} - \text{ct}_{i=0}^n(y_i = s)| + \sum_i^n \max(0, \text{ct}_{j=0}^i(y_j = r) - \text{ct}_{j=0}^i(y_j \in \{s, !\})) \right\}.$$

The first term provides loss when the number or shifts equals the number of input tokens, the second term provides loss when attempting to reduce an empty stack and the third term provides loss when the number of reduces is not sufficient to attach every lexical item to the tree.

### C.3 Transduction

For the aforementioned transducer we chose for the experiment,  $\mathcal{L}_S$  is  $(az|bz)^*$  and the target language  $\mathcal{L}_T$  is  $(aaa|zb)^*$ . The transducer is defined to map occurrences of  $az$  in the source string to  $aaa$  in the target string, and occurrences of  $bz$  in the source string to  $zb$  in the target string.

For the provided transduction function, the number of  $a$ ’s in the output should be three times the number of  $a$ ’s in the input. To express this constraint, we define following constraint loss  $g$ , a length-normalized quadratic

$$g(\mathbf{y}, \mathcal{L}^x) = (3x_a - y_a)^2 / (m + n)$$

that is zero when the number of  $a$ ’s in the output ( $y_a$ ) is exactly three times the number in the input ( $x_a$ ) with  $m, n$  denoting input length, output length, respectively.

## D Analyzing the behavior of different inference procedures in the presence of noisy constraints

Table 10 reports the performance of GBI and  $A^*$  in the presence of noisy constraints. We can see that the overall performance (F1-score) for  $A^*$  drops drastically ( $-6.92$ ) in the presence of noisy constraints while we still see gains with GBI ( $+0.47$ ). We further analyze the improvement of GBI by looking at the precision and recall scores individually. We see that recall drops slightly for GBI which suggests that noisy constraints does inhibit predicting actual argument spans. On the other hand, we see that precision goes up significantly. After analyzing predicted argument spans, we noticed that GBI prefers to predict no argument spans instead of incorrect spans in the presence of noisy constraints which leads to an increase in precision. Thus, GBI provides flexibility in terms of strictness with enforcing constraints which makes it robust to noisy constraints. On the other hand, constrained- $A^*$  decoding algorithm is too strict when it comes to enforcing noisy constraints resulting in significant drop both in precision and recall.

Decoding	Precision	Recall	F1-score	Exact Match (%)
Viterbi	84.03	84.78	84.40	69.37
Noisy constraints				
A <sup>*</sup>	78.13 (-5.90)	76.85 (-7.93)	77.48 (-6.92)	58.30 (-11.70)
GBI	85.51 (+1.48)	84.25 (-0.53)	<b>84.87 (+0.47)</b>	68.45 (-0.92)
Noise-free constraints				
A <sup>*</sup>	84.19 (+0.16)	84.83 (+0.05)	84.51 (+0.11)	70.52 (+1.15)
GBI	85.39 (+1.36)	85.88 (+1.10)	<b>85.63 (+1.23)</b>	71.04 (+1.67)

Table 10: Comparison of different inference procedures: Viterbi, A<sup>\*</sup> (He et al. 2017) and GBI with noisy and noise-free constraints. Note that the (+/-) F1 are reported w.r.t Viterbi decoding on the same column.

## E Analyzing the runtime of different inference procedures with varying dataset sizes and genres

Network	Genre(s)	No. of examples	Failure rate (%)	Inference time (approx. mins)		
				Viterbi	GBI	A <sup>*</sup>
SRL-100	All	25.6k	9.82	109	288	377
SRL-NW	BC	4.9k	26.86	23	110	117
	BN	3.9K	18.51	18	64	100
	PT	2.8k	10.01	8	19	15
	TC	2.2k	19.01	5	23	20
	WB	2.3k	20.32	12	49	69

Table 11: Comparison of runtime for difference inference procedures in noise-free constraint setting: Viterbi, A<sup>\*</sup> (He et al. 2017) and GBI. For SRL-100 refer Table 1 and SRL-NW is a model trained on NW genre.

Table 11 reports the runtime for different inference procedures with varying dataset sizes. In general, we observe that GBI tends to be faster than A<sup>\*</sup>, especially when the dataset is large enough. One exception is the BC domain where GBI is just slightly faster than A<sup>\*</sup>. We hypothesize it might be due to the difficulty of the constraint violation due to its failure rate being higher than usual. GBI will search for the correct output for longer time (more iterations) if it is harder to find the solution.

Also note that we explicitly set max epochs for GBI after which it will stop iterating to avoid pathological cases. In our SRL experiments, we have set the max epochs to be 10 (GBI-10). To study its scalability, we ran GBI with max epochs set to 30 (GBI-30). In terms of the runtime, we do see a difference with GBI-30 taking 556 mins as opposed to GBI-10 taking 288 mins. However, we also get significantly better results with GBI-30 in terms overall F1 (+0.34), F1 on failure set (+3.4), exact match (+4.35%), and conversion rate (+11.24%) compared with GBI-10 at the cost of runtime increment.

# Using Butterfly-patterned Partial Sums to Draw from Discrete Distributions

GUY L. STEELE JR. and JEAN-BAPTISTE TRISTAN, Oracle Labs, USA

We describe a SIMD technique for drawing values from multiple discrete distributions, such as sampling from the random variables of a mixture model, that avoids computing a complete table of partial sums of the relative probabilities. A table of alternate (“butterfly-patterned”) form is faster to compute, making better use of coalesced memory accesses; from this table, complete partial sums are computed on the fly during a binary search. Measurements using CUDA 7.5 on an NVIDIA Titan Black GPU show that this technique makes an entire machine-learning application that uses a Latent Dirichlet Allocation topic model with 1,024 topics about 13% faster (when using single-precision floating-point data) or about 35% faster (when using double-precision floating-point data) than doing a straightforward matrix transposition after using coalesced accesses.

**CCS Concepts:** • Mathematics of computing → Gibbs sampling; • Theory of computation → Sorting and searching; Concurrent algorithms; • Computing methodologies → Concurrent algorithms; • Computer systems organization → Single instruction, multiple data;

**Additional Key Words and Phrases:** Butterfly, coalesced memory access, discrete distribution, GPU, latent Dirichlet allocation, LDA, machine learning, multithreading, memory bottleneck, parallel computing, random sampling, SIMD, transposed memory access

22

## ACM Reference format:

Guy L. Steele Jr. and Jean-Baptiste Tristan. 2019. Using Butterfly-patterned Partial Sums to Draw from Discrete Distributions. *ACM Trans. Parallel Comput.* 6, 4, Article 22 (November 2019), 30 pages.

<https://doi.org/10.1145/3365662>

## 1 OVERVIEW

The successful use of Graphics Processing Units (GPUs) to train neural networks is a great example of how machine learning can benefit from such massively parallel architecture. Generative probabilistic modeling [3] and associated inference methods (such as Monte Carlo methods) can also benefit. Indeed, authors such as Suchard et al. [26] and Lee et al. [14] have pointed out that many algorithms of interest are embarrassingly parallel. However, the potential for massively parallel computation is only the first step toward full use of GPU capacity. One bottleneck that such embarrassingly parallel algorithms run into is related to memory bandwidth; one must design key probabilistic primitives with such constraints in mind.

We address the case where parallel threads draw independently from distinct discrete distributions. This can arise when implementing any mixture model, and Latent Dirichlet Allocation (LDA) models in particular, which are probabilistic mixture models used to discover abstract “topics” in

---

Authors’ addresses: G. L. Steele Jr. and J.-B. Tristan, Oracle Labs, 35 Network Drive UBUR02-313, Burlington, MA, 01803, USA; emails: {steele, jean.baptiste.tristan}@oracle.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2019 Association for Computing Machinery.

2329-4949/2019/11-ART22 \$15.00

<https://doi.org/10.1145/3365662>

a collection of documents (a *corpus*) [4]. This model can be fitted (or “trained”) in an unsupervised fashion using sampling methods [2, chapter 11][7]. Each document is modeled as a distribution  $\theta$  over topics, and each word in a document is assumed to be drawn from a distribution  $\phi$  of words. Understanding the methods described in this article does not require a deep understanding of sampling algorithms for LDA. What is important is that each word in a corpus is associated with a so-called “latent” random variable [2, chapter 9], usually referred to as  $z$ , that takes on one of  $K$  integer values, indicating a topic to which the word belongs. Broadly speaking, the iterative training process works by tentatively choosing a topic (that is, sampling the random latent variable  $z$ ) for a given word using relative probabilities calculated from  $\theta$  and  $\phi$ , then updating  $\theta$  and  $\phi$  accordingly.

In this article, we focus on the step that, given a discrete distribution represented as a length- $K$  array  $a$  of relative probabilities, chooses an integer  $j$  such that  $0 \leq j < K$  in such a way that the probability of choosing any specific value  $j'$  is  $a_{j'}/\sigma$ , where  $\sigma$  is the sum of all elements of  $a$ . (We use zero-based indexing throughout this article.) This is easily done using a three-step process:

1. Normalize  $a$  (divide each entry by the sum of all entries).
2. Let  $u$  be chosen uniformly at random (or pseudorandomly) from the real interval  $[0, 1)$ .
3. Find the smallest index  $j$  such that the sum of all entries of  $a$  at or below index  $j$  is larger than  $u$ .

In practice, this sequence of steps may be improved to run much faster by (a) doing a bit of algebra to eliminate the division operations and (b) using a binary search:

1. Compute  $p$ , the prefix-sum of  $a$ , such that  $p_j = \sum_{i=0}^j a_i$ .
2. Choose  $u$  uniformly at random (or pseudorandomly) from the real interval  $[0, 1)$ .
3. Let  $u' = p_{(K-1)} \times u$ .
4. Use a binary search to find the smallest index  $j$  such that the entry at index  $p_j$  is larger than  $u'$ .

This works because all elements of  $a$  are nonnegative and therefore elements of  $p$  are monotonically nondecreasing.

Now suppose that we have many discrete distributions (thousands or millions) and wish to draw one sample from each, using a SIMD-style GPU. An obvious approach is to assign each distribution to a separate thread and have each thread execute the optimized (four-step) algorithm. However, in the context of the LDA application, a problem arises: when the threads fetch entries from their respective arrays (especially the  $\phi$  arrays), the values to be fetched will likely reside at unrelated locations in memory, resulting in poor memory-fetch performance. A standard technique is to have all the lanes in a warp (a group of threads being executed simultaneously by the SIMD engine) cooperate with each other. For concreteness, suppose there are 32 threads in a warp, and for simplicity, assume that each array  $a$  of relative probabilities is also of length 32. We can furthermore assume that the elements of any single  $a$  array are stored sequentially in memory (and therefore fit within a small number of cache lines); the problem arises solely because we cannot assume any specific relationship within memory among the 32  $a$  instances to be processed simultaneously. The technique that gets around this problem is *transposed memory access*: as the 32 elements are fetched for each of 32 instances of  $a$ , on each step  $j$  of 32 steps ( $0 \leq j < 32$ ), lane  $i$  ( $0 \leq i < 32$ ) fetches not  $a[i][j]$  but instead  $a[j][i]$ . (Compare, for example, the storage of floating-point numbers as “slicewise” rather than “fieldwise” in the architecture of the Connection Machine Model CM-2, so that 32 1-bit processors cooperate on each of 32 clock cycles to fetch and store an entire 32-bit floating-point value that logically belongs to just one of the 32 processors [10].) In words, on step  $j$  all 32 lanes fetch the 32 values needed by lane  $j$ ; as a result, on each memory cycle all

32 values being fetched are in a contiguous region of memory, allowing improved memory-fetch performance.

It is then necessary for the lanes to exchange information among themselves so that the rest of the algorithm may be carried out, including the summation arithmetic.

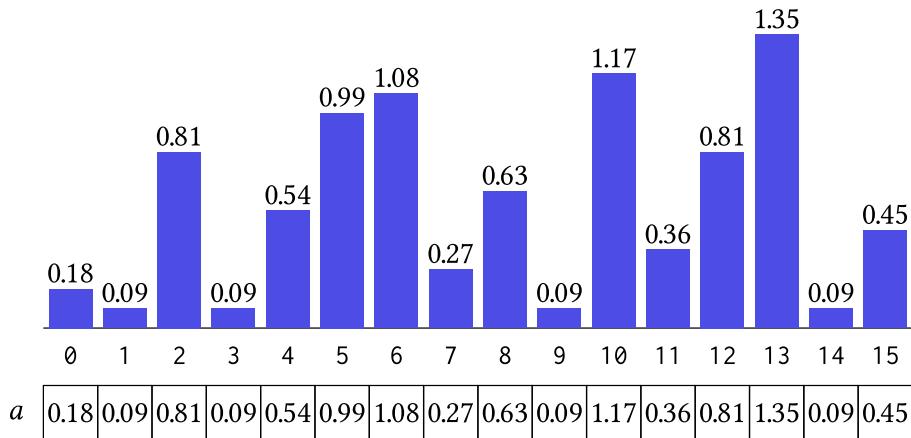
The novel contribution of this article is to observe and then exploit the fact that the binary search of the array  $p$  (which is computed from  $a$ ) does not access all entries of  $p$ ; in fact, for an array of size  $K$  it examines only about  $\log_2 K$  entries. Therefore, it is not necessary to compute all entries of the prefix-sum table. We present an alternate technique that computes a “butterfly-patterned” partial-sums table, using less computational and communication effort when implemented on a GPU; a modified binary search then uses this table to compute, on the fly, entries that would have been in the original complete prefix-sum table. This requires more work per table entry during the binary search, but because the search examines only a few table entries, the result is a net reduction in execution time. This technique may be effective for collapsed LDA Gibbs samplers [16, 27, 33] as well as uncollapsed samplers and may also be useful for GPU implementations of other algorithms [35] whose inner loops sample from discrete distributions.

This article is not about machine learning algorithms in general or LDA in particular; rather, we use an LDA application as a convenient and practical benchmark for evaluating data-parallel sampling algorithms. The specific LDA algorithm that we use is state-of-the-art [28] and had already been carefully tuned for speed before application of the techniques described in this article.

(This article is a revised and expanded version of Reference [25]. There is also video of a talk based on this material [22], and the slides for the talk are separately available [23].)

## 2 BACKGROUND

Suppose we are given a discrete distribution described as an array  $a$  of length  $K$  such that  $a_j$  is the relative probability that sampling the distribution will produce the value  $j$  ( $0 \leq j < K$ , and for purposes of illustration, we will use  $K = 16$ ):



From  $a$ , compute the prefix-sum array  $p$ :

```

1: let sum = 0.0
2: for k from 0 through K - 1 do
3:   sum += a[k]
4:   p[k] := sum
5: end for

```

$p$	0.18	0.27	1.08	1.17	1.71	2.70	3.78	4.05	4.68	4.77	5.94	6.30	7.11	8.46	8.55	9.00
-----	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------	------

To sample the distribution, a simple linear search can do the job:

```

1: let  $u$  = value chosen uniformly at random (or pseudorandomly) from [0.0, 1.0)
2: let  $u' = \text{sum} \times u$ 
3: let  $j = 0$ 
4: while  $j < K - 1$  and  $u' \geq p[j]$  do  $j += 1$ 
```

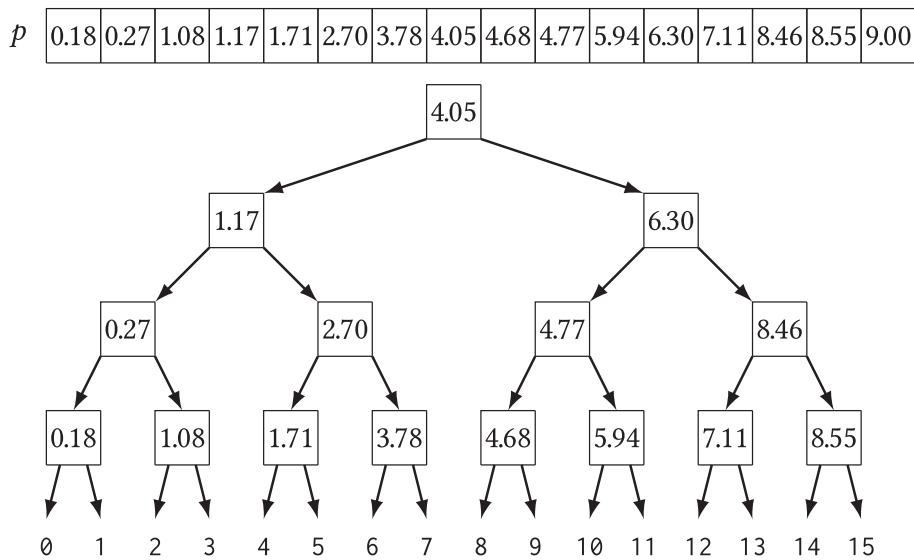
It is easy to see that, on completion of the search loop,  $j$  will be the index of the leftmost (lowest-indexed) element of  $p$  smaller than  $u'$ .

Alternatively, one may use a binary search:

```

1: let  $u$  = value chosen uniformly at random (or pseudorandomly) from [0.0, 1.0)
2: let  $u' = \text{sum} \times u$ 
3: let  $j = 0$ 
4: let  $k = K - 1$ 
5: while  $j < k$  do
6:   let  $mid = \lfloor \frac{j+k}{2} \rfloor$ 
7:   if  $u' < p[mid]$  then  $k := mid$  else  $j := mid + 1$ 
8: end while
```

A binary search on an array amounts to walking down a binary tree whose leaves are the array indices and whose internal nodes are labeled with all entries except the last. We can draw such a tree by starting with a drawing of the array and then displacing each entry (except the last) vertically:

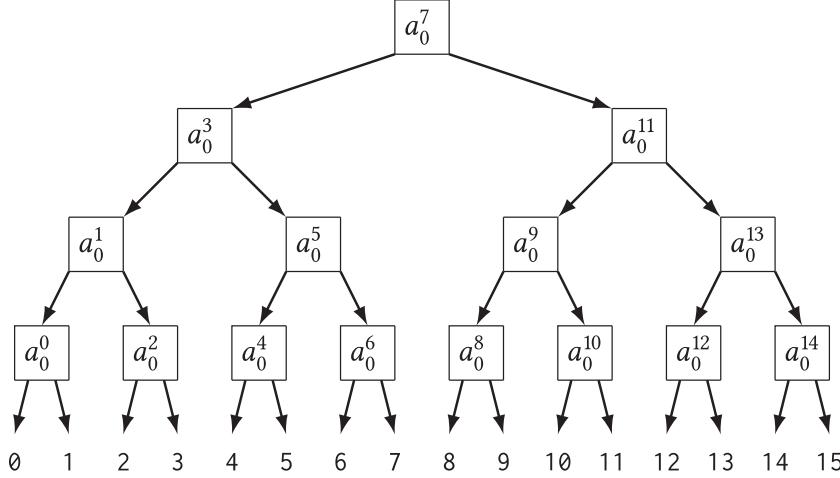


Starting from the root, we compare the quantity we are searching for to the label of each internal node that we encounter; if it is smaller, we descend to the left child, otherwise to the right child. When the process eventually arrives at a leaf, that is the desired index into the array  $a$ .

We can picture the general process by using a convenient abbreviation:  $a_m^n$  means  $\sum_{m \leq i \leq n} a_i$ . Then array  $p$  looks like this:

$a_0^0$	$a_0^1$	$a_0^2$	$a_0^3$	$a_0^4$	$a_0^5$	$a_0^6$	$a_0^7$	$a_0^8$	$a_0^9$	$a_0^{10}$	$a_0^{11}$	$a_0^{12}$	$a_0^{13}$	$a_0^{14}$	$a_0^{15}$
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	------------	------------	------------	------------	------------	------------

and the corresponding binary tree looks like this:



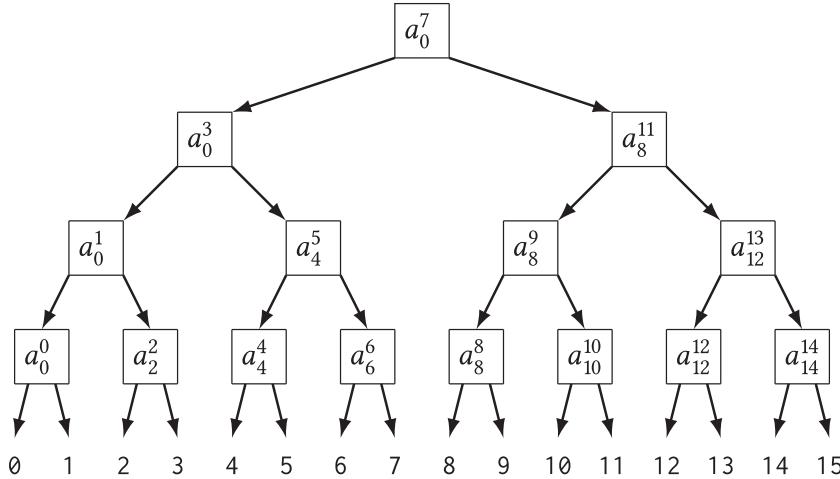
One of the central ideas in this article is that instead of computing the prefix-sum  $p$  of  $a$ , we can compute a different set of partial sums, which we will call  $p'$ :

```

1: for  $k$  from 0 through  $K - 1$  do  $p'[k] := a[k]$ 
2: for  $b$  from 1 through  $\log_2 K$  do
3:   for  $i$  from 1 through  $2^{(\log_2 K)-b}$  do
4:      $p'[(2i)2^{b-1} - 1] += p'[(2i - 1)2^{b-1} - 1]$ 
5:   end for
6: end for
  
```

$$p' \quad \boxed{a_0^0 \mid a_0^1 \mid a_2^2 \mid a_0^3 \mid a_4^4 \mid a_4^5 \mid a_6^6 \mid a_0^7 \mid a_8^8 \mid a_8^9 \mid a_{10}^{10} \mid a_8^{11} \mid a_{12}^{12} \mid a_{12}^{13} \mid a_{14}^{14} \mid a_0^{15}}$$

and we can likewise treat this array  $p'$  as a binary tree to be searched:



Notice that the partial sums in  $p'$  are generally “smaller” than those in  $p$ —that is, they are sums over fewer elements of  $a$ . In fact, half of them are equal to single elements of  $a$ ; for example,  $a_2^2 = \sum_{2 \leq i \leq 2} a_i = a_2$ .

Here, then, is the trick: while the internal nodes of this tree do not contain the complete partial sums of  $p$  needed for comparison, the values of  $p$  that we actually need can be computed from  $p'$  on the fly as we walk down the tree.

```

1: let  $u$  = value chosen uniformly at random (or pseudorandomly) from [0.0, 1.0)
2: let  $u' = sum \times u$ 
3: let  $j = 0$ 
4: let  $k = K - 1$ 
5: let  $lowValue = 0$ 
6: while  $j < k$  do
7:   let  $mid = \lfloor \frac{j+k}{2} \rfloor$ 
8:   let  $compareValue = lowValue + p'[mid]$ 
9:   if  $u' < compareValue$  then
10:     $k := mid$ 
11:   else
12:     $j := mid + 1$ 
13:     $lowValue := compareValue$ 
14:   end if
15: end while

```

This tree  $p'$  is familiar: it is an intermediate state in one parallel algorithm for computing the prefix-sum array  $p$ . Because the iterations of the inner **for** loop used to construct  $p'$  are independent, they may be executed in parallel, and so  $p'$  can be constructed from  $a$  in  $\log_2 K$  steps, building the tree from bottom to top; and  $p$  can likewise be constructed from  $p'$  in  $\log_2 K$  steps, passing information down the tree.

### 3 PARALLEL SIMD IMPLEMENTATION ON A GPU

In the CUDA programming model, as used on GPU products from NVIDIA, one may pretend that one has thousands or millions of threads, each with its own local memory, stack, and registers, and one may assign to each such thread an instance of a computation. The operating system multiplexes the hardware resources to give the illusion of more or less simultaneous execution. For management purposes, threads are grouped into *warps*, each having exactly  $W$  threads. In the actual hardware used for our experiments,  $W = 32$ ; however, for purposes of illustration, we will use  $W = 16$ , to make the figures a manageable size while keeping fonts at a readable size. Warps are automatically scheduled onto multiple SIMD processing engines, where each engine has  $W$  *lanes*, and each lane performs the computation for one thread. When an engine has completed computation for one warp, it goes on to process another warp, and so on, until all warps have been processed.

The programmer can count on absolutely simultaneous execution of the  $W$  threads that constitute any single warp. There are a few low-level operations that can exchange data synchronously among the lanes of a warp [20, 32]; two are of interest here. The expression `_shfl(value, lane)`, when executed (necessarily simultaneously) as a SIMD operation by the threads in a warp, causes each lane to make its computed *value* available to all lanes, and then returns the *value* provided by lane *lane*; in short, each lane gets to decide which of the other lanes to read from. The expression `_shfl_xor(value, m)` is, in effect, an abbreviation for `_shfl(value, myLaneId ⊕ m)`, where *myLaneId* has the value  $i$  in lane  $i$  ( $0 \leq i < W$ ), and where  $\oplus$  is the bitwise exclusive-OR operation on unsigned integers. If the same value of *m* is provided on all lanes, then `_shfl_xor` performs a permutation specified by *m*; for example, if *m* = 2, then lanes whose numbers differ in exactly one bit position (the second-least-significant bit) will exchange data.

Certain instructions can cause individual lanes of a warp to become conditionally inactive; others can force some or all lanes to become active again. These are typically used by a compiler (such as the CUDA compiler) to implement **if-then-else** statements and loops.

When the lanes of a warp execute a “load” instruction, then  $W$  words are read from memory; more precisely, at most  $W$  words are read from memory, because the memory controller performs memory reads only for active lanes (but this nicety will not matter for our purposes). The memory controller makes an effort to *coalesce* these read requests, so that if multiple words to be read happen to reside in the same cache line, then the cache line is read only once. If the lanes happen to fetch from consecutive memory locations, then maximal coalescing occurs. For the NVIDIA Titan Black processors we used,  $W = 32$  and each cache line is 128 bytes. If lane  $k$  fetches a 32-bit word at address  $x + 4k$ , then the accessed words occupy exactly 2 cache lines if  $x$  is a multiple of 128, and otherwise straddle 3 cache lines; if lane  $k$  fetches a 64-bit word at address  $x + 8k$ , then the accessed words occupy exactly 4 cache lines if  $x$  is a multiple of 128, and otherwise straddle 5 cache lines. Any of these situations is a great improvement over having to access 32 distinct cache lines.

In our LDA application, we have many documents to be processed, each with a different number of words (we pre-sort the documents so that those of any one warp likely have similar lengths). We assign one document to each thread. The thread processes each word in the document; for that word it computes a discrete distribution to be sampled, and then samples it once. All threads in a warp process the  $k$ th word of their respective documents simultaneously.

Within each thread, the array  $a$  representing the distribution to be sampled is computed on the fly, used once, and then discarded, so  $a$  resides in registers. It is the elementwise product of two other arrays,  $\theta$  and  $\phi$ ;  $\theta$  represents a discrete distribution associated with the document, so it is kept in the local memory of the thread, but  $\phi$  represents a discrete distribution associated with the word type, and comes from a very large table that must reside in main memory. There is no reason to expect significant correlation of word types among the  $k$ th words of the  $W$  documents in a warp, so very likely their  $\phi$  arrays are scattered throughout main memory.

To keep the discussion simple, for now we assume  $K = W$ . (We lift this restriction in Section 5.)

If we use a straightforward approach, then every lane loads only the data it needs and computes its own  $a$  values. This situation is pictured in Figure 1(a), where for each element the variable name  $a$  is replaced with one of the letters A through P to indicate which lane that particular element logically belongs to. In this case, every  $a$  element resides in some register of the lane to which it logically belongs. The downside is poor performance, because few memory accesses will be coalesced.

An alternative approach is to use transposed memory access. The lanes work together so that memory accesses are coalesced, then compute the  $a$  values for the data they happen to have (the precomputed  $\theta$  arrays can also be pre-transposed, so that every lane will have the elements of  $\theta$  that it needs in its own local memory). This situation is pictured in Figure 1(b). The downside is that no lane has direct access to all the  $a$  values that it needs to compute its  $p$  values.

One way out is to arrange to have lanes use `_shfl_xor` operations to exchange data, to turn the situation of Figure 1(b) into that of Figure 1(a). This is a well-known problem with well-known solutions; one that works as well as could be expected is illustrated in Figure 2. On step  $i$  ( $0 \leq i < \log_2 W$ ),  $W/2$  `_shfl_xor` instructions are used (one for every pair of register positions) to exchange registers whose numbers differ by  $2^i$  between lanes whose numbers differ by  $2^i$  (see Figure 2). First,  $W/2$  `_shfl_xor` instructions are used (one for every pair of register positions) to exchange registers whose numbers differ by 1 between lanes whose numbers differ by 1 (Figure 2(a)). Second,  $W/2$  `_shfl_xor` instructions are used to exchange registers whose numbers differ by 2 between lanes whose numbers differ by 2 (Figure 2(b)). Third,  $W/2$  `_shfl_xor` instructions are used to exchange registers whose numbers differ by 4 between lanes whose numbers differ by 4 (Figure 2(c)). Finally,  $W/2$  `_shfl_xor` instructions are used to exchange registers whose numbers differ by 8 between lanes whose numbers differ by 8 (Figure 2(d)). Thus the total number of `_shfl_xor` operations is  $\frac{1}{2}W \log_2 W$ .

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	A <sub>0</sub>	B <sub>0</sub>	C <sub>0</sub>	D <sub>0</sub>	E <sub>0</sub>	F <sub>0</sub>	G <sub>0</sub>	H <sub>0</sub>	I <sub>0</sub>	J <sub>0</sub>	K <sub>0</sub>	L <sub>0</sub>	M <sub>0</sub>	N <sub>0</sub>	O <sub>0</sub>	P <sub>0</sub>
1	A <sub>1</sub>	B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub>	E <sub>1</sub>	F <sub>1</sub>	G <sub>1</sub>	H <sub>1</sub>	I <sub>1</sub>	J <sub>1</sub>	K <sub>1</sub>	L <sub>1</sub>	M <sub>1</sub>	N <sub>1</sub>	O <sub>1</sub>	P <sub>1</sub>
2	A <sub>2</sub>	B <sub>2</sub>	C <sub>2</sub>	D <sub>2</sub>	E <sub>2</sub>	F <sub>2</sub>	G <sub>2</sub>	H <sub>2</sub>	I <sub>2</sub>	J <sub>2</sub>	K <sub>2</sub>	L <sub>2</sub>	M <sub>2</sub>	N <sub>2</sub>	O <sub>2</sub>	P <sub>2</sub>
3	A <sub>3</sub>	B <sub>3</sub>	C <sub>3</sub>	D <sub>3</sub>	E <sub>3</sub>	F <sub>3</sub>	G <sub>3</sub>	H <sub>3</sub>	I <sub>3</sub>	J <sub>3</sub>	K <sub>3</sub>	L <sub>3</sub>	M <sub>3</sub>	N <sub>3</sub>	O <sub>3</sub>	P <sub>3</sub>
4	A <sub>4</sub>	B <sub>4</sub>	C <sub>4</sub>	D <sub>4</sub>	E <sub>4</sub>	F <sub>4</sub>	G <sub>4</sub>	H <sub>4</sub>	I <sub>4</sub>	J <sub>4</sub>	K <sub>4</sub>	L <sub>4</sub>	M <sub>4</sub>	N <sub>4</sub>	O <sub>4</sub>	P <sub>4</sub>
5	A <sub>5</sub>	B <sub>5</sub>	C <sub>5</sub>	D <sub>5</sub>	E <sub>5</sub>	F <sub>5</sub>	G <sub>5</sub>	H <sub>5</sub>	I <sub>5</sub>	J <sub>5</sub>	K <sub>5</sub>	L <sub>5</sub>	M <sub>5</sub>	N <sub>5</sub>	O <sub>5</sub>	P <sub>5</sub>
6	A <sub>6</sub>	B <sub>6</sub>	C <sub>6</sub>	D <sub>6</sub>	E <sub>6</sub>	F <sub>6</sub>	G <sub>6</sub>	H <sub>6</sub>	I <sub>6</sub>	J <sub>6</sub>	K <sub>6</sub>	L <sub>6</sub>	M <sub>6</sub>	N <sub>6</sub>	O <sub>6</sub>	P <sub>6</sub>
7	A <sub>7</sub>	B <sub>7</sub>	C <sub>7</sub>	D <sub>7</sub>	E <sub>7</sub>	F <sub>7</sub>	G <sub>7</sub>	H <sub>7</sub>	I <sub>7</sub>	J <sub>7</sub>	K <sub>7</sub>	L <sub>7</sub>	M <sub>7</sub>	N <sub>7</sub>	O <sub>7</sub>	P <sub>7</sub>
8	A <sub>8</sub>	B <sub>8</sub>	C <sub>8</sub>	D <sub>8</sub>	E <sub>8</sub>	F <sub>8</sub>	G <sub>8</sub>	H <sub>8</sub>	I <sub>8</sub>	J <sub>8</sub>	K <sub>8</sub>	L <sub>8</sub>	M <sub>8</sub>	N <sub>8</sub>	O <sub>8</sub>	P <sub>8</sub>
9	A <sub>9</sub>	B <sub>9</sub>	C <sub>9</sub>	D <sub>9</sub>	E <sub>9</sub>	F <sub>9</sub>	G <sub>9</sub>	H <sub>9</sub>	I <sub>9</sub>	J <sub>9</sub>	K <sub>9</sub>	L <sub>9</sub>	M <sub>9</sub>	N <sub>9</sub>	O <sub>9</sub>	P <sub>9</sub>
10	A <sub>10</sub>	B <sub>10</sub>	C <sub>10</sub>	D <sub>10</sub>	E <sub>10</sub>	F <sub>10</sub>	G <sub>10</sub>	H <sub>10</sub>	I <sub>10</sub>	J <sub>10</sub>	K <sub>10</sub>	L <sub>10</sub>	M <sub>10</sub>	N <sub>10</sub>	O <sub>10</sub>	P <sub>10</sub>
11	A <sub>11</sub>	B <sub>11</sub>	C <sub>11</sub>	D <sub>11</sub>	E <sub>11</sub>	F <sub>11</sub>	G <sub>11</sub>	H <sub>11</sub>	I <sub>11</sub>	J <sub>11</sub>	K <sub>11</sub>	L <sub>11</sub>	M <sub>11</sub>	N <sub>11</sub>	O <sub>11</sub>	P <sub>11</sub>
12	A <sub>12</sub>	B <sub>12</sub>	C <sub>12</sub>	D <sub>12</sub>	E <sub>12</sub>	F <sub>12</sub>	G <sub>12</sub>	H <sub>12</sub>	I <sub>12</sub>	J <sub>12</sub>	K <sub>12</sub>	L <sub>12</sub>	M <sub>12</sub>	N <sub>12</sub>	O <sub>12</sub>	P <sub>12</sub>
13	A <sub>13</sub>	B <sub>13</sub>	C <sub>13</sub>	D <sub>13</sub>	E <sub>13</sub>	F <sub>13</sub>	G <sub>13</sub>	H <sub>13</sub>	I <sub>13</sub>	J <sub>13</sub>	K <sub>13</sub>	L <sub>13</sub>	M <sub>13</sub>	N <sub>13</sub>	O <sub>13</sub>	P <sub>13</sub>
14	A <sub>14</sub>	B <sub>14</sub>	C <sub>14</sub>	D <sub>14</sub>	E <sub>14</sub>	F <sub>14</sub>	G <sub>14</sub>	H <sub>14</sub>	I <sub>14</sub>	J <sub>14</sub>	K <sub>14</sub>	L <sub>14</sub>	M <sub>14</sub>	N <sub>14</sub>	O <sub>14</sub>	P <sub>14</sub>
15	A <sub>15</sub>	B <sub>15</sub>	C <sub>15</sub>	D <sub>15</sub>	E <sub>15</sub>	F <sub>15</sub>	G <sub>15</sub>	H <sub>15</sub>	I <sub>15</sub>	J <sub>15</sub>	K <sub>15</sub>	L <sub>15</sub>	M <sub>15</sub>	N <sub>15</sub>	O <sub>15</sub>	P <sub>15</sub>

(a) Normal per-lane arrays

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	A <sub>6</sub>	A <sub>7</sub>	A <sub>8</sub>	A <sub>9</sub>	A <sub>10</sub>	A <sub>11</sub>	A <sub>12</sub>	A <sub>13</sub>	A <sub>14</sub>	A <sub>15</sub>
1	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	B <sub>3</sub>	B <sub>4</sub>	B <sub>5</sub>	B <sub>6</sub>	B <sub>7</sub>	B <sub>8</sub>	B <sub>9</sub>	B <sub>10</sub>	B <sub>11</sub>	B <sub>12</sub>	B <sub>13</sub>	B <sub>14</sub>	B <sub>15</sub>
2	C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	C <sub>5</sub>	C <sub>6</sub>	C <sub>7</sub>	C <sub>8</sub>	C <sub>9</sub>	C <sub>10</sub>	C <sub>11</sub>	C <sub>12</sub>	C <sub>13</sub>	C <sub>14</sub>	C <sub>15</sub>
3	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	D <sub>5</sub>	D <sub>6</sub>	D <sub>7</sub>	D <sub>8</sub>	D <sub>9</sub>	D <sub>10</sub>	D <sub>11</sub>	D <sub>12</sub>	D <sub>13</sub>	D <sub>14</sub>	D <sub>15</sub>
4	E <sub>0</sub>	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	E <sub>7</sub>	E <sub>8</sub>	E <sub>9</sub>	E <sub>10</sub>	E <sub>11</sub>	E <sub>12</sub>	E <sub>13</sub>	E <sub>14</sub>	E <sub>15</sub>
5	F <sub>0</sub>	F <sub>1</sub>	F <sub>2</sub>	F <sub>3</sub>	F <sub>4</sub>	F <sub>5</sub>	F <sub>6</sub>	F <sub>7</sub>	F <sub>8</sub>	F <sub>9</sub>	F <sub>10</sub>	F <sub>11</sub>	F <sub>12</sub>	F <sub>13</sub>	F <sub>14</sub>	F <sub>15</sub>
6	G <sub>0</sub>	G <sub>1</sub>	G <sub>2</sub>	G <sub>3</sub>	G <sub>4</sub>	G <sub>5</sub>	G <sub>6</sub>	G <sub>7</sub>	G <sub>8</sub>	G <sub>9</sub>	G <sub>10</sub>	G <sub>11</sub>	G <sub>12</sub>	G <sub>13</sub>	G <sub>14</sub>	G <sub>15</sub>
7	H <sub>0</sub>	H <sub>1</sub>	H <sub>2</sub>	H <sub>3</sub>	H <sub>4</sub>	H <sub>5</sub>	H <sub>6</sub>	H <sub>7</sub>	H <sub>8</sub>	H <sub>9</sub>	H <sub>10</sub>	H <sub>11</sub>	H <sub>12</sub>	H <sub>13</sub>	H <sub>14</sub>	H <sub>15</sub>
8	I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>	I <sub>10</sub>	I <sub>11</sub>	I <sub>12</sub>	I <sub>13</sub>	I <sub>14</sub>	I <sub>15</sub>
9	J <sub>0</sub>	J <sub>1</sub>	J <sub>2</sub>	J <sub>3</sub>	J <sub>4</sub>	J <sub>5</sub>	J <sub>6</sub>	J <sub>7</sub>	J <sub>8</sub>	J <sub>9</sub>	J <sub>10</sub>	J <sub>11</sub>	J <sub>12</sub>	J <sub>13</sub>	J <sub>14</sub>	J <sub>15</sub>
10	K <sub>0</sub>	K <sub>1</sub>	K <sub>2</sub>	K <sub>3</sub>	K <sub>4</sub>	K <sub>5</sub>	K <sub>6</sub>	K <sub>7</sub>	K <sub>8</sub>	K <sub>9</sub>	K <sub>10</sub>	K <sub>11</sub>	K <sub>12</sub>	K <sub>13</sub>	K <sub>14</sub>	K <sub>15</sub>
11	L <sub>0</sub>	L <sub>1</sub>	L <sub>2</sub>	L <sub>3</sub>	L <sub>4</sub>	L <sub>5</sub>	L <sub>6</sub>	L <sub>7</sub>	L <sub>8</sub>	L <sub>9</sub>	L <sub>10</sub>	L <sub>11</sub>	L <sub>12</sub>	L <sub>13</sub>	L <sub>14</sub>	L <sub>15</sub>
12	M <sub>0</sub>	M <sub>1</sub>	M <sub>2</sub>	M <sub>3</sub>	M <sub>4</sub>	M <sub>5</sub>	M <sub>6</sub>	M <sub>7</sub>	M <sub>8</sub>	M <sub>9</sub>	M <sub>10</sub>	M <sub>11</sub>	M <sub>12</sub>	M <sub>13</sub>	M <sub>14</sub>	M <sub>15</sub>
13	N <sub>0</sub>	N <sub>1</sub>	N <sub>2</sub>	N <sub>3</sub>	N <sub>4</sub>	N <sub>5</sub>	N <sub>6</sub>	N <sub>7</sub>	N <sub>8</sub>	N <sub>9</sub>	N <sub>10</sub>	N <sub>11</sub>	N <sub>12</sub>	N <sub>13</sub>	N <sub>14</sub>	N <sub>15</sub>
14	O <sub>0</sub>	O <sub>1</sub>	O <sub>2</sub>	O <sub>3</sub>	O <sub>4</sub>	O <sub>5</sub>	O <sub>6</sub>	O <sub>7</sub>	O <sub>8</sub>	O <sub>9</sub>	O <sub>10</sub>	O <sub>11</sub>	O <sub>12</sub>	O <sub>13</sub>	O <sub>14</sub>	O <sub>15</sub>
15	P <sub>0</sub>	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	P <sub>5</sub>	P <sub>6</sub>	P <sub>7</sub>	P <sub>8</sub>	P <sub>9</sub>	P <sub>10</sub>	P <sub>11</sub>	P <sub>12</sub>	P <sub>13</sub>	P <sub>14</sub>	P <sub>15</sub>

(b) Transposed arrays

Fig. 1. Array storage in lane registers (illustrated for  $W = 16$ ). Each column represents  $W$  consecutive registers in the register file of one lane. Array elements are numbered, and lanes are labeled by letters, so “C<sub>4</sub>” denotes element 4 of the array that logically belongs to (is intended to be processed by) lane C. In the normal layout, all array elements to be processed by lane C reside in the registers of lane C, but in the transposed layout, only element C<sub>2</sub> resides in a register of lane C—all other elements reside in other lanes (in register 2 of each lane).

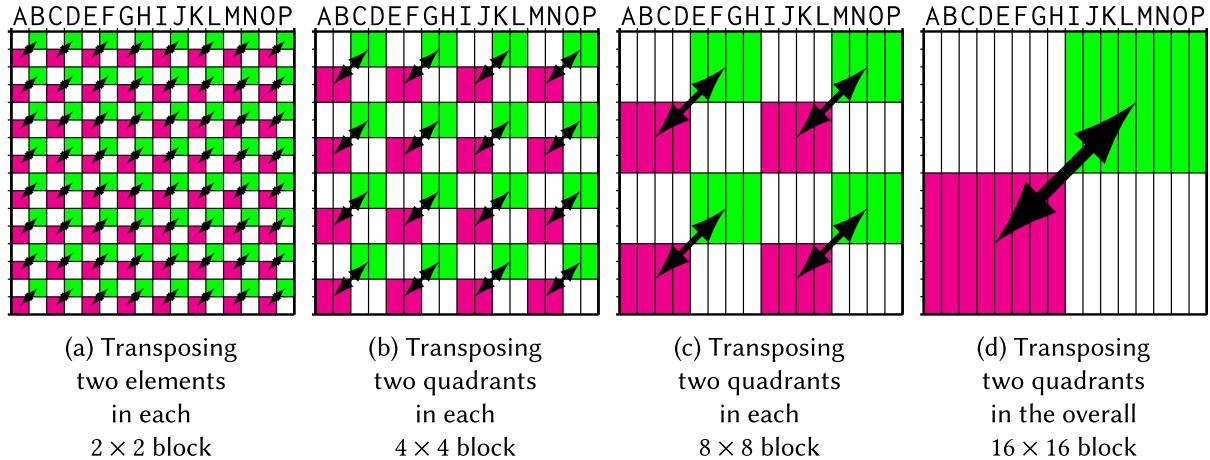


Fig. 2. Transposing a matrix consisting of  $W$  registers in each of the  $W$  lanes of the GPU (shown for  $W = 16$ ).

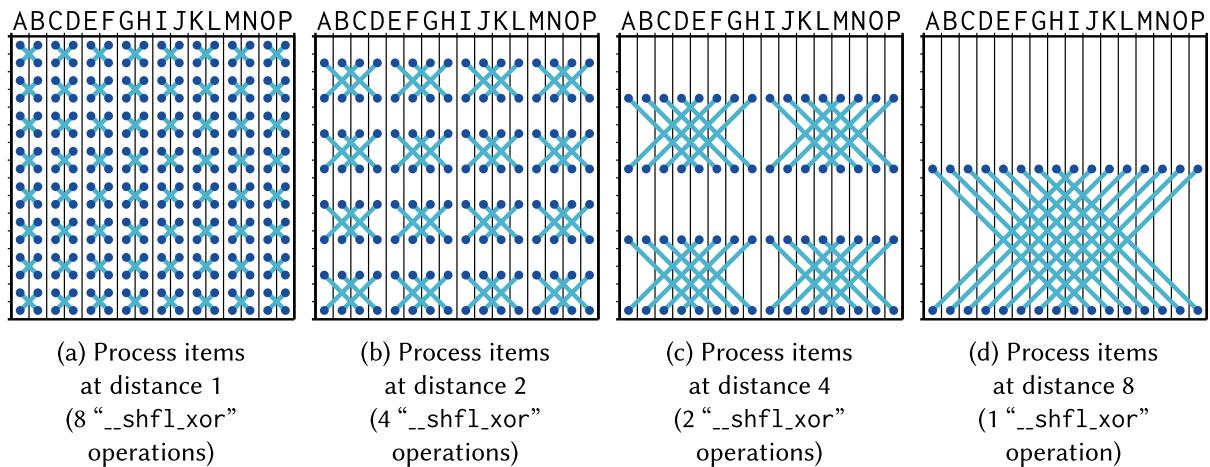


Fig. 3. Patterns of application of butterfly computations to compute butterfly-patterned  $p'$  arrays ( $W = 16$ ).

We offer a novel approach that uses  $3(W - 1)$  shuffle operations<sup>1</sup> and avoids all scattered memory access. The idea is to have the lanes cooperate to construct a partial sums table that is related to the  $p'$  arrays discussed in Section 2. Instead of ending up with every lane having all its own  $p'$  array elements, each array is distributed across multiple lanes—but instead of every lane containing exactly one element of every  $p'$  array as in the transposed layout shown in Figure 1(b), they are distributed in a more complicated way: We call it a “butterfly-patterned partial sums table.” The construction of this table requires only  $W - 1$  `__shfl_xor` operations. During the binary search, an additional  $2(W - 1)$  `__shfl` and `__shfl_xor` operations are used as the lanes assist each other in accessing array elements.

Here is how the butterfly-patterned table is constructed. There are  $\log_2 W$  steps, and during step  $i$  ( $0 \leq i < \log_2 W$ ),  $2^{(\log_2 W)-i-1}$  `__shfl_xor` instructions are used to perform  $(2^{(\log_2 W)-i-1}) \frac{W}{2}$  butterfly computations (see Figure 3). Each butterfly computation operates on four entries, within

<sup>1</sup>We realize that  $3(W - 1) > \frac{1}{2}W \log_2 W$  for  $W = 16$  or even for  $W = 32$ . We remark on the algorithmic complexity as a function of the number of shuffle operations as a matter of mathematical principle, but we recognize that this comparison does not explain the faster speed observed in practice with the butterfly-patterned partial sums for  $W = 32$ . Rather, the observed speedup comes from an overall reduction of instructions and the manner in which the CUDA compiler manages to schedule them. In the future, if GPU chips are ever built with  $W \geq 64$ , then the improvement in algorithmic complexity may also begin to matter.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A <sub>0</sub> <sup>0</sup>	B <sub>0</sub> <sup>0</sup>	A <sub>2</sub> <sup>2</sup>	B <sub>2</sub> <sup>2</sup>	A <sub>4</sub> <sup>4</sup>	B <sub>4</sub> <sup>4</sup>	A <sub>6</sub> <sup>6</sup>	B <sub>6</sub> <sup>6</sup>	A <sub>8</sub> <sup>8</sup>	B <sub>8</sub> <sup>8</sup>	A <sub>10</sub> <sup>10</sup>	B <sub>10</sub> <sup>10</sup>	A <sub>12</sub> <sup>12</sup>	B <sub>12</sub> <sup>12</sup>	A <sub>14</sub> <sup>14</sup>	B <sub>14</sub> <sup>14</sup>
3	A <sub>0</sub> <sup>1</sup>	B <sub>0</sub> <sup>1</sup>	C <sub>0</sub> <sup>1</sup>	D <sub>0</sub> <sup>1</sup>	A <sub>4</sub> <sup>5</sup>	B <sub>4</sub> <sup>5</sup>	C <sub>4</sub> <sup>5</sup>	D <sub>4</sub> <sup>5</sup>	A <sub>8</sub> <sup>9</sup>	B <sub>8</sub> <sup>9</sup>	C <sub>8</sub> <sup>9</sup>	D <sub>8</sub> <sup>9</sup>	A <sub>12</sub> <sup>13</sup>	B <sub>12</sub> <sup>13</sup>	C <sub>12</sub> <sup>13</sup>	D <sub>12</sub> <sup>13</sup>
4	C <sub>0</sub> <sup>0</sup>	D <sub>0</sub> <sup>0</sup>	C <sub>2</sub> <sup>2</sup>	D <sub>2</sub> <sup>2</sup>	C <sub>4</sub> <sup>4</sup>	D <sub>4</sub> <sup>4</sup>	C <sub>6</sub> <sup>6</sup>	D <sub>6</sub> <sup>6</sup>	C <sub>8</sub> <sup>8</sup>	D <sub>8</sub> <sup>8</sup>	C <sub>10</sub> <sup>10</sup>	D <sub>10</sub> <sup>10</sup>	C <sub>12</sub> <sup>12</sup>	D <sub>12</sub> <sup>12</sup>	C <sub>14</sub> <sup>14</sup>	D <sub>14</sub> <sup>14</sup>
2	A <sub>0</sub> <sup>3</sup>	B <sub>0</sub> <sup>3</sup>	C <sub>0</sub> <sup>3</sup>	D <sub>0</sub> <sup>3</sup>	E <sub>0</sub> <sup>3</sup>	F <sub>0</sub> <sup>3</sup>	G <sub>0</sub> <sup>3</sup>	H <sub>0</sub> <sup>3</sup>	A <sub>8</sub> <sup>11</sup>	B <sub>8</sub> <sup>11</sup>	C <sub>8</sub> <sup>11</sup>	D <sub>8</sub> <sup>11</sup>	E <sub>8</sub> <sup>11</sup>	F <sub>8</sub> <sup>11</sup>	G <sub>8</sub> <sup>11</sup>	H <sub>8</sub> <sup>11</sup>
4	E <sub>0</sub> <sup>0</sup>	F <sub>0</sub> <sup>0</sup>	E <sub>2</sub> <sup>2</sup>	F <sub>2</sub> <sup>2</sup>	E <sub>4</sub> <sup>4</sup>	F <sub>4</sub> <sup>4</sup>	E <sub>6</sub> <sup>6</sup>	F <sub>6</sub> <sup>6</sup>	E <sub>8</sub> <sup>8</sup>	F <sub>8</sub> <sup>8</sup>	E <sub>10</sub> <sup>10</sup>	F <sub>10</sub> <sup>10</sup>	E <sub>12</sub> <sup>12</sup>	F <sub>12</sub> <sup>12</sup>	E <sub>14</sub> <sup>14</sup>	F <sub>14</sub> <sup>14</sup>
3	E <sub>0</sub> <sup>1</sup>	F <sub>0</sub> <sup>1</sup>	G <sub>0</sub> <sup>1</sup>	H <sub>0</sub> <sup>1</sup>	E <sub>4</sub> <sup>5</sup>	F <sub>4</sub> <sup>5</sup>	G <sub>4</sub> <sup>5</sup>	H <sub>4</sub> <sup>5</sup>	E <sub>8</sub> <sup>9</sup>	F <sub>8</sub> <sup>9</sup>	G <sub>8</sub> <sup>9</sup>	H <sub>8</sub> <sup>9</sup>	E <sub>12</sub> <sup>13</sup>	F <sub>12</sub> <sup>13</sup>	G <sub>12</sub> <sup>13</sup>	H <sub>12</sub> <sup>13</sup>
4	G <sub>0</sub> <sup>0</sup>	H <sub>0</sub> <sup>0</sup>	G <sub>2</sub> <sup>2</sup>	H <sub>2</sub> <sup>2</sup>	G <sub>4</sub> <sup>4</sup>	H <sub>4</sub> <sup>4</sup>	G <sub>6</sub> <sup>6</sup>	H <sub>6</sub> <sup>6</sup>	G <sub>8</sub> <sup>8</sup>	H <sub>8</sub> <sup>8</sup>	G <sub>10</sub> <sup>10</sup>	H <sub>10</sub> <sup>10</sup>	G <sub>12</sub> <sup>12</sup>	H <sub>12</sub> <sup>12</sup>	G <sub>14</sub> <sup>14</sup>	H <sub>14</sub> <sup>14</sup>
1	A <sub>0</sub> <sup>7</sup>	B <sub>0</sub> <sup>7</sup>	C <sub>0</sub> <sup>7</sup>	D <sub>0</sub> <sup>7</sup>	E <sub>0</sub> <sup>7</sup>	F <sub>0</sub> <sup>7</sup>	G <sub>0</sub> <sup>7</sup>	H <sub>0</sub> <sup>7</sup>	I <sub>0</sub> <sup>7</sup>	J <sub>0</sub> <sup>7</sup>	K <sub>0</sub> <sup>7</sup>	L <sub>0</sub> <sup>7</sup>	M <sub>0</sub> <sup>7</sup>	N <sub>0</sub> <sup>7</sup>	O <sub>0</sub> <sup>7</sup>	P <sub>0</sub> <sup>7</sup>
4	I <sub>0</sub> <sup>0</sup>	J <sub>0</sub> <sup>0</sup>	I <sub>2</sub> <sup>2</sup>	J <sub>2</sub> <sup>2</sup>	I <sub>4</sub> <sup>4</sup>	J <sub>4</sub> <sup>4</sup>	I <sub>6</sub> <sup>6</sup>	J <sub>6</sub> <sup>6</sup>	I <sub>8</sub> <sup>8</sup>	J <sub>8</sub> <sup>8</sup>	I <sub>10</sub> <sup>10</sup>	J <sub>10</sub> <sup>10</sup>	I <sub>12</sub> <sup>12</sup>	J <sub>12</sub> <sup>12</sup>	I <sub>14</sub> <sup>14</sup>	J <sub>14</sub> <sup>14</sup>
3	I <sub>0</sub> <sup>1</sup>	J <sub>0</sub> <sup>1</sup>	K <sub>0</sub> <sup>1</sup>	L <sub>0</sub> <sup>1</sup>	I <sub>4</sub> <sup>5</sup>	J <sub>4</sub> <sup>5</sup>	K <sub>4</sub> <sup>5</sup>	L <sub>4</sub> <sup>5</sup>	I <sub>8</sub> <sup>9</sup>	J <sub>8</sub> <sup>9</sup>	K <sub>8</sub> <sup>9</sup>	L <sub>8</sub> <sup>9</sup>	I <sub>12</sub> <sup>13</sup>	J <sub>12</sub> <sup>13</sup>	K <sub>12</sub> <sup>13</sup>	L <sub>12</sub> <sup>13</sup>
4	K <sub>0</sub> <sup>0</sup>	L <sub>0</sub> <sup>0</sup>	K <sub>2</sub> <sup>2</sup>	L <sub>2</sub> <sup>2</sup>	K <sub>4</sub> <sup>4</sup>	L <sub>4</sub> <sup>4</sup>	K <sub>6</sub> <sup>6</sup>	L <sub>6</sub> <sup>6</sup>	K <sub>8</sub> <sup>8</sup>	L <sub>8</sub> <sup>8</sup>	K <sub>10</sub> <sup>10</sup>	L <sub>10</sub> <sup>10</sup>	K <sub>12</sub> <sup>12</sup>	L <sub>12</sub> <sup>12</sup>	K <sub>14</sub> <sup>14</sup>	L <sub>14</sub> <sup>14</sup>
2	I <sub>0</sub> <sup>3</sup>	J <sub>0</sub> <sup>3</sup>	K <sub>0</sub> <sup>3</sup>	L <sub>0</sub> <sup>3</sup>	M <sub>0</sub> <sup>3</sup>	N <sub>0</sub> <sup>3</sup>	O <sub>0</sub> <sup>3</sup>	P <sub>0</sub> <sup>3</sup>	I <sub>8</sub> <sup>11</sup>	J <sub>8</sub> <sup>11</sup>	K <sub>8</sub> <sup>11</sup>	L <sub>8</sub> <sup>11</sup>	M <sub>8</sub> <sup>11</sup>	N <sub>8</sub> <sup>11</sup>	O <sub>8</sub> <sup>11</sup>	P <sub>8</sub> <sup>11</sup>
4	M <sub>0</sub> <sup>0</sup>	N <sub>0</sub> <sup>0</sup>	M <sub>2</sub> <sup>2</sup>	N <sub>2</sub> <sup>2</sup>	M <sub>4</sub> <sup>4</sup>	N <sub>4</sub> <sup>4</sup>	M <sub>6</sub> <sup>6</sup>	N <sub>6</sub> <sup>6</sup>	M <sub>8</sub> <sup>8</sup>	N <sub>8</sub> <sup>8</sup>	M <sub>10</sub> <sup>10</sup>	N <sub>10</sub> <sup>10</sup>	M <sub>12</sub> <sup>12</sup>	N <sub>12</sub> <sup>12</sup>	M <sub>14</sub> <sup>14</sup>	N <sub>14</sub> <sup>14</sup>
3	M <sub>0</sub> <sup>1</sup>	N <sub>0</sub> <sup>1</sup>	O <sub>0</sub> <sup>1</sup>	P <sub>0</sub> <sup>1</sup>	M <sub>4</sub> <sup>5</sup>	N <sub>4</sub> <sup>5</sup>	O <sub>4</sub> <sup>5</sup>	P <sub>4</sub> <sup>5</sup>	M <sub>8</sub> <sup>9</sup>	N <sub>8</sub> <sup>9</sup>	O <sub>8</sub> <sup>9</sup>	P <sub>8</sub> <sup>9</sup>	M <sub>12</sub> <sup>13</sup>	N <sub>12</sub> <sup>13</sup>	O <sub>12</sub> <sup>13</sup>	P <sub>12</sub> <sup>13</sup>
4	O <sub>0</sub> <sup>0</sup>	P <sub>0</sub> <sup>0</sup>	O <sub>2</sub> <sup>2</sup>	P <sub>2</sub> <sup>2</sup>	O <sub>4</sub> <sup>4</sup>	P <sub>4</sub> <sup>4</sup>	O <sub>6</sub> <sup>6</sup>	P <sub>6</sub> <sup>6</sup>	O <sub>8</sub> <sup>8</sup>	P <sub>8</sub> <sup>8</sup>	O <sub>10</sub> <sup>10</sup>	P <sub>10</sub> <sup>10</sup>	O <sub>12</sub> <sup>12</sup>	P <sub>12</sub> <sup>12</sup>	O <sub>14</sub> <sup>14</sup>	P <sub>14</sub> <sup>14</sup>
S	A <sub>0</sub> <sup>15</sup>	B <sub>0</sub> <sup>15</sup>	C <sub>0</sub> <sup>15</sup>	D <sub>0</sub> <sup>15</sup>	E <sub>0</sub> <sup>15</sup>	F <sub>0</sub> <sup>15</sup>	G <sub>0</sub> <sup>15</sup>	H <sub>0</sub> <sup>15</sup>	I <sub>0</sub> <sup>15</sup>	J <sub>0</sub> <sup>15</sup>	K <sub>0</sub> <sup>15</sup>	L <sub>0</sub> <sup>15</sup>	M <sub>0</sub> <sup>15</sup>	N <sub>0</sub> <sup>15</sup>	O <sub>0</sub> <sup>15</sup>	P <sub>0</sub> <sup>15</sup>

Fig. 4. The butterfly-patterned table that results from the butterfly computations of Figure 3 ( $W = 16$ ).

the  $W p'$  arrays, that are at the intersection of two rows whose indices differ by a power of 2 and two columns whose indices differ by that same power of 2. Suppose the four values in those entries are  $\left[ \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right]$ ; they are replaced by  $\left[ \begin{smallmatrix} a & c \\ a+b & c+d \end{smallmatrix} \right]$ . Each such replacement operation on four entries is symbolized by  $\otimes$  in the figures. Here is CUDA code for constructing the butterfly-patterned table using this replacement operation:

```

int r = threadIdx.x & 0x1f; /* lane ID */
for (int b=1; b < W; b+=b) { /* 1, 2, 4, 8, ... */
    for (int j=0; j < (W>>(b+1)); j++) {
        d = (((j << 1) + 1) << b) - 1;
        h = (r & (1<<b)) ? a[d] : a[d+(1<<b)];
        v = __shfl_xor(h, 1<<b);
        if (r & (1<<b)) a[d] = v;
        else a[d+(1<<b)] = v;
        a[d+(1<<b)] += a[d];
        p[d] = a[d];
    }
    p[W-1] = a[W-1];
}

```

These 3 lines are  
replaced below to  
make a new version.

The result is shown in Figure 4. The rows of this figure are labeled with tree levels (1 through 4) and S. Observe that in the row labeled S, every lane has the sum of its own a array. Observe also in the row labeled 1, every lane has the root of its own binary search tree for  $p'$ . The two rows labeled 2 collectively contain all the level-2 internal nodes of the trees, the four rows labeled 3 contain all the level-3 nodes, and the eight rows labeled 4 contain all the level-4 nodes.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	$A_0^0$	$B_0^0$	$A_2^2$	$B_2^2$	$A_4^4$	$B_4^4$	$A_6^6$	$B_6^6$	$A_8^8$	$B_8^8$	$A_{10}^{10}$	$B_{10}^{10}$	$A_{12}^{12}$	$B_{12}^{12}$	$A_{14}^{14}$	$B_{14}^{14}$
3	$A_0^1$	$B_0^1$	$C_0^1$	$D_0^1$	$A_4^5$	$B_4^5$	$C_4^5$	$D_4^5$	$A_8^9$	$B_8^9$	$C_8^9$	$D_8^9$	$A_{10}^{11}$	$B_{10}^{11}$	$A_{12}^{13}$	$B_{12}^{13}$
2	$A_0^3$	$B_0^3$	$C_0^3$	$D_0^3$	$E_0^3$	$F_0^3$	$G_0^3$	$H_0^3$	$A_8^{11}$	$B_8^{11}$	$C_8^{11}$	$D_8^{11}$	$E_8^{11}$	$F_8^{11}$	$G_8^{11}$	$H_8^{11}$
1	$A_0^7$	$B_0^7$	$C_0^7$	$D_0^7$	$E_0^7$	$F_0^7$	$G_0^7$	$H_0^7$	$I_0^7$	$J_0^7$	$K_0^7$	$L_0^7$	$M_0^7$	$N_0^7$	$O_0^7$	$P_0^7$
4	$I_0^0$	$J_0^0$	$I_2^2$	$J_2^2$	$I_4^4$	$J_4^4$	$I_6^6$	$J_6^6$	$I_8^8$	$J_8^8$	$I_{10}^{10}$	$J_{10}^{10}$	$I_{12}^{12}$	$J_{12}^{12}$	$I_{14}^{14}$	$J_{14}^{14}$
3	$I_0^1$	$J_0^1$	$K_0^1$	$L_0^1$	$I_4^5$	$J_4^5$	$K_4^5$	$L_4^5$	$I_8^9$	$J_8^9$	$K_8^9$	$L_8^9$	$I_{12}^{13}$	$J_{12}^{13}$	$K_{12}^{13}$	$L_{12}^{13}$
4	$K_0^0$	$L_0^0$	$K_2^2$	$L_2^2$	$K_4^4$	$L_4^4$	$K_6^6$	$L_6^6$	$K_8^8$	$L_8^8$	$K_{10}^{10}$	$L_{10}^{10}$	$K_{12}^{12}$	$L_{12}^{12}$	$K_{14}^{14}$	$L_{14}^{14}$
2	$I_0^3$	$J_0^3$	$K_0^3$	$L_0^3$	$M_0^3$	$N_0^3$	$O_0^3$	$P_0^3$	$I_8^{11}$	$J_8^{11}$	$K_8^{11}$	$L_8^{11}$	$M_8^{11}$	$N_8^{11}$	$O_8^{11}$	$P_8^{11}$
4	$M_0^0$	$N_0^0$	$M_2^2$	$N_2^2$	$M_4^4$	$N_4^4$	$M_6^6$	$N_6^6$	$M_8^8$	$N_8^8$	$M_{10}^{10}$	$N_{10}^{10}$	$M_{12}^{12}$	$N_{12}^{12}$	$M_{14}^{14}$	$N_{14}^{14}$
3	$M_0^1$	$N_0^1$	$O_0^1$	$P_0^1$	$M_4^5$	$N_4^5$	$O_4^5$	$P_4^5$	$M_8^9$	$N_8^9$	$O_8^9$	$P_8^9$	$M_{12}^{13}$	$N_{12}^{13}$	$O_{12}^{13}$	$P_{12}^{13}$
4	$O_0^0$	$P_0^0$	$O_2^2$	$P_2^2$	$O_4^4$	$P_4^4$	$O_6^6$	$P_6^6$	$O_8^8$	$P_8^8$	$O_{10}^{10}$	$P_{10}^{10}$	$O_{12}^{12}$	$P_{12}^{12}$	$O_{14}^{14}$	$P_{14}^{14}$
S	$A_0^{15}$	$B_0^{15}$	$C_0^{15}$	$D_0^{15}$	$E_0^{15}$	$F_0^{15}$	$G_0^{15}$	$H_0^{15}$	$I_0^{15}$	$J_0^{15}$	$K_0^{15}$	$L_0^{15}$	$M_0^{15}$	$N_0^{15}$	$O_0^{15}$	$P_0^{15}$

Fig. 5. How the tree and total sum for lane A are embedded within the butterfly-patterned table.

Figure 5 is a copy of Figure 4, but highlights entries that logically belong to lane A; it is easy to see the sum (in the bottom row) as well as the entire binary search tree, indicated by the arrows. Similarly, Figure 6 is a copy of Figure 4 but highlights entries that logically belong to lane O. In the same manner, Figure 7 highlights entries that logically belong to lane F; the arrangement of the tree nodes is a bit more contorted, but the arrows show how they are organized. And indeed the first 15 rows of the table collectively contain a tree for each of the 16 lanes A through P, with the root of each tree appearing in the eighth line, and the last row of the table (labeled “S”) contains the overall sum for each of the 16 lanes.

Here is a precise description of the pattern: The entry in row  $i$  and column  $j$  contains the value  $X_y^w$  where  $m = i \oplus (i + 1)$ ,  $k = \lfloor \frac{m}{2} \rfloor$ ,  $\ell = (i \& \neg m) + (j \& m)$ ,  $X = \text{"ABCDEFGHIJKLMNP"}[\ell]$ ,  $y = j \& (\neg k)$ , and  $w = y + k$ . We use “ $\neg$ ” to indicate bitwise NOT, “ $\&$ ” to indicate bitwise AND, and (as earlier) “ $\oplus$ ” to indicate bitwise XOR.

Given this table, it is just a matter of making sure that each lane has access to the tree nodes it needs during the binary search. During step  $i$  of the binary search ( $0 \leq i < \log_2 W$ ),  $2^{(\log_2 W)-i-1}$  `_shfl_xor` instructions are executed to make available all entries in rows labeled  $(\log_2 W) - i$ , and each lane picks off the entry of interest by computing which lane contains it and which `_shfl_xor` instruction will post it;  $2^{(\log_2 W)-i-1}$  `_shfl` instructions are also needed to exchange array-address information.

A variation of this technique proved to be even faster in our experiments. It uses a modified butterfly computation: instead of replacing  $\frac{a|b}{c|d}$  with  $\frac{a}{a+b} \frac{c}{c+d}$ , it replaces them with  $\frac{a}{a+b} \frac{d}{c+d}$  (the only difference is providing  $d$  instead of  $c$  at the upper right). This alternate replacement saves one machine instruction at the lowest level in the innermost loop. Merely replace the three lines of CUDA code indicated above with two lines, to produce this code:

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A <sub>0</sub> <sup>0</sup>	B <sub>0</sub> <sup>0</sup>	A <sub>2</sub> <sup>2</sup>	B <sub>2</sub> <sup>2</sup>	A <sub>4</sub> <sup>4</sup>	B <sub>4</sub> <sup>4</sup>	A <sub>6</sub> <sup>6</sup>	B <sub>6</sub> <sup>6</sup>	A <sub>8</sub> <sup>8</sup>	B <sub>8</sub> <sup>8</sup>	A <sub>10</sub> <sup>10</sup>	B <sub>10</sub> <sup>10</sup>	A <sub>12</sub> <sup>12</sup>	B <sub>12</sub> <sup>12</sup>	A <sub>14</sub> <sup>14</sup>	B <sub>14</sub> <sup>14</sup>
3	A <sub>0</sub> <sup>1</sup>	B <sub>0</sub> <sup>1</sup>	C <sub>0</sub> <sup>1</sup>	D <sub>0</sub> <sup>1</sup>	A <sub>5</sub> <sup>5</sup>	B <sub>5</sub> <sup>5</sup>	C <sub>4</sub> <sup>4</sup>	D <sub>4</sub> <sup>5</sup>	A <sub>9</sub> <sup>8</sup>	B <sub>9</sub> <sup>8</sup>	C <sub>9</sub> <sup>8</sup>	D <sub>8</sub> <sup>8</sup>	A <sub>13</sub> <sup>13</sup>	B <sub>12</sub> <sup>12</sup>	C <sub>13</sub> <sup>13</sup>	D <sub>12</sub> <sup>12</sup>
4	C <sub>0</sub> <sup>0</sup>	D <sub>0</sub> <sup>0</sup>	C <sub>2</sub> <sup>2</sup>	D <sub>2</sub> <sup>2</sup>	C <sub>4</sub> <sup>4</sup>	D <sub>4</sub> <sup>4</sup>	C <sub>6</sub> <sup>6</sup>	D <sub>6</sub> <sup>6</sup>	C <sub>8</sub> <sup>8</sup>	D <sub>8</sub> <sup>8</sup>	C <sub>10</sub> <sup>10</sup>	D <sub>10</sub> <sup>10</sup>	C <sub>12</sub> <sup>12</sup>	D <sub>12</sub> <sup>12</sup>	C <sub>14</sub> <sup>14</sup>	D <sub>14</sub> <sup>14</sup>
2	A <sub>3</sub> <sup>3</sup>	B <sub>3</sub> <sup>3</sup>	C <sub>3</sub> <sup>3</sup>	D <sub>3</sub> <sup>3</sup>	E <sub>3</sub> <sup>0</sup>	F <sub>3</sub> <sup>3</sup>	G <sub>3</sub> <sup>3</sup>	H <sub>3</sub> <sup>0</sup>	A <sub>11</sub> <sup>8</sup>	B <sub>11</sub> <sup>8</sup>	C <sub>11</sub> <sup>8</sup>	D <sub>11</sub> <sup>8</sup>	E <sub>11</sub> <sup>8</sup>	F <sub>11</sub> <sup>8</sup>	G <sub>11</sub> <sup>8</sup>	H <sub>8</sub> <sup>8</sup>
4	E <sub>0</sub> <sup>0</sup>	F <sub>0</sub> <sup>0</sup>	E <sub>2</sub> <sup>2</sup>	F <sub>2</sub> <sup>2</sup>	E <sub>4</sub> <sup>4</sup>	F <sub>4</sub> <sup>4</sup>	E <sub>6</sub> <sup>6</sup>	F <sub>6</sub> <sup>6</sup>	E <sub>8</sub> <sup>8</sup>	F <sub>8</sub> <sup>8</sup>	E <sub>10</sub> <sup>10</sup>	F <sub>10</sub> <sup>10</sup>	E <sub>12</sub> <sup>12</sup>	F <sub>12</sub> <sup>12</sup>	E <sub>14</sub> <sup>14</sup>	F <sub>14</sub> <sup>14</sup>
3	E <sub>1</sub> <sup>1</sup>	F <sub>1</sub> <sup>1</sup>	G <sub>1</sub> <sup>1</sup>	H <sub>1</sub> <sup>0</sup>	E <sub>5</sub> <sup>5</sup>	F <sub>5</sub> <sup>4</sup>	G <sub>5</sub> <sup>4</sup>	H <sub>5</sub> <sup>4</sup>	E <sub>9</sub> <sup>8</sup>	F <sub>9</sub> <sup>8</sup>	G <sub>9</sub> <sup>8</sup>	H <sub>9</sub> <sup>8</sup>	E <sub>13</sub> <sup>12</sup>	F <sub>13</sub> <sup>12</sup>	G <sub>13</sub> <sup>12</sup>	H <sub>12</sub> <sup>12</sup>
4	G <sub>0</sub> <sup>0</sup>	H <sub>0</sub> <sup>0</sup>	G <sub>2</sub> <sup>2</sup>	H <sub>2</sub> <sup>2</sup>	G <sub>4</sub> <sup>4</sup>	H <sub>4</sub> <sup>4</sup>	G <sub>6</sub> <sup>6</sup>	H <sub>6</sub> <sup>6</sup>	G <sub>8</sub> <sup>8</sup>	H <sub>8</sub> <sup>8</sup>	G <sub>10</sub> <sup>10</sup>	H <sub>10</sub> <sup>10</sup>	G <sub>12</sub> <sup>12</sup>	H <sub>12</sub> <sup>12</sup>	G <sub>14</sub> <sup>14</sup>	H <sub>14</sub> <sup>14</sup>
1	A <sub>0</sub> <sup>7</sup>	B <sub>0</sub> <sup>7</sup>	C <sub>0</sub> <sup>7</sup>	D <sub>0</sub> <sup>7</sup>	E <sub>7</sub> <sup>0</sup>	F <sub>7</sub> <sup>0</sup>	G <sub>7</sub> <sup>0</sup>	H <sub>7</sub> <sup>0</sup>	I <sub>7</sub> <sup>0</sup>	J <sub>7</sub> <sup>0</sup>	K <sub>7</sub> <sup>0</sup>	L <sub>7</sub> <sup>0</sup>	M <sub>7</sub> <sup>0</sup>	N <sub>7</sub> <sup>0</sup>	O <sub>7</sub> <sup>0</sup>	P <sub>7</sub> <sup>0</sup>
4	I <sub>0</sub> <sup>0</sup>	J <sub>0</sub> <sup>0</sup>	I <sub>2</sub> <sup>2</sup>	J <sub>2</sub> <sup>2</sup>	I <sub>4</sub> <sup>4</sup>	J <sub>4</sub> <sup>4</sup>	I <sub>6</sub> <sup>6</sup>	J <sub>6</sub> <sup>6</sup>	I <sub>8</sub> <sup>8</sup>	J <sub>8</sub> <sup>8</sup>	I <sub>10</sub> <sup>10</sup>	J <sub>10</sub> <sup>10</sup>	I <sub>12</sub> <sup>12</sup>	J <sub>12</sub> <sup>12</sup>	I <sub>14</sub> <sup>14</sup>	J <sub>14</sub> <sup>14</sup>
3	I <sub>1</sub> <sup>0</sup>	J <sub>1</sub> <sup>0</sup>	K <sub>1</sub> <sup>0</sup>	L <sub>0</sub> <sup>1</sup>	I <sub>5</sub> <sup>4</sup>	J <sub>5</sub> <sup>4</sup>	K <sub>5</sub> <sup>4</sup>	L <sub>4</sub> <sup>5</sup>	I <sub>9</sub> <sup>8</sup>	J <sub>9</sub> <sup>8</sup>	K <sub>9</sub> <sup>8</sup>	L <sub>8</sub> <sup>8</sup>	I <sub>13</sub> <sup>12</sup>	J <sub>12</sub> <sup>12</sup>	K <sub>13</sub> <sup>12</sup>	L <sub>12</sub> <sup>12</sup>
4	K <sub>0</sub> <sup>0</sup>	L <sub>0</sub> <sup>0</sup>	K <sub>2</sub> <sup>2</sup>	L <sub>2</sub> <sup>2</sup>	K <sub>4</sub> <sup>4</sup>	L <sub>4</sub> <sup>4</sup>	K <sub>6</sub> <sup>6</sup>	L <sub>6</sub> <sup>6</sup>	K <sub>8</sub> <sup>8</sup>	L <sub>8</sub> <sup>8</sup>	K <sub>10</sub> <sup>10</sup>	L <sub>10</sub> <sup>10</sup>	K <sub>12</sub> <sup>12</sup>	L <sub>12</sub> <sup>12</sup>	K <sub>14</sub> <sup>14</sup>	L <sub>14</sub> <sup>14</sup>
2	I <sub>3</sub> <sup>0</sup>	J <sub>3</sub> <sup>0</sup>	K <sub>3</sub> <sup>0</sup>	L <sub>3</sub> <sup>0</sup>	M <sub>3</sub> <sup>3</sup>	N <sub>3</sub> <sup>3</sup>	O <sub>3</sub> <sup>0</sup>	P <sub>3</sub> <sup>3</sup>	I <sub>11</sub> <sup>8</sup>	J <sub>11</sub> <sup>8</sup>	K <sub>11</sub> <sup>8</sup>	L <sub>8</sub> <sup>8</sup>	M <sub>11</sub> <sup>8</sup>	N <sub>11</sub> <sup>8</sup>	O <sub>11</sub> <sup>8</sup>	P <sub>11</sub> <sup>8</sup>
4	M <sub>0</sub> <sup>0</sup>	N <sub>0</sub> <sup>0</sup>	M <sub>2</sub> <sup>2</sup>	N <sub>2</sub> <sup>2</sup>	M <sub>4</sub> <sup>4</sup>	N <sub>4</sub> <sup>4</sup>	M <sub>6</sub> <sup>6</sup>	N <sub>6</sub> <sup>6</sup>	M <sub>8</sub> <sup>8</sup>	N <sub>8</sub> <sup>8</sup>	M <sub>10</sub> <sup>10</sup>	N <sub>10</sub> <sup>10</sup>	M <sub>12</sub> <sup>12</sup>	N <sub>12</sub> <sup>12</sup>	M <sub>14</sub> <sup>14</sup>	N <sub>14</sub> <sup>14</sup>
3	M <sub>1</sub> <sup>0</sup>	N <sub>1</sub> <sup>0</sup>	O <sub>1</sub> <sup>0</sup>	P <sub>1</sub> <sup>0</sup>	M <sub>5</sub> <sup>4</sup>	N <sub>5</sub> <sup>4</sup>	O <sub>5</sub> <sup>4</sup>	P <sub>5</sub> <sup>4</sup>	M <sub>9</sub> <sup>8</sup>	N <sub>9</sub> <sup>8</sup>	O <sub>9</sub> <sup>8</sup>	P <sub>9</sub> <sup>8</sup>	M <sub>13</sub> <sup>12</sup>	N <sub>13</sub> <sup>12</sup>	O <sub>13</sub> <sup>12</sup>	P <sub>13</sub> <sup>12</sup>
4	O <sub>0</sub> <sup>0</sup>	P <sub>0</sub> <sup>0</sup>	O <sub>2</sub> <sup>2</sup>	P <sub>2</sub> <sup>2</sup>	O <sub>4</sub> <sup>4</sup>	P <sub>4</sub> <sup>4</sup>	O <sub>6</sub> <sup>6</sup>	P <sub>6</sub> <sup>6</sup>	O <sub>8</sub> <sup>8</sup>	P <sub>8</sub> <sup>8</sup>	O <sub>10</sub> <sup>10</sup>	P <sub>10</sub> <sup>10</sup>	O <sub>12</sub> <sup>12</sup>	P <sub>12</sub> <sup>12</sup>	O <sub>14</sub> <sup>14</sup>	P <sub>14</sub> <sup>14</sup>
S	A <sub>0</sub> <sup>15</sup>	B <sub>0</sub> <sup>15</sup>	C <sub>0</sub> <sup>15</sup>	D <sub>0</sub> <sup>15</sup>	E <sub>0</sub> <sup>15</sup>	F <sub>0</sub> <sup>15</sup>	G <sub>0</sub> <sup>15</sup>	H <sub>0</sub> <sup>15</sup>	I <sub>0</sub> <sup>15</sup>	J <sub>0</sub> <sup>15</sup>	K <sub>0</sub> <sup>15</sup>	L <sub>0</sub> <sup>15</sup>	M <sub>0</sub> <sup>15</sup>	N <sub>0</sub> <sup>15</sup>	O <sub>0</sub> <sup>15</sup>	P <sub>0</sub> <sup>15</sup>

Fig. 6. How the tree and total sum for lane 0 are embedded within the butterfly-patterned table.

```

int r = threadIdx.x & 0x1f; /* lane ID */
for (int b=1; b < W; b+=b) { /* 1,2,4,8,... */
    for (int j=0; j < (W>>(b+1)); j++) {
        d = (((j << 1) + 1) << b) - 1;
        h = (r & (1<<b)) ? a[d] : a[d+(1<<b)];
        v = __shfl_xor(h, 1<<b);
        if (r & (1<<b)) a[d] = a[d+(1<<b)];
        a[d+(1<<b)] = a[d] + v;
        p[d] = a[d];
    }
    p[W-1] = a[W-1];
}

```

Because the two loops are unrolled (the outer loop we actually unrolled manually, and the CUDA compiler then automatically unrolls the inner one), many of the expressions are reduced to constants or hoisted out of the loops at compile time. As a result, the if statement represents a single conditional-move operation, whereas in the first version of the code, the if-else statement represents two conditional-move operations. Eliminating one conditional-move operation results in a substantial speed improvement (about 15%). The resulting table has a slightly more complicated pattern (see Figure 8), which in turn requires a slightly more complicated version of the binary search code, one that either adds or subtracts at each step, and whether to add or subtract at a given step depends on a bit of the lane number (that is, at each step some lanes must add while others subtract). This alternate strategy is the one we ultimately used to achieve best speed in our measurements, and is presented in more detail in Section 6.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	$A_0^0$	$B_0^0$	$A_2^2$	$B_2^2$	$A_4^4$	$B_4^4$	$A_6^6$	$B_6^6$	$A_8^8$	$B_8^8$	$A_{10}^{10}$	$B_{10}^{10}$	$A_{12}^{12}$	$B_{12}^{12}$	$A_{14}^{14}$	$B_{14}^{14}$
3	$A_0^1$	$B_0^1$	$C_0^1$	$D_0^1$	$A_5^5$	$B_4^5$	$C_4^5$	$D_4^5$	$A_8^9$	$B_8^9$	$C_8^9$	$D_8^9$	$A_{13}^{13}$	$B_{12}^{13}$	$C_{13}^{13}$	$D_{12}^{12}$
4	$C_0^0$	$D_0^0$	$C_2^2$	$D_2^2$	$C_4^4$	$D_4^4$	$C_6^6$	$D_6^6$	$C_8^8$	$D_8^8$	$C_{10}^{10}$	$D_{10}^{10}$	$C_{12}^{12}$	$D_{12}^{12}$	$C_{14}^{14}$	$D_{14}^{14}$
2	$A_3^2$	$B_0^2$	$C_0^2$	$D_0^2$	$E_0^2$	$F_3^3$	$G_3^3$	$H_3^3$	$A_{11}^{11}$	$B_8^{11}$	$C_8^{11}$	$D_8^{11}$	$E_8^{11}$	$F_8^{11}$	$G_{11}^{11}$	$H_8^{11}$
4	$F_0^0$	$E_2^2$	$F_2^2$	$E_4^4$	$F_4^4$	$E_6^6$	$F_6^6$	$E_8^8$	$F_8^8$	$E_{10}^{10}$	$F_{10}^{10}$	$E_{12}^{12}$	$F_{12}^{12}$	$E_{14}^{14}$	$F_{14}^{14}$	
3	$F_0^1$	$G_0^1$	$H_0^1$	$I_4^4$	$F_4^5$	$G_4^5$	$H_4^5$	$I_8^9$	$F_8^9$	$G_8^9$	$H_8^9$	$I_{12}^{13}$	$F_{12}^{13}$	$G_{13}^{13}$	$H_{12}^{12}$	
4	$G_0^0$	$H_0^0$	$G_2^2$	$H_2^2$	$G_4^4$	$H_4^4$	$G_6^6$	$H_6^6$	$G_8^8$	$H_8^8$	$G_{10}^{10}$	$H_{10}^{10}$	$G_{12}^{12}$	$H_{12}^{12}$	$G_{14}^{14}$	$H_{14}^{14}$
1	$A_7^7$	$B_7^7$	$C_7^7$	$D_7^7$	$E_7^7$	$F_7^7$	$G_0^7$	$H_7^7$	$I_0^7$	$J_0^7$	$K_0^7$	$L_0^7$	$M_0^7$	$N_0^7$	$O_0^7$	$P_0^7$
4	$I_0^0$	$J_0^0$	$I_2^2$	$J_2^2$	$I_4^4$	$J_4^4$	$I_6^6$	$J_6^6$	$I_8^8$	$J_8^8$	$I_{10}^{10}$	$J_{10}^{10}$	$I_{12}^{12}$	$J_{12}^{12}$	$I_{14}^{14}$	$J_{14}^{14}$
3	$I_0^1$	$J_0^1$	$K_0^1$	$L_0^1$	$I_4^5$	$J_4^5$	$K_4^5$	$L_4^5$	$I_8^9$	$J_8^9$	$K_8^9$	$L_8^9$	$I_{12}^{13}$	$J_{12}^{13}$	$K_{13}^{13}$	$L_{12}^{12}$
4	$K_0^0$	$L_0^0$	$K_2^2$	$L_2^2$	$K_4^4$	$L_4^4$	$K_6^6$	$L_6^6$	$K_8^8$	$L_8^8$	$K_{10}^{10}$	$L_{10}^{10}$	$K_{12}^{12}$	$L_{12}^{12}$	$K_{14}^{14}$	$L_{14}^{14}$
2	$I_0^3$	$J_0^3$	$K_0^3$	$L_0^3$	$M_0^3$	$N_0^3$	$O_0^3$	$P_0^3$	$I_8^{11}$	$J_8^{11}$	$K_8^{11}$	$L_8^{11}$	$M_8^{11}$	$N_8^{11}$	$O_8^{11}$	$P_8^{11}$
4	$M_0^0$	$N_0^0$	$M_2^2$	$N_2^2$	$M_4^4$	$N_4^4$	$M_6^6$	$N_6^6$	$M_8^8$	$N_8^8$	$M_{10}^{10}$	$N_{10}^{10}$	$M_{12}^{12}$	$N_{12}^{12}$	$M_{14}^{14}$	$N_{14}^{14}$
3	$M_0^1$	$N_0^1$	$O_0^1$	$P_0^1$	$M_4^5$	$N_4^5$	$O_4^5$	$P_4^5$	$M_8^9$	$N_8^9$	$O_8^9$	$P_8^9$	$M_{12}^{13}$	$N_{12}^{13}$	$O_{12}^{13}$	$P_{12}^{13}$
4	$O_0^0$	$P_0^0$	$O_2^2$	$P_2^2$	$O_4^4$	$P_4^4$	$O_6^6$	$P_6^6$	$O_8^8$	$P_8^8$	$O_{10}^{10}$	$P_{10}^{10}$	$O_{12}^{12}$	$P_{12}^{12}$	$O_{14}^{14}$	$P_{14}^{14}$
S	$A_0^{15}$	$B_0^{15}$	$C_0^{15}$	$D_0^{15}$	$E_0^{15}$	$F_0^{15}$	$G_0^{15}$	$H_0^{15}$	$I_0^{15}$	$J_0^{15}$	$K_0^{15}$	$L_0^{15}$	$M_0^{15}$	$N_0^{15}$	$O_0^{15}$	$P_0^{15}$

Fig. 7. How the tree and total sum for lane F are embedded within the butterfly-patterned table.

The choice of whether to replace  $\left[ \begin{smallmatrix} a & b \\ c & d \end{smallmatrix} \right]$  with  $\left[ \begin{smallmatrix} a & c \\ a+b & c+d \end{smallmatrix} \right]$  or with  $\left[ \begin{smallmatrix} a & d \\ a+b & c+d \end{smallmatrix} \right]$  (or possibly some other related pattern of values) depends on details of GPU architecture and implementation. It might well be that for another GPU architecture or another generation of NVIDIA GPU implementation, a different choice may be preferable for best speed. If a future architecture happens to provide a faster way to transpose a  $W \times W$  matrix within the registers (perhaps by storing registers into a memory and then reloading them with a different access pattern, or by providing a mechanism to “index into the registers” so that each lane can access a different one of its own registers during a single SIMD instruction), then that should also be considered. In our experiments, storing and reloading proved to be much slower, and the GPU architecture we used did not support indexing into the registers, which would allow a  $W \times W$  matrix such as  $a_{\text{reg}}$  ( $W$  elements in each of  $W$  lanes) to be transposed in place using just  $W - 1$  shuffle operations (rather than the  $3(W - 1)$  required by the butterfly pattern) and almost no computational overhead:

```

1: for  $k$  from 1 through  $W - 1$  do
2:    $a_{\text{reg}}[\text{myLaneId} \oplus k] := \text{__shfl\_xor}(a_{\text{reg}}[\text{myLaneId} \oplus k], k)$ 
3: end for

```

#### 4 USE IN AN LDA APPLICATION

In Sections 4, 5, and 6, we show how to use these sampling techniques in the context of a kernel for sampling  $z$  values for a complete machine learning application.

For a Latent Dirichlet Allocation model of, for example, a set of documents to which we want to assign topics (one topic to each document) probabilistically using Gibbs sampling, let  $M$  be the number of documents,  $K$  be the number of topics, and  $V$  be the size of the vocabulary, which is a set of distinct words. Each document is a bag of words, each of which belongs to the vocabulary; any given word can appear in any number of documents, and may appear any number of times in any single document. The documents may be of different lengths.

lane	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
4	A <sub>0</sub> <sup>0</sup>	B <sub>1</sub> <sup>1</sup>	A <sub>2</sub> <sup>2</sup>	B <sub>3</sub> <sup>3</sup>	A <sub>4</sub> <sup>4</sup>	B <sub>5</sub> <sup>5</sup>	A <sub>6</sub> <sup>6</sup>	B <sub>7</sub> <sup>7</sup>	A <sub>8</sub> <sup>8</sup>	B <sub>9</sub> <sup>9</sup>	A <sub>10</sub> <sup>10</sup>	B <sub>11</sub> <sup>11</sup>	A <sub>12</sub> <sup>12</sup>	B <sub>13</sub> <sup>13</sup>	A <sub>14</sub> <sup>14</sup>	B <sub>15</sub> <sup>15</sup>
3	A <sub>0</sub> <sup>1</sup>	B <sub>0</sub> <sup>1</sup>	C <sub>2</sub> <sup>3</sup>	D <sub>2</sub> <sup>3</sup>	A <sub>4</sub> <sup>5</sup>	B <sub>4</sub> <sup>5</sup>	C <sub>6</sub> <sup>7</sup>	D <sub>6</sub> <sup>7</sup>	A <sub>8</sub> <sup>9</sup>	B <sub>8</sub> <sup>10</sup>	C <sub>10</sub> <sup>11</sup>	D <sub>10</sub> <sup>11</sup>	A <sub>12</sub> <sup>13</sup>	B <sub>12</sub> <sup>13</sup>	C <sub>14</sub> <sup>15</sup>	D <sub>14</sub> <sup>15</sup>
4	C <sub>0</sub> <sup>0</sup>	D <sub>1</sub> <sup>1</sup>	C <sub>2</sub> <sup>2</sup>	D <sub>3</sub> <sup>3</sup>	C <sub>4</sub> <sup>4</sup>	D <sub>5</sub> <sup>5</sup>	C <sub>6</sub> <sup>6</sup>	D <sub>7</sub> <sup>7</sup>	C <sub>8</sub> <sup>8</sup>	D <sub>9</sub> <sup>9</sup>	C <sub>10</sub> <sup>10</sup>	D <sub>11</sub> <sup>11</sup>	C <sub>12</sub> <sup>12</sup>	D <sub>13</sub> <sup>13</sup>	C <sub>14</sub> <sup>14</sup>	D <sub>15</sub> <sup>15</sup>
2	A <sub>0</sub> <sup>3</sup>	B <sub>0</sub> <sup>3</sup>	C <sub>0</sub> <sup>3</sup>	D <sub>0</sub> <sup>3</sup>	E <sub>7</sub> <sup>4</sup>	F <sub>7</sub> <sup>7</sup>	G <sub>4</sub> <sup>7</sup>	H <sub>4</sub> <sup>7</sup>	A <sub>8</sub> <sup>11</sup>	B <sub>8</sub> <sup>11</sup>	C <sub>8</sub> <sup>11</sup>	D <sub>8</sub> <sup>11</sup>	E <sub>12</sub> <sup>15</sup>	F <sub>12</sub> <sup>15</sup>	G <sub>12</sub> <sup>15</sup>	H <sub>12</sub> <sup>15</sup>
4	E <sub>0</sub> <sup>0</sup>	F <sub>1</sub> <sup>1</sup>	E <sub>2</sub> <sup>2</sup>	F <sub>3</sub> <sup>3</sup>	E <sub>4</sub> <sup>4</sup>	F <sub>5</sub> <sup>5</sup>	E <sub>6</sub> <sup>6</sup>	F <sub>7</sub> <sup>7</sup>	E <sub>8</sub> <sup>9</sup>	F <sub>9</sub> <sup>9</sup>	E <sub>10</sub> <sup>10</sup>	F <sub>11</sub> <sup>11</sup>	E <sub>12</sub> <sup>12</sup>	F <sub>13</sub> <sup>13</sup>	E <sub>14</sub> <sup>14</sup>	F <sub>15</sub> <sup>15</sup>
3	E <sub>0</sub> <sup>1</sup>	F <sub>0</sub> <sup>1</sup>	G <sub>2</sub> <sup>3</sup>	H <sub>2</sub> <sup>3</sup>	E <sub>5</sub> <sup>4</sup>	F <sub>5</sub> <sup>5</sup>	G <sub>6</sub> <sup>7</sup>	H <sub>6</sub> <sup>7</sup>	E <sub>8</sub> <sup>9</sup>	F <sub>8</sub> <sup>9</sup>	G <sub>10</sub> <sup>11</sup>	H <sub>10</sub> <sup>11</sup>	E <sub>12</sub> <sup>13</sup>	F <sub>12</sub> <sup>13</sup>	G <sub>14</sub> <sup>15</sup>	H <sub>14</sub> <sup>15</sup>
4	G <sub>0</sub> <sup>0</sup>	H <sub>1</sub> <sup>1</sup>	G <sub>2</sub> <sup>2</sup>	H <sub>3</sub> <sup>3</sup>	G <sub>4</sub> <sup>4</sup>	H <sub>5</sub> <sup>5</sup>	G <sub>6</sub> <sup>6</sup>	H <sub>7</sub> <sup>7</sup>	G <sub>8</sub> <sup>8</sup>	H <sub>9</sub> <sup>9</sup>	G <sub>10</sub> <sup>10</sup>	H <sub>11</sub> <sup>11</sup>	G <sub>12</sub> <sup>12</sup>	H <sub>13</sub> <sup>13</sup>	G <sub>14</sub> <sup>14</sup>	H <sub>15</sub> <sup>15</sup>
1	A <sub>0</sub> <sup>7</sup>	B <sub>0</sub> <sup>7</sup>	C <sub>0</sub> <sup>7</sup>	D <sub>0</sub> <sup>7</sup>	E <sub>0</sub> <sup>7</sup>	F <sub>0</sub> <sup>7</sup>	G <sub>0</sub> <sup>7</sup>	H <sub>0</sub> <sup>7</sup>	I <sub>0</sub> <sup>15</sup>	J <sub>0</sub> <sup>15</sup>	K <sub>8</sub> <sup>15</sup>	L <sub>8</sub> <sup>15</sup>	M <sub>8</sub> <sup>15</sup>	N <sub>8</sub> <sup>15</sup>	O <sub>8</sub> <sup>15</sup>	P <sub>8</sub> <sup>15</sup>
4	I <sub>0</sub> <sup>0</sup>	J <sub>1</sub> <sup>1</sup>	I <sub>2</sub> <sup>2</sup>	J <sub>3</sub> <sup>3</sup>	I <sub>4</sub> <sup>4</sup>	J <sub>5</sub> <sup>5</sup>	I <sub>6</sub> <sup>6</sup>	J <sub>7</sub> <sup>7</sup>	I <sub>8</sub> <sup>9</sup>	J <sub>9</sub> <sup>9</sup>	I <sub>10</sub> <sup>10</sup>	J <sub>11</sub> <sup>11</sup>	I <sub>12</sub> <sup>12</sup>	J <sub>13</sub> <sup>13</sup>	I <sub>14</sub> <sup>14</sup>	J <sub>15</sub> <sup>15</sup>
3	I <sub>0</sub> <sup>1</sup>	J <sub>0</sub> <sup>1</sup>	K <sub>2</sub> <sup>3</sup>	L <sub>2</sub> <sup>3</sup>	I <sub>4</sub> <sup>5</sup>	J <sub>4</sub> <sup>5</sup>	K <sub>6</sub> <sup>7</sup>	L <sub>6</sub> <sup>7</sup>	I <sub>8</sub> <sup>9</sup>	J <sub>9</sub> <sup>9</sup>	K <sub>10</sub> <sup>11</sup>	L <sub>10</sub> <sup>11</sup>	I <sub>12</sub> <sup>13</sup>	J <sub>12</sub> <sup>13</sup>	K <sub>14</sub> <sup>15</sup>	L <sub>14</sub> <sup>15</sup>
4	K <sub>0</sub> <sup>0</sup>	L <sub>1</sub> <sup>1</sup>	K <sub>2</sub> <sup>2</sup>	L <sub>3</sub> <sup>3</sup>	K <sub>4</sub> <sup>4</sup>	L <sub>5</sub> <sup>5</sup>	K <sub>6</sub> <sup>6</sup>	L <sub>7</sub> <sup>7</sup>	K <sub>8</sub> <sup>9</sup>	L <sub>9</sub> <sup>9</sup>	K <sub>10</sub> <sup>10</sup>	L <sub>11</sub> <sup>11</sup>	K <sub>12</sub> <sup>12</sup>	L <sub>13</sub> <sup>13</sup>	K <sub>14</sub> <sup>14</sup>	L <sub>15</sub> <sup>15</sup>
2	I <sub>0</sub> <sup>3</sup>	J <sub>0</sub> <sup>3</sup>	K <sub>0</sub> <sup>3</sup>	L <sub>0</sub> <sup>3</sup>	M <sub>7</sub> <sup>4</sup>	N <sub>7</sub> <sup>7</sup>	O <sub>4</sub> <sup>7</sup>	P <sub>7</sub> <sup>7</sup>	I <sub>8</sub> <sup>11</sup>	J <sub>8</sub> <sup>11</sup>	K <sub>8</sub> <sup>11</sup>	L <sub>8</sub> <sup>11</sup>	M <sub>12</sub> <sup>15</sup>	N <sub>12</sub> <sup>15</sup>	O <sub>12</sub> <sup>15</sup>	P <sub>12</sub> <sup>15</sup>
4	M <sub>0</sub> <sup>0</sup>	N <sub>1</sub> <sup>1</sup>	M <sub>2</sub> <sup>2</sup>	N <sub>3</sub> <sup>3</sup>	M <sub>4</sub> <sup>4</sup>	N <sub>5</sub> <sup>5</sup>	M <sub>6</sub> <sup>6</sup>	N <sub>7</sub> <sup>7</sup>	M <sub>8</sub> <sup>8</sup>	N <sub>9</sub> <sup>9</sup>	M <sub>10</sub> <sup>10</sup>	N <sub>11</sub> <sup>11</sup>	M <sub>12</sub> <sup>13</sup>	N <sub>13</sub> <sup>14</sup>	M <sub>14</sub> <sup>15</sup>	N <sub>15</sub> <sup>15</sup>
3	M <sub>0</sub> <sup>1</sup>	N <sub>0</sub> <sup>1</sup>	O <sub>2</sub> <sup>3</sup>	P <sub>2</sub> <sup>3</sup>	M <sub>4</sub> <sup>5</sup>	N <sub>4</sub> <sup>5</sup>	O <sub>6</sub> <sup>7</sup>	P <sub>7</sub> <sup>7</sup>	M <sub>8</sub> <sup>9</sup>	N <sub>8</sub> <sup>9</sup>	O <sub>10</sub> <sup>11</sup>	P <sub>10</sub> <sup>11</sup>	M <sub>12</sub> <sup>13</sup>	N <sub>12</sub> <sup>13</sup>	O <sub>14</sub> <sup>15</sup>	P <sub>14</sub> <sup>15</sup>
4	O <sub>0</sub> <sup>0</sup>	P <sub>1</sub> <sup>1</sup>	O <sub>2</sub> <sup>2</sup>	P <sub>3</sub> <sup>3</sup>	O <sub>4</sub> <sup>4</sup>	P <sub>5</sub> <sup>5</sup>	O <sub>6</sub> <sup>6</sup>	P <sub>7</sub> <sup>7</sup>	O <sub>8</sub> <sup>8</sup>	P <sub>9</sub> <sup>9</sup>	O <sub>10</sub> <sup>10</sup>	P <sub>11</sub> <sup>11</sup>	O <sub>12</sub> <sup>12</sup>	P <sub>13</sub> <sup>13</sup>	O <sub>14</sub> <sup>14</sup>	P <sub>15</sub> <sup>15</sup>
S	A <sub>0</sub> <sup>15</sup>	B <sub>0</sub> <sup>15</sup>	C <sub>0</sub> <sup>15</sup>	D <sub>0</sub> <sup>15</sup>	E <sub>0</sub> <sup>15</sup>	F <sub>0</sub> <sup>15</sup>	G <sub>0</sub> <sup>15</sup>	H <sub>0</sub> <sup>15</sup>	I <sub>0</sub> <sup>15</sup>	J <sub>0</sub> <sup>15</sup>	K <sub>0</sub> <sup>15</sup>	L <sub>0</sub> <sup>15</sup>	M <sub>0</sub> <sup>15</sup>	N <sub>0</sub> <sup>15</sup>	O <sub>0</sub> <sup>15</sup>	P <sub>0</sub> <sup>15</sup>

Fig. 8. Alternate “add-subtract” butterfly pattern for  $p'$ .

We are interested in the phase of an uncollapsed Gibbs sampler that draws new  $z$  values, given  $\theta$  and  $\phi$  distributions. Because no  $z$  value directly depends on any other  $z$  value in this formulation, new  $z$  values may all be computed independently (and therefore in parallel to any extent desired).

We assume that we are given an  $M \times K$  matrix  $\theta$  and a  $V \times K$  matrix  $\phi$ ; the elements of these matrices are non-negative numbers, typically represented as floating-point values. Row  $m$  of  $\theta$  (that is,  $\theta[m, \cdot]$ ) is the (currently assumed) distribution of topics for document  $m$ , that is, the relative probabilities (weights) for each of the  $K$  possible topics to which the document might be assigned. Note that columns of  $\theta$  are *not* to be considered as distributions. Similarly, column  $k$  of  $\phi$  (that is,  $\phi[\cdot, k]$ ) is the (currently assumed) distribution of words for topic  $k$ , that is, the weights with which the  $V$  possible words in the vocabulary are associated with the topic. Note that rows of  $\phi$  are *not* to be considered as distributions. We organize  $\theta$  as rows and  $\phi$  as columns for engineering reasons: We want the  $K$  entries obtained by ranging over all possible topics to be contiguous in memory to take advantage of memory cache structure.

(For presentation purposes, we likewise present array and matrices in transposed form in this article; in every diagram depicting a two-dimensional array, the axis indexed by  $K$  runs vertically.)

We also assume that we are given (i) a length- $M$  vector of nonnegative integers  $N$  such that  $N[m]$  is the number of words in document  $m$ , and (ii) an  $M \times N$  ragged array  $w$ , by which we mean that for  $0 \leq m < M$ ,  $w[m]$  is a vector of length  $N[m]$ . Each element of  $w$  is a word type (a nonnegative integer less than  $V$ ) and may therefore be used as a first index for  $\phi$ . Our goal, given  $K$ ,  $M$ ,  $V$ ,  $N$ ,  $\phi$ ,  $\theta$ , and  $w$  and assuming the use of a temporary  $M \times N \times K$  ragged work array  $a$  (which we will later optimize away), is to compute all the elements for an  $M \times N$  ragged array  $z$  as follows: For all  $m$  such that  $0 \leq m < M$  and for all  $i$  such that  $0 \leq i < N[m]$ , do two things: First, for all  $k$  such that  $0 \leq k < K$ , let  $a[m][i][k] = \theta[m, k] \times \phi[w[m, i], k]$ ; second, let  $z[m, i]$  be a nonnegative integer less than  $K$ , chosen randomly in such a way that the probability of choosing the value  $k'$  is  $a[m][i][k']/\sigma$  where  $\sigma = \sum_{0 \leq k < K} a[m][i][k]$ . Thus,  $a[m][i][k']$  is a relative (unnormalized) probability, and  $a[m][i][k']/\sigma$  is an absolute (normalized) probability.

---

**ALGORITHM 1:** Drawing new  $z$  values

---

```

1: procedure DRAWZ( $N[M]$ ,  $\theta[M, K]$ ,  $\phi[V, K]$ ,  $w[M][N]$ ; output  $z[M, N]$ )
2:   local array  $a[M][N][K], p[M][N][K]$ 
3:   for all  $0 \leq m < M$  do
4:     for all  $0 \leq i < N[m]$  do
5:        $\triangleright$  Compute  $\theta$ - $\phi$  products
6:       for all  $0 \leq k < K$  do
7:          $a[m][i][k] := \theta[m, k] \times \phi[w[m][i], k]$ 
8:       end for
9:        $\triangleright$  Compute partial sums of the products
10:      let  $sum = 0.0$ 
11:      for  $k$  from 0 through  $K - 1$  do
12:         $sum += a[m][i][k]$ 
13:         $p[m][i][k] := sum$ 
14:      end for
15:      let  $j = 0$ 
16:       $\langle$ search the table  $p[m][i]$  of partial sums $\rangle$ 
17:       $z[m, i] := j$ 
18:    end for
19:  end for
20: end procedure

```

---



---

**ALGORITHM 2:** Simple linear search

---

```

1: code chunk  $\langle$ search the table  $P$  of partial sums $\rangle$ 
2:   let  $u$  = value chosen uniformly at random (or pseudorandomly) from  $[0.0, 1.0)$ 
3:   let  $u' = sum \times u$ 
4:   while  $j < K - 1$  and  $u' \geq P[j]$  do  $j += 1$ 
5: end

```

---

Algorithm 1 is a basic implementation of this process. We remark that a “**let**” statement creates a local binding of a scalar (single-valued) variable and gives it a value; that a “**local array**” declaration creates a local binding of an array variable, containing an (initially undefined) element value for each indexable position in the array; and that distinct iterations of any containing “**for**” or “**for all**” construct create distinct and independent instantiations of such local variables. The iterations of “**for ... from ... through ...**” are executed sequentially; but the iterations of a “**for all**” construct are intended to be computationally independent and therefore may be executed in any order, or in parallel, or in any sequential-parallel combination. We use angle brackets to indicate the use of a “code chunk” that is defined as a separate algorithm; such a use indicates that the definition of the code chunk should be inserted at the use site, possibly with parameter substitution, as if it were a C macro, but surrounded by **begin** and **end** (this programming-language technicality ensures that the scope of any variable declared within the code chunk is confined to that code chunk).

The computation of the  $\theta$ - $\phi$  products (lines 6–8 of Algorithm 1) is straightforward. The computation of partial sums (lines 10–14) is sequential; the variable  $sum$  accumulates the products, and successive values of  $sum$  are stored into the array  $p$ . A random integer is chosen for  $z[m, i]$  by choosing a random value uniformly from the range  $[0, 0, 1.0)$ , scaling it by the final value of  $sum$  (which has the same algorithmic effect as dividing each  $p[m][i][k]$  by that value, for all  $0 \leq k < K$ , to turn it into an absolute probability), and then searching the subarray  $p[m][i]$  to find the smallest entry that is larger than the scaled value (and if there are several such entries, all equal, then

---

**ALGORITHM 3:** Simple binary search

---

```

1: code chunk <search the table  $P$  of partial sums>:
2:   let  $u$  = value chosen uniformly at random (or pseudorandomly) from  $[0.0, 1.0)$ 
3:   let  $u' = sum \times u$ 
4:   let  $k = K - 1$ 
5:   while  $j < k$  do
6:     let  $mid = \left\lfloor \frac{j + k}{2} \right\rfloor$ 
7:     if  $u' < P[mid]$  then  $k := mid$ 
8:     else  $j := mid + 1$ 
9:   end while
10: end

```

---

the one with the smallest index is chosen); the index  $j$  of that entry is used as the desired randomly chosen integer. A simple linear search (Algorithm 2) can do the job, but a binary search (Algorithm 3) can be used instead, which is faster, on average, for  $K$  sufficiently large [13, exercise 6.2.1-5].

## 5 BLOCKING AND TRANSPOSITION

Anticipating certain characteristics of the hardware, we now make some commitments as to how the algorithm will be executed. We assume that arrays are laid out in row-major order (as they are when using C or CUDA). Let  $W$  be a machine-dependent constant (typically 16 or 32, but for now we do not require that  $W$  be a power of 2). For purposes of illustration, we assume  $W = 16$  and  $K = 71$ . We divide the documents into groups of size  $W$  and assume that  $M$  is an exact multiple of  $W$ . (In the overall application, the set of documents can be padded with empty documents to make  $M$  be an exact multiple of  $W$  without affecting the overall behavior of the algorithm on the “real” documents.) We turn the outermost loop of Algorithm 1 (with index variable  $m$ ) into two nested loops with index variables  $q$  and  $r$ , from which the equivalent value for  $m$  is then computed. We commit to making the loop with index variable  $i$  sequential, to treating the iterations of the loop on  $q$  as independent (and therefore possibly parallel), and to treating the iterations of the loop on  $r$  as executed by a SIMD “thread warp” of size  $W$ , that is, parallel and implicitly lock-step synchronized. As a result, we view each of the  $M$  documents as being processed by a separate thread. A benefit of making the loop on  $i$  sequential is that the array  $p$  can be made two-dimensional and non-ragged, having size  $M \times K$ . We fuse the loop that computes  $\theta \cdot \phi$  products with the loop that computes partial sums; this eliminates the need for the array  $a$ , but instead (for reasons explained below), we use  $a_{\text{local}}$  as a two-dimensional, non-ragged array of size  $M \times W$  that is used only when  $K \geq W$ . Within the loop on  $q$ , we declare a local work array  $c_{\text{warp}}$  of size  $W \times W$  that will be used to exchange information by the  $W$  threads within a warp; our eventual intent is that this array will reside in GPU registers. We cache values from the array  $\theta$  in a per-thread array  $\theta_{\text{local}}$  of length  $K$ , anticipating that such cached values will reside in a faster memory and be used repeatedly by the loop on  $i$ .

There is, however, a subtle problem with the loop controlling index variable  $i$ : the upper bound  $N[m]$  for the loop variable may be different for different threads. As a result, in the last iterations it may be that some threads have “gone to sleep,” because they reached their upper loop bound earlier than other threads in the warp. This is undesirable, because, as we shall see, we rely on all threads “staying awake” so that they can assist each other. Therefore, we rewrite the loop control to use a “master index” idiom and exploit the trick of allowing a thread to perform its last iteration

---

**ALGORITHM 4:** Drawing  $z$  values (transposed access)

---

```

1: procedure DRAWZ( $N[M]$ ,  $\theta[M, K]$ ,  $\phi[V, K]$ ,  $w[M][N]$ ; output  $z[M, N]$ )
2:   local array  $p_{\text{local}}[M][K]$ ,  $a_{\text{local}}[M][W]$ 
3:   for all  $0 \leq q < M/W$  do
4:     local array  $c_{\text{warp}}[W, W]$ 
5:     for SIMD  $0 \leq r < W$  do
6:       let  $m = q \times W + r$ 
7:       local array  $\theta_{\text{local}}[K]$ 
8:       ⟨cache  $\theta$  values into  $\theta_{\text{local}}$ ⟩
9:       let  $i_{\text{master}} = 0$ 
10:      while any( $i_{\text{master}} < N[m]$ ) do
11:        let  $i = \min(i_{\text{master}}, N[m] - 1)$ 
12:        ⟨compute partial sums of  $\theta$ - $\phi$  products⟩
13:        let  $j = 0$ 
14:        ⟨search the table  $p_{\text{local}}[m]$  of partial sums⟩
15:         $z[m, i] := j$ 
16:         $i_{\text{master}} += 1$ 
17:      end while
18:    end for
19:  end for
20: end procedure

```

---



---

**ALGORITHM 5:** Caching  $\theta$  values (transposed access)

---

```

1: code chunk ⟨cache  $\theta$  values into  $\theta_{\text{local}}$ ⟩:
2:   let  $j = 0$ 
3:   while  $j < (K \bmod W)$  do ▷ Cache the remnant
4:      $\theta_{\text{local}}[j] := j1$ 
5:      $\theta[m, j] += j1$ 
6:   end while
7:   while  $j < K$  do ▷ Cache all  $W \times W$  blocks
8:     for  $k$  from 0 through  $W - 1$  do
9:       ▷ Next line uses transposed access to  $\theta$ 
10:       $\theta_{\text{local}}[j + k] := \theta[q \times W + k, j + r]$ 
11:    end for
12:     $j += W$ 
13:  end while
14: end

```

---

(with  $i = N[m] - 1$ ) multiple times, which does not work for many algorithms but is acceptable for LDA Gibbs.

The result of all these code transformations is Algorithm 4, which makes use of three code chunks: Algorithm 5, Algorithm 6, and either Algorithm 2 or Algorithm 3. Algorithms 5 and 6, besides using SIMD thread warps of size  $W$  to process documents in groups of size  $W$ , also process topics in blocks of size  $W$ . This allows the innermost loops to process “little” arrays of size  $W \times W$ . If  $K$  (the number of topics) is not a multiple of  $W$ , then there will be a *remnant* of size  $K \bmod W$ . To make looping code slightly simpler, we put the remnant at the *front* of each array, rather than at the end. For  $W = 16$  and  $K = 71$ , topics 0, 1, 2, 3, 4, 5, and 6 form a remnant of length 7; topics 3–18 form a first block of length 16; topics 19–34 form a second block; topics 35–50 form a third

---

**ALGORITHM 6:** Compute partial sums (transposed access)

---

```

1: code chunk <compute partial sums of  $\theta\phi$  products>:
2:   let  $c = w[m][i]$ 
3:   for all  $0 \leq k < W$  do
4:      $c_{\text{warp}}[k, r] := c$                                  $\triangleright$  Transposed access to  $c_{\text{warp}}$ 
5:   end for
6:   let  $sum = 0.0$ 
7:   let  $j = 0$ 
8:   while  $j < (K \bmod W)$  do                          $\triangleright$  Process the remnant
9:      $sum += (\theta_{\text{local}}[j] \times \phi[c, j])$ 
10:     $p_{\text{local}}[m][j] := sum$ 
11:     $j += 1$ 
12:   end while
13:   while  $j < K$  do                                $\triangleright$  Process all  $W \times W$  blocks
14:     for  $k$  from 0 through  $W - 1$  do
15:        $\triangleright$  Next line uses transposed access to  $\phi$ 
16:        $a_{\text{local}}[m, k] := \theta_{\text{local}}[j + k] \times \phi[c_{\text{warp}}[r, k], j + r]$ 
17:     end for
18:     for  $k$  from 0 through  $W - 1$  do
19:        $\triangleright$  Next line uses transposed access to  $a_{\text{local}}$ , alas
20:        $sum += a_{\text{local}}[q \times W + k, r]$ 
21:        $p_{\text{local}}[m, j + k] := sum$ 
22:     end for
23:      $j += W$ 
24:   end while
25: end

```

---

block; and topics 51–66 form a fourth block (see Figure 9). This organization of arrays into blocks allows reduction of the cost of accessing data in main memory by performing *transposed accesses*.

The simplest use of transposed memory access occurs in Algorithm 5. For every document, a  $\theta$  value is fetched for every topic. The topics are regarded as divided into a leading remnant (if any) and then a sequence of blocks of length  $W$ . The **while** loop on lines 3–6 handles the remnant, and then the following **while** loop processes successive blocks. On line 10 within the inner loop, note that the reference is to  $\theta[q \times W + k, j + r]$  rather than the expected  $\theta[q \times W + r, j + k]$  (which would be the same as  $\theta[m, j + k]$ , because  $m = q \times W + r$ ). The result is that when the  $W$  threads of a SIMD warp execute this code and all access  $\theta$  simultaneously, they access  $W$  consecutive memory locations, which can typically be fetched by a hardware memory controller much more efficiently than  $W$  memory locations separated by stride  $K$ . Another way to think about it is that on any given single iteration of the loop on lines 8–11 (which overall is designed to fetch one  $W \times W$  block of  $\theta$  values) instead of every thread in the warp fetching its  $k$ th value from the  $\theta$  array, all the threads work together to fetch all  $W$  values that are needed by thread  $k$  of the warp. Each thread then stores what it has fetched into its local copy of the array  $\theta_{\text{local}}$ .

Algorithm 6 compensates for this transposition of  $\theta$ . The idea is also to divide each row of  $\phi$  into blocks (possibly preceded by a remnant) and perform transposed accesses to  $\phi$ . To do this, each thread needs to know which row of  $\phi$  every other thread is interested in; this is done through the  $W \times W$  local work array  $c_{\text{warp}}$ . In line 2, each thread figures out which word is the  $i$ th word of its document and calls it  $c$ ; in lines 3–5 it then stores its value for  $c$  into every element of row  $r$  of the array  $c_{\text{warp}}$ . This is not an especially fast operation, but it pays for itself later on. The loop in lines 8–12 computes  $\theta\phi$  products and partial sums  $p$  in the usual way (remember that the remnant

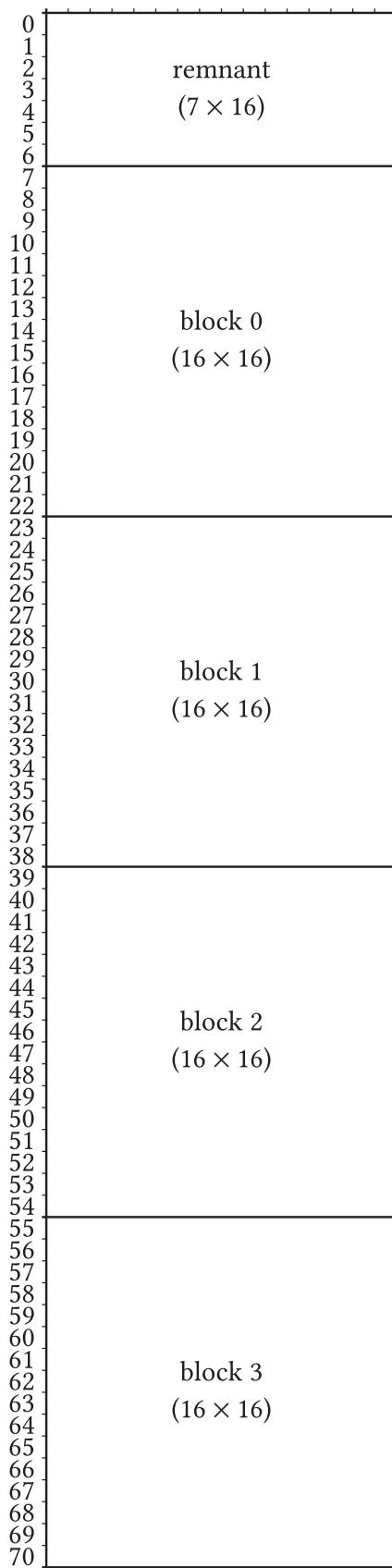


Fig. 9. Example division of an array into a remnant and four blocks ( $W = 16, K = 71$ ).

---

**ALGORITHM 7:** Drawing new  $z$  values using a butterfly table

---

```

1: procedure DRAWZ( $N[M]$ ,  $\theta[M, K]$ ,  $\phi[V, K]$ ,  $w[M][N]$ ; output  $z[M, N]$ )
2:    $\triangleright W$  (the “warp size”) must be a power of 2, and  $M$  must be a multiple of  $W$ .
3:   for all  $0 \leq q < M/W$  do
4:     for SIMD  $0 \leq r < W$  do
5:       let  $m = q \times W + r$ 
6:       local array  $p'[K], \theta_{\text{local}}[K]$ 
7:       register array  $a_{\text{reg}}[W], c_{\text{warp}}[W]$ 
8:       ⟨cache  $\theta$  values into  $\theta_{\text{local}}let  $i_{\text{master}} = 0$ 
10:      while any( $i_{\text{master}} < N[m]$ ) do
11:        let  $i = \min(i_{\text{master}}, N[m] - 1)$ 
12:        ⟨SIMD compute butterfly partial sums⟩
13:        let  $j = 0$ 
14:        ⟨SIMD search butterfly partial sums⟩
15:         $z[m, i] := j$ 
16:         $i_{\text{master}} += 1$ 
17:      end while
18:    end for
19:  end for
20: end procedure$ 
```

---

in  $\theta_{\text{local}}$  is not transposed), but the loop in lines 14–17 processes a block to compute product values to store into the  $a_{\text{local}}$  array; the access to  $\phi$  on line 16 is transposed (note that the accesses to  $\theta_{\text{local}}$  and  $c_{\text{warp}}$  are *not* transposed; because they were constructed and stored in transposed form, normal fetches cause their values to line up correctly with the  $\phi$  values obtained by a transposed fetch). So this is pretty good; but in line 20, we finally pay the piper: To have the finally computed partial sums  $p$  reside in the correct lane for the binary search, it is necessary to perform a transposed access to  $a_{\text{local}}$  on line 20; but  $a_{\text{local}}$  is a local array, so transposed accesses are bad rather than good, and this occurs in an inner loop, so performance still suffers.

## 6 USING BUTTERFLY-PATTERNEDE PARTIAL SUMS

We avoid the cost of the final transposition of  $a_{\text{local}}$  by not requiring the partial sums table  $p$  for each thread to be entirely in the local memory of that thread. Instead, for each  $W \times W$  block we use a butterfly-patterned table  $p'$ .

Our final version is Algorithm 7. It is similar to Algorithm 4, but declares all local arrays to be thread-local (and specifies that arrays  $a_{\text{reg}}$  and  $c_{\text{warp}}$  should reside in registers). It uses Algorithm 5 to cache  $\theta$  values in  $\theta_{\text{local}}$ , and also uses three new code chunks: Algorithms 8, 9, and 10. For Algorithm 7 to work properly,  $W$  must be a power of 2.

Algorithm 8 computes the butterfly-patterned table. The tricky part is the loop on lines 18–30, which is implemented by the alternate (faster) version of the CUDA code shown in Section 3, that is, the one that has each butterfly  replace  $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$  with  $\begin{bmatrix} a & d \\ a+b & c+d \end{bmatrix}$ .

Within a butterfly-patterned block of partial sums, Algorithm 9 performs a binary search as follows. The  $u'$  value is computed exactly as in Algorithms 2 and 3, and a block to be searched is identified by performing a binary search on the subarray consisting of just the last row of each block; this identifies a specific block to search. If  $K \geq W$ , then some  $W \times W$  block is identified, and it is searched, but it is possible that the desired  $u'$  value does not lie within that block; in that case, the remnant is searched using a linear search.

**ALGORITHM 8:** Compute a butterfly-patterned table of sums

---

```

1: code chunk <SIMD compute butterfly partial sums>:
2:   let  $c = w[m][i]$ 
3:   for all  $0 \leq k < W$  do
4:      $c_{\text{warp}}[k] := \text{__shfl1}(c, k)$ 
5:   end for
6:   let  $sum = 0.0$ 
7:   let  $j = 0$ 
8:   while  $j < (K \bmod W)$  do                                 $\triangleright$  Process the remnant
9:      $sum += (\theta_{\text{local}}[j] \times \phi[c, j])$ 
10:     $p'[j] := sum$ 
11:     $j := j + 1$ 
12:   end while
13:   while  $j < K$  do                                 $\triangleright$  Process all  $W \times W$  blocks
14:     for  $k$  from 0 through  $W - 1$  do
15:        $\triangleright$  Next line uses transposed access to  $\phi$ 
16:        $a_{\text{reg}}[k] := \theta_{\text{local}}[j + k] \times \phi[c_{\text{warp}}[k], j + r]$ 
17:     end for
18:     for  $b$  from 0 through  $(\log_2 W) - 1$  do
19:       let  $bit = 2^b$ 
20:       for  $i$  from 0 through  $\frac{W}{2 \times bit} - 1$  do
21:         let  $d = 2 \times bit \times i + (bit - 1)$ 
22:         let  $h = (\text{if } (m \& bit) \neq 0$ 
23:           then  $a_{\text{reg}}[d]$ 
24:           else  $a_{\text{reg}}[d + bit]\b)$ 
25:         let  $v = \text{__shfl\_xor}(h, bit)$ 
26:         if  $(r \& bit) \neq 0$  then  $a_{\text{reg}}[d] := a_{\text{reg}}[d + bit]$ 
27:          $a_{\text{reg}}[d + bit] := a_{\text{reg}}[d] + v$ 
28:          $p'[j + d] := a_{\text{reg}}[d]$ 
29:       end for
30:     end for
31:      $sum += a_{\text{reg}}[W - 1]$ 
32:      $p'[W - 1] := sum$ 
33:      $j := j + W$ 
34:   end while
35: end

```

---

To search within a block, Algorithm 10 maintains two additional state variables  $lowValue$  and  $highValue$ . An invariant is that if lane  $m$  has indices  $j$  through  $k$  of a block still under consideration, then  $lowValue = m_0^{blockBase+j-1}$  and  $highValue = m_0^{blockBase+k}$ . To cut the search range in half, the binary search needs to compare the  $u'$  value to the midpoint value  $m_0^{blockBase+mid}$  where  $mid = \lfloor \frac{j+k}{2} \rfloor$ ; in Algorithm 3 this value is of course an entry in the  $p$  array, but in Algorithm 10 the midpoint value is calculated by choosing an appropriate entry from the butterfly-patterned  $p'$  array and then either adding it to  $lowValue$  or subtracting it from  $highValue$ . Whether to add or subtract on iteration number  $b$  (where the  $\log_2 W$  iterations are numbered starting from 0) depends on whether bit  $b$  (counting from the right starting at 0) of the binary representation of  $m$  is 0 or 1, respectively. Depending on the result of the comparison of the midpoint value with the  $u'$  value, the midpoint value is assigned to either  $lowValue$  and  $highValue$ , maintaining the invariant, and

---

**ALGORITHM 9:** Searching within a butterfly-patterned table

---

```

1: code chunk ⟨SIMD search butterfly partial sums⟩:
2:   let  $u$  = value chosen uniformly at random (or pseudorandomly) from [0.0, 1.0)
3:   let  $u' = sum \times u$ 
4:   let  $j = 0$ 
5:   let  $k = \left\lfloor \frac{K}{W} \right\rfloor - 1$ 
6:   let  $searchBase = (K \bmod W) + (W - 1)$ 
7:    $\triangleright$  Binary search to find correct block of size  $W$ 
8:   while  $j < k$  do
9:     let  $mid = \lfloor \frac{j+k}{2} \rfloor$ 
10:    if  $u' < p'[mid \times W + searchBase]$  then  $k := mid$ 
11:    else  $j := mid + 1$ 
12:   end while
13:   let  $blockBase = (K \bmod W) + j \times W$ 
14:   if  $K \geq W$  then
15:     ⟨SIMD butterfly search one block⟩
16:   end if
17:   if  $blockBase > 0$  then
18:     if  $u' < p'[m, blockBase - 1]$  then
19:        $\triangleright$  Not in a block after all, so search the remnant
20:       for  $i$  from 0 through  $(K \bmod W) - 1$  do
21:         if  $u' < p'[i]$  then
22:            $j := i$ 
23:           break
24:         end if
25:       end for
26:     end if
27:   end if
28: end

```

---

a bit of a third state variable  $flip$  (initially 0) is updated. When the binary search is complete, the correct index to select is computed from the value in  $flip$ .

The threads in a warp assist one another in fetching tree nodes using the loop in lines 12–18; the function `_shfl_xor` effects this data transfer in line 16.

## 7 EVALUATION

We coded four versions of a complete LDA topic-modeling algorithm using Gibbs sampling (which is a specific kind of Markov chain Monte Carlo algorithm, or MCMC) in CUDA 6.5 for an NVIDIA Titan Black GPU ( $W = 32$ ). The LDA model has two sets of parameters,  $\theta$  and  $\phi$ , as we have already indicated; moreover, the model is designed in such a way that integrating these parameters has a closed form. Integrating a parameter introduces a tradeoff between convergence rate and the amount of parallelism of the MCMC algorithm. The four versions we consider are therefore the traditional Gibbs sampler for which both  $\theta$  and  $\phi$  are integrated out, a version in which  $\theta$  is integrated but  $\phi$  is not, a version in which  $\phi$  is integrated but  $\theta$  is not, and a version in which neither is integrated.

For each of the four versions, we tested two variants, one using Algorithm 1 (using the binary search of Algorithm 3) and one using Algorithm 7. (These algorithms are the ones on which we reported at ICML 2015 [28]; that paper includes only a passing mention of this use of butterfly-patterned partial sums, and refers to an early version of this article [24].) All eight

---

**ALGORITHM 10:** Butterfly search within one  $W \times W$  block

---

```

1: code chunk <SIMD butterfly search one block>:
2:   let lowValue = (if blockBase > 0
3:     then p'[blockBase − 1]
4:     else 0)
5:   let highValue = p'[blockBase + ( $W - 1$ )]
6:   let flip = 0
7:    $\triangleright$  Butterfly search within the block of size  $W$ 
8:   for b from 0 through  $(\log_2 W) - 1$  do
9:     let bit =  $2^{((\log_2 W) - 1) - b}$ 
10:    let mask =  $((W - 1) \times (2 \times \text{bit})) \& (W - 1)$ 
11:    let y = 0
12:    for i from 0 through  $\frac{W}{2 \times \text{bit}} - 1$  do
13:      let d =  $(\text{bit} - 1) + 2 \times \text{bit} \times i$ 
14:      let him =  $(d \& \text{mask}) + (r \& \neg \text{mask})$ 
15:      let hisBlockBase = _shfl(blockBase, him)
16:      let t = _shfl_xor(p'[hisBlockBase + d], flip)
17:      if  $((r \oplus d) \& \text{mask}) = 0$  then y := t
18:    end for
19:    let compareValue = (if  $(r \& \text{bit}) \neq 0$ 
20:      then highValue − y
21:      else lowValue + y)
22:    if stop < compareValue then
23:      highValue := compareValue
24:      flip := flip  $\oplus (\text{bit} \& r)$ 
25:    else
26:      lowValue := compareValue
27:      flip := flip  $\oplus (\text{bit} \& \neg r)$ 
28:    end if
29:  end for
30:  j := blockBase + (flip  $\oplus r$ )
31: end

```

---

variants were tested for speed using a Wikipedia-based dataset with number of documents  $M = 43,556$ , vocabulary size  $V = 37,286$ , total number of words in corpus  $\Sigma N = 3,072,662$  (therefore, average document size  $(\Sigma N)/M \approx 70.5$ ), and maximum document size  $\max N = 307$ . Each variant was measured using eight different values for the number of topics  $K$  (16, 48, 80, 112, 144, 176, 208, and 240), in each case performing 100 sampling iterations and measuring the execution time of the entire application, not just the part that draws  $z$  values. Best performance requires unrolling three loops in Algorithm 8; we had to manually unroll the loop that starts on line 18, and the CUDA compiler then automatically unrolled the loops that start on lines 14 and 20. The performance results are shown in Figure 10. The butterfly variants are faster for  $K \geq 80$ . For  $K \geq 200$ , for each of the four versions the butterfly variant is more than twice as fast.

Subsequently, we performed a more extensive set of measurements, after upgrading the CUDA software. Using CUDA 7.5 on the same hardware, we measured Algorithm 1, Algorithm 7, and a variant of Algorithm 4 that transposes in registers (as in Figure 2) for 32-bit intermediate values (Figure 11(a)) and for 64-bit intermediate values (Figure 11(b)), using the same Wikipedia-based dataset. Each of the three algorithms was measured for  $K = 4, 8, 12, 16, 20, \dots, 1020, 1024$ , again in each case performing 100 sampling iterations and measuring the execution time of the entire

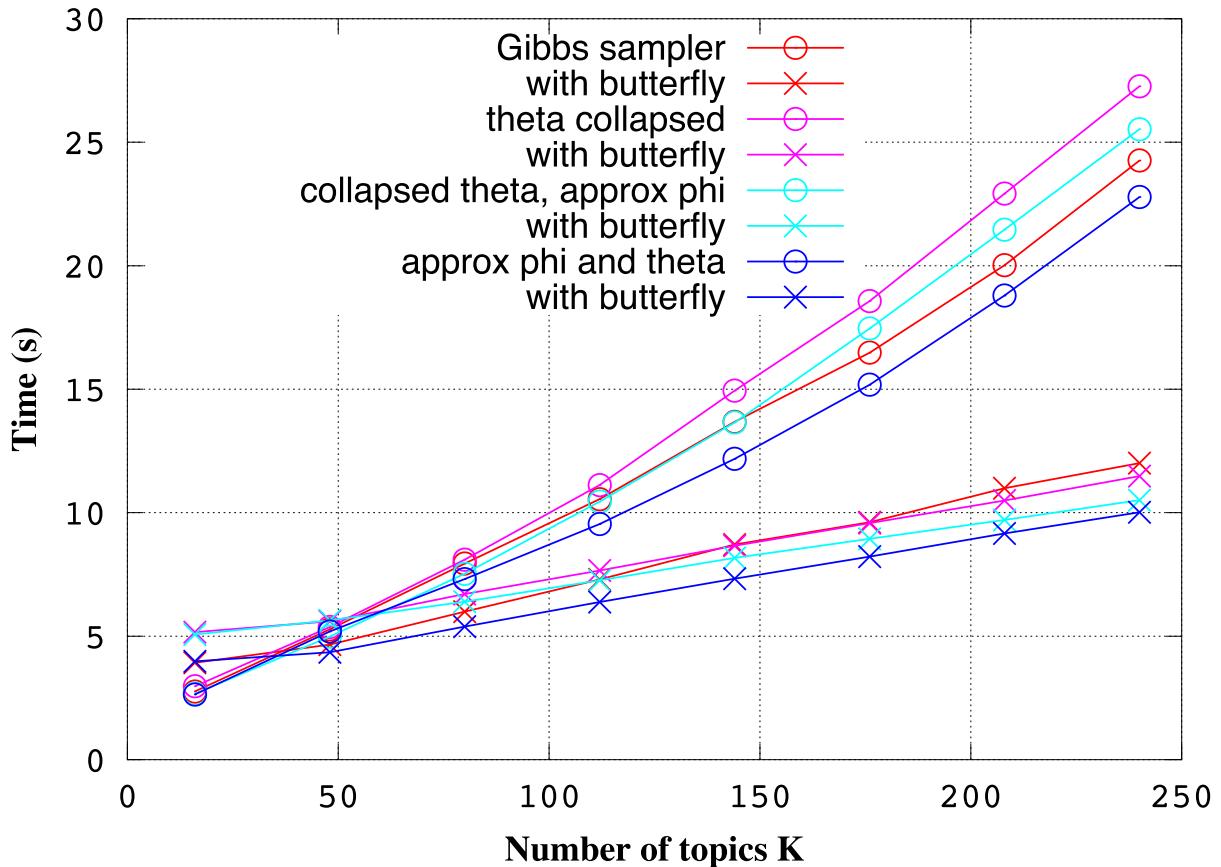


Fig. 10. Measurements of execution time, with and without butterfly-patterned partial sums, for four variations of a machine learning application (topic modeling by Gibbs sampling; CUDA 6.5;  $K = 32k + 16$ ,  $0 \leq k \leq 7$ ).

application, not just the part that draws  $z$  values. Each data point shown in Figure 11 (and in Figure 12) is an average of five runs; the maximum relative standard deviation was 0.38. One can see that the measurements for Algorithms 4 and 7 have a distinctive sawtooth pattern: the amount by which a measurement dips below an upper-bounding line depends on the number of trailing zero-bits in the binary representation of the length of the remnant (that is, the value of  $K \bmod 32$ ).

For 32-bit intermediate data (Figure 11(a)), Algorithm 7 is faster than Algorithm 4 for all  $K > 576$ ; moreover, it is also faster for all multiples of 32 greater than 64. For  $K = 512$ , Algorithm 7 is 8% faster than Algorithm 4; for  $K = 1024$ , it is 13% faster. For 64-bit intermediate data (Figure 11(b)), Algorithm 7 is faster than Algorithm 4 for all  $K > 64$ . For  $K = 512$ , Algorithm 7 is 33% faster than Algorithm 4; for  $K = 1024$ , it is 35% faster.

Measurements of just Algorithms 4 and 7, for all  $480 \leq K \leq 544$  (not just multiples of 4), are shown in Figures 12(a) and 12(b), which exhibit the sawtooth pattern in greater detail in the measurements for both algorithms.

It is difficult to measure GPU memory bandwidth and computational costs directly, because CUDA “abstracts” the hardware architecture, and the optimizing compiler does extraordinarily complex instruction scheduling, so we relied entirely on measuring wall-clock time for entire application executions. Nevertheless, we can draw some inferences.

Comparing Figures 11(a) and 11(b), it may seem at first glance that the “butterfly partial sums” technique provides a substantial improvement over the “register transpose” technique in the case of 64-bit data, but hardly any improvement in the case of 32-bit data. Why? Closer inspection reveals that actually the “register transpose” technique is substantially *worse* than might be expected

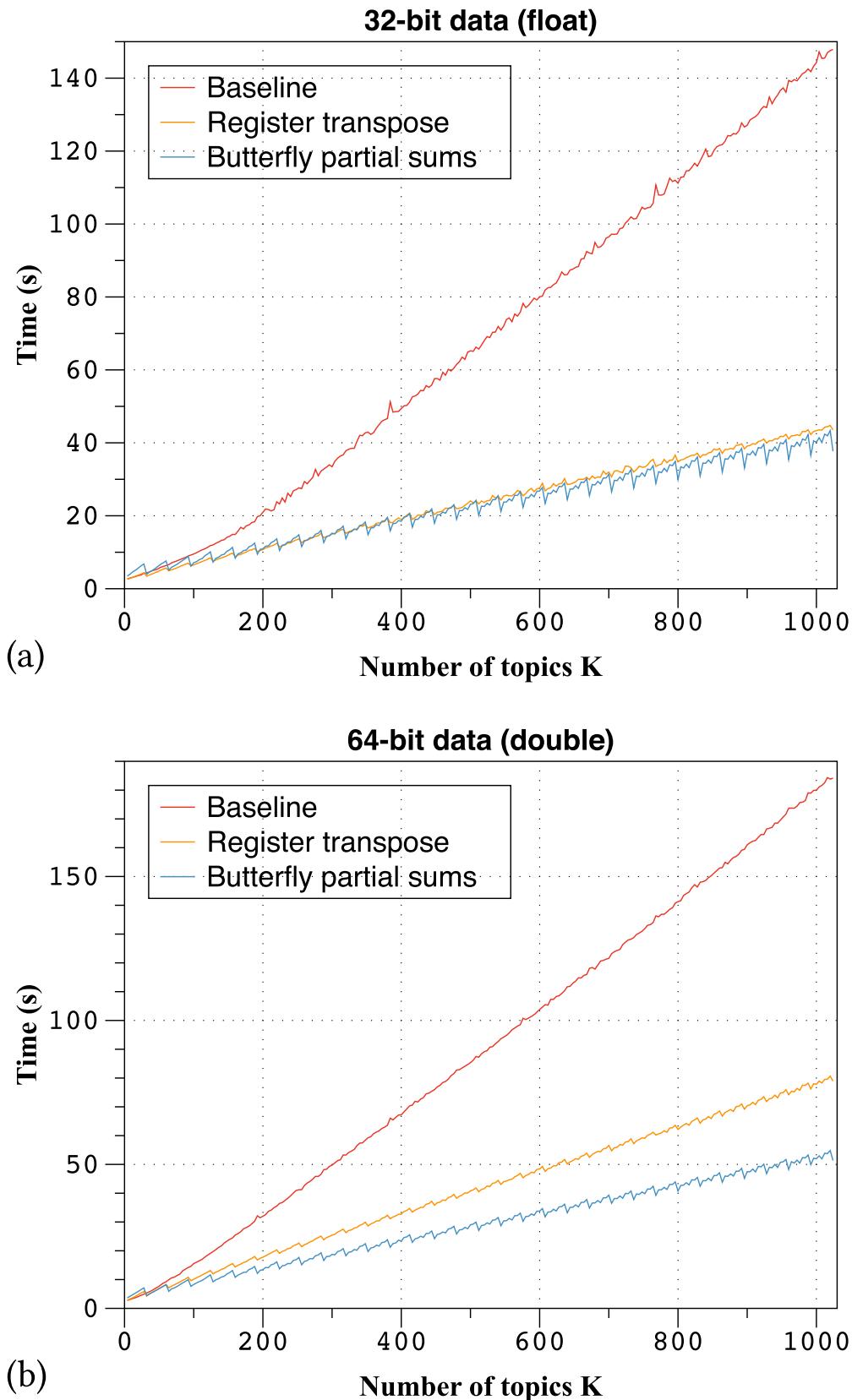


Fig. 11. Two sets of measurements of execution time for a complete machine learning application (topic modeling by Gibbs sampling; CUDA 7.5;  $K = 4k + 4$ ,  $0 \leq k \leq 255$ ).

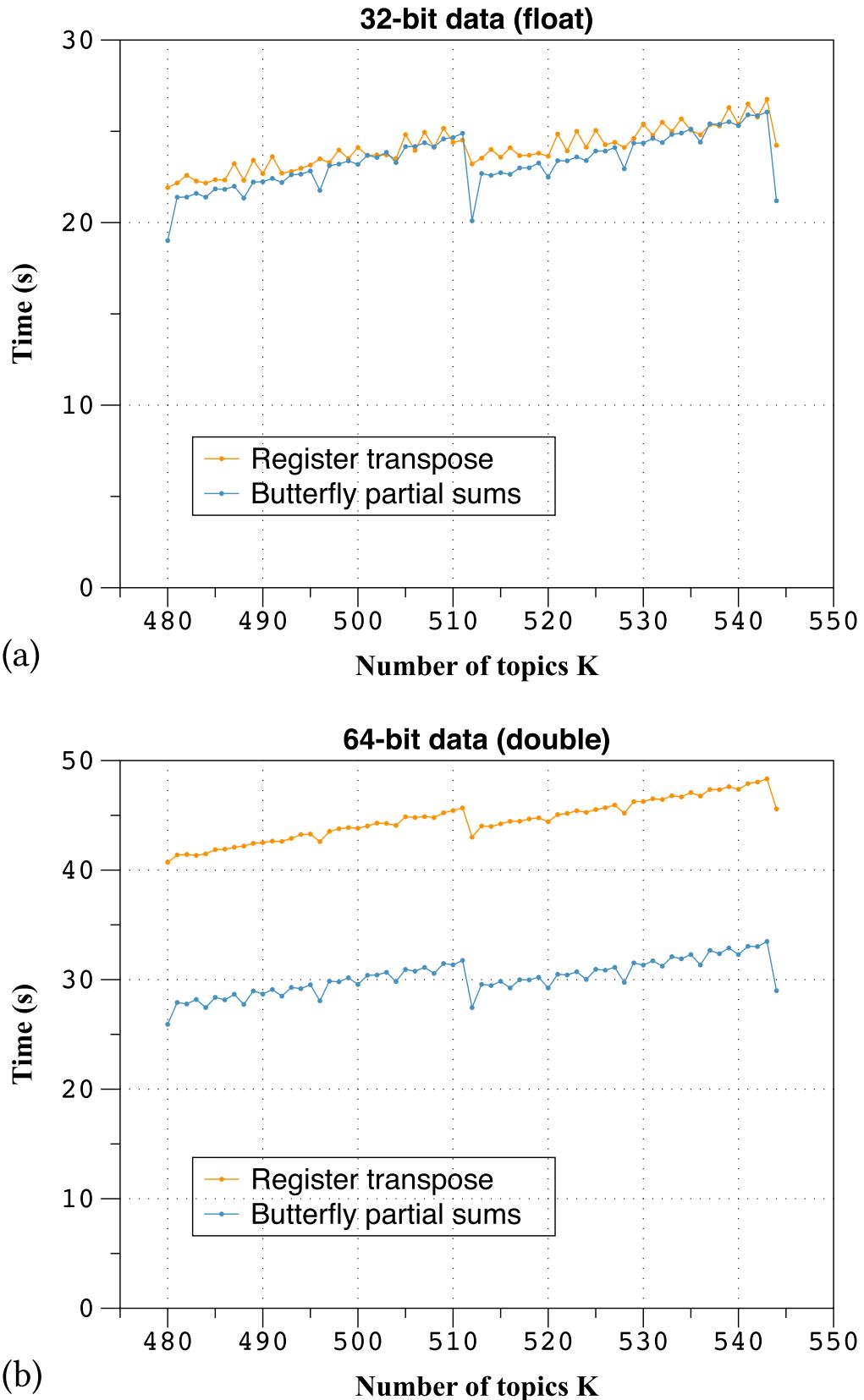


Fig. 12. Two additional (zoomed-in) sets of measurements of execution time for a complete machine learning application (topic modeling by Gibbs sampling;  $480 \leq K \leq 544$ ).

in the case 64-bit data. In each of Figures 11(a) and 11(b), the “butterfly partial sums” runs (Algorithm 7) show an improvement over the baseline (Algorithm 1) of somewhat over 70% (74% for 32-bit data, 71% for 64-bit data, at  $K = 1024$ ), so this improvement is consistent. However, for the “register transpose” runs (Algorithm 4) we see an improvement of just about 70% in the case of 32-bit data but of only about 56% in the case of 64-bit data, and this is a discrepancy that warrants explanation. We believe that the problem lies with the unfavorable transposed accesses that occur in line 20 of Algorithm 6. For the most part these accesses cannot be effectively coalesced. Now, computational instructions are overlapped with memory accesses; we conjecture that in the 32-bit cases these memory accesses take only slightly longer than the computational work, but the 64-bit case likely touches about twice as many lines of memory as the 32-bit case, and these additional memory accesses take much longer than the computational work.

## 8 RELATED WORK

Because the computed probabilities are relative in our LDA application, it is necessary to compute all of them and then to compute, if nothing else, their sum, so that the relative probabilities can be effectively normalized. Therefore every method for drawing from a discrete distribution represented by a set of relative probabilities involves some amount of preprocessing before drawing from the distribution. The various algorithms in the literature have differing tradeoffs according to what technique is used for preprocessing and what technique is used for drawing; some algorithms also accommodate incremental updating of the relative probabilities by providing a technique for incremental preprocessing.

Instead of doing a binary search on the partial sums, one can instead (as Marsaglia [18] observes in passing) construct a search tree using the principles of Huffman encoding [9] (independently rediscovered by Zimmerman [36]) to minimize the expected number of comparisons. In either case, the complexity of the search is  $O(\log n)$ , but the optimized search may have a smaller constant, obtained at the expense of a preprocessing step that must sort the relative probabilities and therefore has complexity  $\Omega(n \log n)$ .

Walker [30, 31] describes what we now call the “alias” method, in which  $n$  relative probabilities are preprocessed into two additional tables  $F$  and  $A$  of length  $n$ . To draw a value from the distribution, let  $k$  be a integer chosen uniformly at random from  $\{0, 1, 2, \dots, n - 1\}$  and let  $u$  be chosen uniformly at random from the real interval  $[0, 1]$ . Then the value drawn is (*if*  $u < F_k$  *then*  $k$  *else*  $A_k$ ). Therefore, once the tables  $F$  and  $A$  have been produced, the complexity of drawing a value from the distribution is  $O(1)$ , assuming that the cost of an array access is  $O(1)$ . Walker’s method [31] for producing the tables  $F$  and  $A$  requires time  $\Theta(n^2)$ ; it is easy to reduce this to  $\Omega(n \log n)$  by sorting the probabilities [12, exercise 3.4.1-7] and then using, say, priority heaps instead of a list for the intermediate data structure. Either version heuristically attempts to minimize the probability of having to access the table  $A$ .

Vose [29] describes a preprocessing algorithm, with proof, that further reduces the preprocessing complexity of the alias method to  $\Theta(n)$ . The tradeoff that permits this improvement is that the preprocessing algorithm makes no attempt to minimize the probability of accessing the array  $A$ .

Matias et al. [19] describe a technique for preprocessing a set of relative probabilities into a set of trees, after which a sequence of intermixed generate (draw) and update operations can be performed, where an update operation changes just one of the relative probabilities; a single generate operation takes  $O(\log^* n)$  expected time, and a single update operation takes  $O(\log^* n)$  amortized expected time.

Li et al. [15] describe a modified LDA topic modeling algorithm, which they call Metropolis-Hastings-Walker sampling, that uses Walker’s alias method but amortizes the cost of constructing the table by drawing from the same table during multiple consecutive sampling iterations of a

Metropolis-Hastings sampler; their paper provides some justification for why it is acceptable to use a “slightly stale” alias table (their words) for the purposes of this application.

The trees embedded in the butterfly-patterned partial-sums table are reminiscent of Fenwick’s binary-indexed trees [6], in that tree nodes containing partial sums are stored as array elements whose addresses are calculated through bit-manipulation of indices. However, the butterfly-patterned table as formulated here differs in three ways: (a) it stores partial sums for multiple distributions in a two-dimensional format rather than partial sums for a single distribution in a one-dimensional format; (b) it requires maintenance of two running values rather than one as a search descends the tree; and (c) at each step of the search it will always perform either an addition to one of the running values or subtraction from the other running value, whereas the Fenwick search always uses subtraction on its single running value, but at each step the subtraction is conditional.

There is a growing literature on interesting and clever techniques for improving the speed of parallel-prefix computations, especially on GPU architectures [5, 17, 34], and these techniques could possibly also be profitably applied to the problem of sampling from discrete distributions. However, we emphasize that, in contrast, the entire point of the butterfly-patterned partial-sums algorithm presented here is not to compute a complete prefix-sum table *faster*, but to *avoid* computing most of it in the first place.

## 9 CONCLUSIONS

This article focuses on one low-level “utility algorithm”: independent sampling from a large number of discrete distributions. The technique presented here can be compared to an optimization that applies “only” to sorting. But sorting is an operation of broad utility that can be exploited in a wide variety of applications. In the same way, drawing from multiple discrete distributions is the most important component of a wide class of machine learning algorithms: discrete latent variable models. This class encompasses mixture models (such as Gaussian mixture models), mixed membership models (such as topic models), mixtures of experts (such as probabilistic decision trees), learning meta-algorithms (such as Bayesian averaging), and more. Just in the specific case of LDA, we are currently aware of more than 50 variants. The currently most scalable and statistically efficient inference training procedures for LDA are based on the Stochastic Expectation Maximization variant of the Gibbs sampling procedure; they all use independent sampling as their most costly step, so improving the speed of independent sampling greatly improves the overall speed of these training algorithms. From an academic perspective, much recent work on Bayesian non-parametrics and hierarchical modeling builds upon LDA (for example, the Pachinko Allocation model and the Chinese Restaurant Franchise model). Speeding up LDA is a fundamental first step toward making such more advanced models practical. From an industrial perspective, LDA and its extensions are now making their way into useful tools and services, for example in product recommendation systems [8], consumer personalization systems [1], audience expansion systems for online advertisement [11], and document summarization [21]. It is precisely as this technology is being deployed that engineering for speed, such as we do in this article, is most useful. Independent sampling is also used in chemistry and physics, for example to compute the ground state of Ising models (and Potts models) and to simulate Stochastic Cellular Automata.

The technique of constructing butterfly-patterned partial sums appears to be best suited for situations where a SIMD processor is used to compute tables of relative probabilities for multiple discrete distributions, each of which is then used just once to draw a single value, and where each thread, when computing its table, must fetch data from a contiguous region of memory whose address is computed from other data. The LDA application for which we developed the technique has these characteristics. The technique uses transposed memory access to allow a SIMD memory

controller to touch at most three cache lines on each fetch, then cheaply constructs a butterfly-patterned set of partial sums that are just adequate to allow partial sums actually needed to be constructed on the fly during the course of a binary search. This butterfly-pattern approach provides significant speedup (up to 35%) over a transposition-only approach for our LDA machine learning application.

## REFERENCES

- [1] Amr Ahmed, Linagjie Hong, and Alexander J. Smola. 2015. Nested Chinese restaurant franchise processes: Applications to user tracking and document modeling. In *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*. Microtome Publishing, Brookline, MA, 1426–1434. Retrieved from <http://www.jmlr.org/proceedings/papers/v28/ahmed13.pdf>.
- [2] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, New York.
- [3] David M. Blei. 2012. Probabilistic topic models. *Commun. ACM* 55, 4 (Apr. 2012), 77–84. DOI : <https://doi.org/10.1145/2133806.2133826>
- [4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet allocation. *J. Mach. Learn. Res.* 3 (Mar. 2003), 993–1022. Retrieved from <http://dl.acm.org/citation.cfm?id=944919.944937>.
- [5] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. 2008. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS '08)*. ACM, New York, 205–213. DOI : <https://doi.org/10.1145/1375527.1375559>
- [6] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software Pract. Exper.* 24, 3 (1994), 327–336. DOI : <https://doi.org/10.1002/spe.4380240306>
- [7] Thomas L. Griffiths and Mark Steyvers. 2004. Finding scientific topics. *Proc. Natl. Acad. Sci. U.S.A.* 101, suppl. 1 (2004), 5228–5235. DOI : <https://doi.org/10.1073/pnas.0307752101>
- [8] Diane Hu, Rob Hall, and Josh Attenberg. 2014. Style in the long tail: Discovering unique interests with latent variable models in large scale social E-commerce. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, New York, 1640–1649. DOI : <https://doi.org/10.1145/2623330.2623338>
- [9] D. A. Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proc. IRE* 40, 9 (Sept. 1952), 1098–1101. DOI : <https://doi.org/10.1109/JRPROC.1952.273898>
- [10] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. 1989. Matrix multiplication on the connection machine. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, 326–332. <http://doi.acm.org/10.1145/76263.76298>
- [11] Joon Hee Kim, Amin Mantrach, Alejandro Jaimes, and Alice Oh. 2016. How to compete online for news audience: Modeling words that attract clicks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, New York, 1645–1654. DOI : <https://doi.org/10.1145/2939672.2939873>
- [12] Donald E. Knuth. 1998. *Seminumerical Algorithms* (3rd edition). The Art of Computer Programming, Vol. 2. Addison-Wesley, Reading, MA.
- [13] Donald E. Knuth. 1998. *Sorting and Searching* (2nd edition). The Art of Computer Programming, Vol. 3. Addison-Wesley, Reading, MA.
- [14] Anthony Lee, Christopher Yau, Michael B. Giles, Arnaud Doucet, and Christopher C. Holmes. 2010. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *J. Comput. Graph. Stat.* 19, 4 (2010), 769–789. <http://arxiv.org/pdf/0905.2441.pdf>
- [15] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. 2014. Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'14)*. ACM, New York, 891–900. DOI : <https://doi.org/10.1145/2623330.2623756>
- [16] Mian Lu, Ge Bai, Qiong Luo, Jie Tang, and Jiuxin Zhao. 2013. Accelerating topic model training on a single machine. In *Web Technologies and Applications (APWeb 2013)*, Yoshiharu Ishikawa, Jianzhong Li, Wei Wang, Rui Zhang, and Wenjie Zhang (Eds.). Lecture Notes in Computer Science, Vol. 7808. Springer, Berlin, 184–195. DOI : [https://doi.org/10.1007/978-3-642-37401-2\\_20](https://doi.org/10.1007/978-3-642-37401-2_20)
- [17] Sepideh Maleki, Annie Yang, and Martin Burtscher. 2016. Higher-order and tuple-based massively-parallel prefix sums. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*. ACM, New York, 539–552. DOI : <https://doi.org/10.1145/2908080.2908089>
- [18] G. Marsaglia. 1963. Generating discrete random variables in a computer. *Commun. ACM* 6, 1 (Jan. 1963), 37–38. DOI : <https://doi.org/10.1145/366193.366228>
- [19] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 1993. Dynamic generation of discrete random variates. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 361–370. Retrieved from <http://dl.acm.org/citation.cfm?id=313559.313807>.

- [20] NVIDIA. 2015. Developer Zone website: CUDA Toolkit documentation: CUDA Toolkit v6.5 Programming Guide, section B.14. Warp shuffle functions. Retrieved from <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions>.
- [21] Daniel Ramage, Susan Dumais, and Dan Liebling. 2010. Characterizing microblogs with topic models. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*. Association for the Advancement of Artificial Intelligence, Palo Alto, CA, 130–137.
- [22] Guy L. Steele Jr. 2016. Using Butterfly-Patterned Partial Sums to Draw from Discrete Distributions. *GTC website*. Retrieved from <http://on-demand.gputechconf.com/gtc/2016/video/s6665-guy-steele-fast-splittable.mp4>.
- [23] Guy L. Steele Jr. 2016. Using butterfly-patterned partial sums to draw from discrete distributions. In *NVIDIA GPU Technology Conference*. Retrieved from <http://on-demand.gputechconf.com/gtc/2016/presentation/s6666-guy-steele-butterfly-pattern.pdf>. Slides for talk S6665. Video available at Reference [22].
- [24] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2015. Using butterfly-patterned partial sums to optimize GPU memory accesses for drawing from discrete distributions. *CoRR (Computing Research Repository at arXiv.org)* (May 2015). Retrieved from <http://arxiv.org/abs/1505.03851>.
- [25] Guy L. Steele Jr. and Jean-Baptiste Tristan. 2017. Using butterfly-patterned partial sums to draw from discrete distributions. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. ACM, New York, 341–355. DOI : <https://doi.org/10.1145/3018743.3018757> An early version of this paper is Reference [24].
- [26] Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. 2010. Understanding GPU programming for statistical computation: Studies in massively parallel massive mixtures. *J. Comput. Graphic. Stat.* 19, 2 (2010), 419–438.
- [27] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele Jr. 2014. Augur: Data-parallel probabilistic modeling. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.). Curran Associates, 2600–2608. Retrieved from <http://papers.nips.cc/book/year-2014>.
- [28] Jean-Baptiste Tristan, Joseph Tassarotti, and Guy L. Steele Jr. 2015. Efficient training of LDA on a GPU by mean-for-mode estimation. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. Microtome Publishing, Brookline, MA, 59–68. Retrieved from <http://jmlr.org/proceedings/papers/v37/tristan15.pdf>.
- [29] M. D. Vose. 1991. A linear algorithm for generating random numbers with a given distribution. *IEEE Trans. Software Engineer.* 17, 9 (Sept. 1991), 972–975. DOI : <https://doi.org/10.1109/32.92917>
- [30] A. J. Walker. 1974. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electron. Lett.* 10, 8 (Apr. 1974), 127–128. DOI : <https://doi.org/10.1049/el:19740097>
- [31] Alastair J. Walker. 1977. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Software* 3, 3 (Sept. 1977), 253–256. DOI : <https://doi.org/10.1145/355744.355749>
- [32] Nicholas Wilt. 2013. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, Upper Saddle River, NJ.
- [33] Feng Yan, Ningyi Xu, and Yuan Qi. 2009. Parallel inference for latent Dirichlet allocation on graphics processing units. In *Advances in Neural Information Processing Systems 22*. Curran Associates, 2134–2142. Retrieved from <http://papers.nips.cc/book/year-2009>.
- [34] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, 229–238. DOI : <https://doi.org/10.1145/2442516.2442539>
- [35] Huasha Zhao, Biye Jiang, and John Canny. 2014. SAME but different: Fast and high-quality Gibbs parameter estimation. *CoRR (Computing Research Repository at arXiv.org)* (Sept. 2014). Retrieved from <http://arxiv.org/abs/1409.5402>.
- [36] Seth Zimmerman. 1959. An optimal search procedure. *Amer. Math. Monthly* 66, 8 (Oct. 1959), 690–693.

Received August 2018; revised March 2019; accepted May 2019

---

# Sketching for Latent Dirichlet-Categorical Models

---

Joseph Tassarotti  
MIT CSAIL

Jean-Baptiste Tristan  
Oracle Labs

Michael Wick  
Oracle Labs

## Abstract

Recent work has explored transforming data sets into smaller, approximate summaries in order to scale Bayesian inference. We examine a related problem in which the parameters of a Bayesian model are very large and expensive to store in memory, and propose more compact representations of parameter values that can be used during inference. We focus on a class of graphical models that we refer to as latent Dirichlet-Categorical models, and show how a combination of two sketching algorithms known as count-min sketch and approximate counters provide an efficient representation for them. We show that this sketch combination – which, despite having been used before in NLP applications, has not been previously analyzed – enjoys desirable properties. We prove that for this class of models, when the sketches are used during Markov Chain Monte Carlo inference, the equilibrium of sketched MCMC converges to that of the exact chain as sketch parameters are tuned to reduce the error rate.

## 1 Introduction

The development of *scalable Bayesian inference* techniques (Angelino et al., 2016) has been the subject of much recent work. A number of these techniques introduce some degree of approximation into inference.

This approximation may arise by altering the inference algorithm. For example, in “noisy” Metropolis Hastings algorithms, acceptance ratios are perturbed because the likelihood function is either simplified or evaluated on a random subset of data in each iteration (Negrea and Rosenthal, 2017; Alquier et al., 2014; Pillai and Smith,

---

Proceedings of the 22<sup>nd</sup> International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89. Copyright 2019 by the author(s).

2014; Bardenet et al., 2014). Similarly, asynchronous Gibbs sampling (Sa et al., 2016) violates some strict sequential dependencies in normal Gibbs sampling in order to avoid synchronization costs in the distributed or concurrent setting.

Other approaches transform the original large data set into a smaller representation on which traditional inference algorithms can then be efficiently run. Huggins et al. (2016) compute a weighted subset of the original data, called a *coreset*. Geppert et al. (2017) consider Bayesian regression with  $n$  data points each of dimension  $d$ , and apply a random projection to shrink the original  $\mathbb{R}^{n \times d}$  data set down to  $\mathbb{R}^{k \times d}$  for  $k < n$ . An advantage of these kinds of transformations is that by shrinking the size of the data, it becomes more feasible to fit the transformed data set entirely in memory.

The transformations described in the previous paragraph reduce the number of data points under consideration, but preserve the *dimension* of each data point, and thus the number of parameters in the model. However, in many Bayesian mixed membership models, the number of parameters themselves can also become extremely large when working with large data sets, and storing these parameters poses a barrier to scalability.

In this paper, we consider an approximation to address this issue for what we call latent Dirichlet-Categorical models, in which there are many latent categorical variables whose distributions are sampled from Dirichlets. This is a fairly general pattern that can be found as a basic building block of many Bayesian models used in NLP (e.g., clustering of discrete data, topic models like LDA, hidden Markov models). The most representative example, which we will use throughout this paper, is the following:

$$z_i \sim \text{Categorical}(\tau) \quad i \in [N] \quad (1)$$

$$\theta_i \sim \text{Dirichlet}(\alpha) \quad i \in [K] \quad (2)$$

$$x_i \sim \text{Categorical}(\theta_{z_i}) \quad i \in [N] \quad (3)$$

Here,  $\alpha$  is a scalar value and  $\tau$  is some fixed hyperparameter of dimension  $K$ . We assume that the dimension of the Dirichlet distribution is  $V$ , a value we refer to as the “vocabulary size”. Each random variable  $x_i$

can take one of  $V$  different values, which we refer to as “data types” (e.g., words in latent Dirichlet allocation). Associated with each  $x_i$  is a latent variable  $z_i$  which represents an assignment of  $x_i$  to one of  $K$  possible topics or categories.

To do Gibbs sampling for a model in which such a pattern occurs, we generally need to compute a certain matrix  $c$  of dimension  $K \times V$ . Each row of this matrix tracks the frequency of occurrence of some data type within one of the components of the model. In general, this matrix can be quite large, often with  $V \gg K$ , and in some cases we may not even know the exact value of  $V$  a priori (e.g., consider the streaming setting where we may encounter new words during inference), making it costly to store these counts. Moreover, if we do distributed inference by dividing the data into subsets, each compute node may need to store this entire large matrix, which reduces the amount of data each node can store in memory and adds communication overhead. Using a sparse or dynamic representation instead of a fixed array makes updates and queries slower, and adds further overhead when merging distributed representations. Also,  $c$  is often *nearly* sparse, but not literally so, in the sense that many entries have a very small but non-zero count, further limiting the effectiveness of sparse representations.

We propose to address these problems by using *sketch* algorithms to store compressed representations of these matrices. These algorithms give approximate answers to certain queries about data streams while using far less space than algorithms that give exact answers. For example, the *count-min sketch* (CM) (Cormode and Muthukrishnan, 2005) can be used to estimate the frequency of items in a data set without having to use space proportional to the number of distinct items, and *approximate counters* (Morris, 1978; Flajolet, 1985) can store very large counts with sublogarithmic number of bits. These algorithms have parameters that can be tuned to trade between estimation error and space usage. Because many natural language processing tasks involve computing estimates of say, the frequency of a word in a corpus, there has been obvious prior interest in using these sketching algorithms for (non-Bayesian) NLP when dealing with very large data sets (Durme and Lall, 2009b; Goyal and Daumé III, 2011; Durme and Lall, 2009a).

We propose representing the matrix  $c$  above using a combination of count-min sketch and approximate counters. It is not clear a priori what effect this would have on the MCMC algorithm. On the one hand, it is plausible that if the sketch parameters are set so that estimation error is small enough, MCMC will still converge to some equilibrium distribution that is close to the equilibrium distribution of the exact non-sketched

version. On the other hand, we might be concerned that even small estimation errors within each iteration of the sampler would compound, causing the equilibrium distribution to be very far from that of the non-sketched algorithm.

In this paper, we resolve these issues both theoretically and empirically. We consider sequences of runs of the sketched MCMC algorithm in which parameters of the sketch are tuned to decrease the error rate between runs. We prove, under fairly general conditions, that the sequence of equilibrium distributions of the sketched runs converges to that of the non-sketched version. This ensures that a user can trade off computational cost for increased accuracy as necessary. Then, we experimentally show that when the combined sketch is used with a highly scalable MCMC algorithm for LDA, we can obtain model quality comparable to that of the non-sketched version while using much less space.

## Contribution

1. We explain how the count-min sketch algorithm and approximate counters can be used to sketch the sufficient statistics of models that contain latent Dirichlet-Categorical subgraphs (section §2). We then provide an analysis of a combined count-min sketch/approximate counter data structure which provides the benefits of both (section §3).
2. We then prove that when the combined sketch is used in an MCMC algorithm, as the parameters of the sketch are tuned to reduce error rates, the equilibrium distributions of sketched chains converge to that of the non-sketched version (section §4).
3. We complement these theoretical results with experimental evidence confirming that learning works despite approximations introduced by the sketches (section §5).

## 2 Sketching for Latent Dirichlet-Categorical Models

As described in the introduction, MCMC algorithms for models involving Dirichlet-Categorical distributions usually require tabulating statistics about the current assignments of items to categories (e.g., the words per topic in LDA). There are two reasons why maintaining this matrix of counts can be expensive. First, the dimensions of the matrix can be large – the dimensions are often proportional to the number of unique words in the corpus. Second, the values in the matrix can also be large, so that tracking them using small sized integers can potentially lead to overflow.

Sketching algorithms can be used to address these problems, providing compact fixed-size representations of these counts that use far less memory than a dense array. We start by explaining two widely used sketches, and then in the next section discuss how they can be combined.

### 2.1 Sketch 1: count-min sketch

To deal with the fact that the matrix of counts is of large dimension, we can use count-min (CM) sketches (Cor-mode and Muthukrishnan, 2005) instead of dense arrays. A CM sketch  $\mathcal{C}$  of dimension  $l \times w$  is represented as an  $l \times w$  matrix of integers, initialized at 0, and supports two operations: `update` and `query`. The CM sketch makes use of  $l$  different 2-universal hash functions of range  $w$  that we denote by  $h_1, \dots, h_l$ . The `update`( $x$ ) operation adjusts the CM sketch to reflect an increment to the frequency of some value  $x$ , and is done by incrementing the matrix at locations  $\mathcal{C}_{i,h_i(x)}$  for  $i \in [1, l]$ . The `query`( $x$ ) operation<sup>1</sup> returns an estimate of the frequency of value  $x$  and is computed by  $\min_i \mathcal{C}_{i,h_i(x)}$ .

It is useful to think of a value  $C_{a,b}$  in the matrix as a random variable. In general, when we study an arbitrary value, say  $x$ , we need not worry about where it is located in row  $i$  and refer to  $\mathcal{C}_{i,h_i(x)}$  simply as  $Z_i$ , and write  $Q(x) := \min_i(Z_i)$  for the result when querying  $x$ . Note that  $Z_i$  equals the true number of occurrences of  $x$ , written  $f_x$ , plus the counts of other keys whose hashes are identical to that of  $x$ . CM sketches have several interesting properties, some of which we summarize here (see Roughgarden and Valiant (2015) for a good expository account). Let  $N$  be the total number of increments to the CM sketch. Then, each  $Z_i$  is a biased estimator, in that:

$$\mathbb{E}[Z_i] = f_x + \frac{N - f_x}{w} \quad (4)$$

However, by adjusting the parameters  $l$  and  $w$ , we can bound the probability of large overestimation. In particular, by taking  $w = \frac{k}{\epsilon}$  one can bound the offset of a query as

$$\Pr [Q(x) \geq f_x + \epsilon N] \leq \frac{1}{k^l} \quad (5)$$

A nice property of CM sketches is that they can be used in parallel: we can split a data stream up, derive a sketch for each piece, and then merge the sketches

<sup>1</sup> Other query rules can be used, such as the count-mean-min (Deng and Rafiei, 2007) rule. However, Goyal et al. (2012) suggest that conventional CM sketch has better average error for queries of mid to high frequency keys in NLP tasks. Therefore, we will focus on the standard CM estimator.

together simply by adding the entries in the different sketches together componentwise.

We want to replace the *matrix* of counts  $c$  in a Dirichlet-Categorical model with sketches. There is some flexibility in how this is done. The simplest thing is to replace the entire matrix with a single sketch (so that the keys are the indices into the matrix). Alternatively, we can divide the matrix into sub-matrices, and use a sketch for each sub-matrix. In the setting of Dirichlet-Categorical models, each row of  $c$  corresponds to the counts for data types within one component of the model (*e.g.*, counts of words for a given topic in LDA), so it is natural to use a sketch per row.

### 2.2 Sketch 2: approximate counting

In order to represent large counts without the memory costs of using a large number of bytes, we can employ approximate counters (Morris, 1978). An approximate counter  $X$  of base  $b$  is represented by an integer (potentially only a few bits) initialized at 0, and supports two operations: increment and read. We write  $X_n$  to denote a counter that has been incremented  $n$  times. The increment operation is randomized and defined as:

$$\Pr(X_{n+1} = k + 1 \mid X_n = k) = b^{-k} \quad (6)$$

$$\Pr(X_{n+1} = k \mid X_n = k) = 1 - b^{-k} \quad (7)$$

Reading a counter  $X$  is written as  $\phi(X)$  and defined as  $\phi(X) = (b^X - 1)/(b - 1)$ . Approximate counters are unbiased, and their variance can be controlled by adjusting  $b$ :

$$\mathbb{E}[\phi(X_n)] = n \quad \mathbb{V}[\phi(X_n)] = \frac{b-1}{2}(n^2 - n) \quad (8)$$

Using approximate counters as part of inference for Dirichlet-Categorical models is very simple: instead of representing the matrix  $c$  as an array of integers, we instead use an array of approximate counters.

## 3 Combined Sketching: Alternatives and Analysis

The problems addressed by the sketches described in the previous section are complementary: CM sketches replace a large matrix with a much smaller set of arrays; but by coalescing increments for distinct items, CM sketches need to potentially store larger counts to avoid overflows, a problem which is resolved with approximate counting. Therefore, it is natural to consider how to combine the two sketching algorithms together.

### 3.1 Combination 1: Independent Counters

The simplest way to combine the CM sketch with approximate counters is to replace each exact counter

in the CM sketch with an approximate counter; then when incrementing a key in the sketch, we independently increment each of the counters it corresponds to. Moreover, because there are ways to efficiently add together two approximate counters (Steele and Tristan, 2016), we can similarly merge together multiple copies of these sketches by once again adding their entries together componentwise.

When we combine the CM sketch and the approximate counters together in this way, the errors introduced by these two kinds of algorithms interact. It is challenging to give a precise analysis of the error rate of the combined structure. However, it is still the case that we can tweak the parameters of the sketch to make the error rate arbitrarily low.

To make this precise, note that we now have three parameters to tune:  $b$ , the base of the approximate counters,  $l$  the number of hashes, and  $w$ , the range of the hashes. Given a parameter triple  $\psi = (b, l, w)$ , write  $Q_\psi(x)$  for the estimate of key  $x$  from a sketch using these parameters. Then, given a sequence  $\psi_n = (b_n, l_n, w_n)$  of parameters, we can ask what happens to the sequence of estimates  $Q_{\psi_n}(x)$  when we use the sketches on the same fixed data set:

**Theorem 3.1.** *Let  $\psi_n = (b_n, l_n, w_n)$ . Suppose  $b_n \rightarrow 1$ ,  $w_n \rightarrow \infty$  and there exists some  $L$  such that  $1 \leq l_n \leq L$  for all  $n$ . Then for all  $x$ ,  $Q_{\psi_n}(x)$  converges in probability to  $f_x$  as  $n \rightarrow \infty$ .*

See Appendix A in the supplementary material for the full proof. This result shows that for appropriate sequences  $\psi_n$  of parameters, the estimator  $Q_{\psi_n}(x)$  is consistent. We call a sequence  $\psi_n$  satisfying the conditions of Theorem 3.1 a *consistent sequence* of parameters.

For our application, we are replacing a matrix of counts with a collection of sketches for each row, so we want to know not just about the behavior of the estimate of a single key in one of these sketches, but about the estimates for all keys across all sketches. Formally, let  $c$  be a  $K \times V$  dimensional matrix of counts. Consider a collection of  $K$  sketches, each with parameters  $\psi$ , where for each key  $v$ , we insert  $v$  a total of  $c_{k,v}$  times into the  $k$ th sketch. then we write  $Q_\psi(c)$  for the random  $K \times V$  matrix giving the estimates of all the keys in each sketch. Because convergence in probability of a random vector follows from convergence of each of the components, the above implies:

**Theorem 3.2.** *If  $\psi_n$  is a consistent sequence, then  $Q_{\psi_n}(c)$  converges in probability to  $c$ .*

Finally, we have been describing the situation where the keys are inserted with some deterministic frequency and the only source of randomness is in the hashing

of the keys and the increments of the approximate counter. However, it is natural to consider the case where the frequency of the keys is randomized as well. To do so, we define the Markov kernel<sup>2</sup>  $T_\psi$  from  $\mathbb{N}^{K \times V}$  to  $\mathbb{R}_{\geq 0}^{K \times V}$ , where for each  $c$ ,  $T_\psi(c, \cdot)$  is the distribution of the random variable  $Q_\psi(c)$  considered above. Then if  $\mu$  is a distribution on count matrices,  $\mu T_\psi$  gives the distribution of query estimates returned for the sketched matrix.

### 3.2 Combination 2: Correlated Counters

Even though the results above show that the approximation error of the combined sketch can be made arbitrarily small, it is still possible for an estimate to be smaller than the true count,  $f_x$ . This underestimation rules out using the so-called *conservative update* rule (Estan and Varghese, 2002), a technique which can be used to reduce bias of normal CM sketches. When using conservative update with a regular CM sketch, to increment a key  $x$ , instead of incrementing each of the counters corresponding to  $x$ , we first find the minimum value and then only increment counters equal to this minimum. But because approximate counters can underestimate, this is no longer justifiable in the combined sketch.

Pitel and Fouquier (2015) proposed an alternative way to combine CM sketches with approximate counters that enables conservative updates. We call their combination *correlated counters*. Figure 1 shows the increment routine with and without conservative update for correlated counters. The idea in each is that we generate a single uniform  $[0, 1]$  random variable  $r$  and use this common  $r$  to decide how to transition each counter value according to the probabilities described in §2.2.

However, Pitel and Fouquier (2015) did not give a proof of any statistical properties of their combination. The following result shows that this variant avoids the underapproximation bias of the independent counter version:

**Theorem 3.3.** *Let  $Q(x)$  be the query result for key  $x$  using correlated counters in a CM sketch with one of the increment procedures from Figure 1. Then,*

$$f_x \leq \mathbb{E}[Q(x)] \leq f_x + \frac{N - f_x}{w}$$

*Proof.* We only discuss the non-conservative update increment procedure, since the proof is similar for the other case. The upper bound is straightforward. The

<sup>2</sup>Throughout, we assume that all topological spaces are endowed with their Borel  $\sigma$ -algebras, and omit writing these  $\sigma$ -algebras.

```

1: procedure INCR-CORRELATED( $C, x$ )
2:   let  $r \leftarrow \text{Uniform}(0, 1)$ 
3:   for  $i$  from 0 to  $l$  do
4:     let  $v \leftarrow C[i][h_i(x)]$ 
5:     if  $r < \frac{1}{b^v}$  then  $C[i][h_i(x)] \leftarrow v + 1$ 
6:   end for
7:
8: procedure INCR-CONSERVATIVE( $C, x$ )
9:   let  $r \leftarrow \text{Uniform}(0, 1)$ 
10:  let  $min \leftarrow \infty$ 
11:  for  $i$  from 0 to  $l$  do
12:    let  $v \leftarrow C[i][h_i(x)]$ 
13:    if  $v < min$  then  $min \leftarrow v$ 
14:  end for
15:
16:  if  $r < \frac{1}{b^{min}}$  then
17:    for  $i$  from 0 to  $l$  do
18:      if  $C[i][h_i(x)] = min$  then
19:         $C[i][h_i(x)] \leftarrow min + 1$ 
20:    end for
21:
```

Figure 1: Increment for CM sketch with correlated approximate counters, with and without conservative update.

lower bound is proved by exhibiting a coupling (Lindvall, 2002) between the sketch counters corresponding to key  $x$  and a counter  $C$  of base  $b$  that will be incremented exactly  $f_x$  times. The coupling is constructed by induction on  $N$ , the total number of increments to the sketch. Throughout, we maintain the invariant that  $\phi(C) \leq Q(x)$ ; it follows that  $\mathbb{E}[\phi(C)] \leq \mathbb{E}[Q(x)]$ . Since  $\mathbb{E}[\phi(C)] = f_x$ , this will give the desired bound.

In the base case, when  $N = 0$ , both  $\phi(C)$  and  $Q(x)$  are 0 so the invariant holds trivially. Suppose the invariant holds after the first  $k$  increments to the sketch, and some key  $y$  is then incremented. If  $x = y$ , then we transition the counter  $C$  using the same random uniform variable  $r$  that is used to transition the counters  $X_1, \dots, X_l$  corresponding to key  $x$  in the sketch. There are two cases: either  $r$  is small enough to cause the minimum  $X_i$  to increase by 1, or not. If it is, then since  $C \leq \min_i(X_i)$ ,  $r$  is also small enough to cause  $C$  to increase by 1, and so  $\phi(C) \leq Q(x)$ . If  $\min_i(X_i)$  does not change, but  $C$  does, then we must have  $C < \min_i(X_i)$  before the transition; since  $C$  can only increase by 1, we still have  $C \leq \min_i(X_i)$  afterward.

If the key  $y$  is not equal to  $x$ , then we leave  $C$  as is. Since each  $X_i$  can only possibly increase while  $C$  stays the same, the invariant holds. Finally, after all  $N$  increments have been performed,  $C$  will have received  $f_x$  increments, so that  $\mathbb{E}[\phi(C)] = f_x$  because

approximate counters are unbiased.  $\square$

In Appendix D we describe various microbenchmarks comparing the behavior of the different ways of combining the two sketches.

## 4 Asymptotic Convergence

In the previous section, we explored some of the statistical properties of the combined sketch. We now turn to the question of the behavior of an MCMC algorithm when we use these sketches in place of exact counts. More precisely, suppose we have a Markov chain whose states are tuples of the form  $(c, z)$ , where  $c$  is a  $K \times V$  matrix of counts, and  $z$  is an element of some complete separable metric space  $Y$ . Now, suppose instead of tabulating  $c$  in a dense array of exact counters, we replace each row with a sketch using parameters  $\psi$ . We can ask whether the resulting sketched chain<sup>3</sup> has an equilibrium distribution, and if so, how it relates to the equilibrium distribution of the original “exact” chain. As we will see, it is often easy to show that the sketched chain still has an equilibrium distribution. However, the relationship between the sketched and exact equilibriums may be quite complicated. Still, a reasonable property to want is that, if we have a consistent sequence of parameters  $\psi_n$ , and we consider a sequence of runs of the MCMC algorithm, where the  $i$ th run uses parameters  $\psi_i$ , then the sequence of equilibrium distributions will converge to that of the exact chain.

The reason such a property is important is that it provides some justification for how these sketched approximations would be used in practice. Most likely, one would first test the algorithm using some set of sketch parameters, and then if the results are not satisfactory, the parameters could be adjusted to decrease error rates in exchange for higher computational cost. (Just as, when using standard MCMC techniques without an a priori bound on mixing times, one can run chains for longer periods of time if various diagnostics suggest poor results). Therefore, we would like to know that asymptotically this approach really would converge to the behavior of the exact chain. We will now show that under reasonable conditions, this convergence does in fact hold.

We assume the state space  $S_e$  of the original chain is a compact, measurable subset of  $\mathbb{N}^{K \times V} \times Y$ . We suppose

<sup>3</sup>Since approximate counters can return floating point estimates of counts, replacing the exact counts with sketches only makes sense if the transition kernel for the Markov chain can be interpreted when these state components involve floating point numbers. But this is usually the case since Bayesian models typically apply non-integer smoothing factors to integer counts anyway.

that the transition kernel of the chain can be divided into three phrases, represented by the composition of kernels  $\kappa_{\text{pre}} \cdot \kappa \cdot \kappa_{\text{post}}$ , where in  $\kappa$  the matrix of counts is updated in a way that depends only on the rest of the state, which is then modified in  $\kappa_{\text{pre}}$  and  $\kappa_{\text{post}}$  (*e.g.*, in a blocked Gibbs sampler  $\kappa$  would correspond to the part of a sweep where  $c$  is updated). Moreover, we assume that the transitions  $\kappa_{\text{pre}}$  and  $\kappa_{\text{post}}$  are well-defined on the extended state space  $\mathbb{R}_{\geq 0}^{K \times V} \times Y$ , where the counts are replaced with positive reals. Formally, these conditions mean we assume that there exist Markov kernels  $\kappa'_{\text{pre}}, \kappa'_{\text{post}} : \mathbb{R}_{\geq 0}^{K \times V} \times Y \rightarrow Y$  and  $\kappa : Y \rightarrow \mathbb{N}^{K \times V}$  such that

$$\begin{aligned}\kappa_{\text{pre}}((c, z), A) &= \int \kappa'_{\text{pre}}((c, z), dz') 1_A(c, z') \\ \kappa((c, z), A) &= \int \kappa'(z, dc') 1_A(c', z) \\ \kappa_{\text{post}}((c, z), A) &= \int \kappa'_{\text{post}}((c, z), dz') 1_A(c, z')\end{aligned}$$

where we write  $1_A$  for the indicator function corresponding to a measurable set  $A$ . We assume this chain has a unique stationary distribution  $\mu$ . Furthermore, we assume  $\kappa_{\text{pre}}$ ,  $\kappa$ , and  $\kappa_{\text{post}}$  are *Feller continuous*, that is, if  $s_n \rightarrow s$ , then  $\kappa(s_n, \cdot) \Rightarrow \kappa(s, \cdot)$ , and similarly for  $\kappa_{\text{pre}}$  and  $\kappa_{\text{post}}$ , where  $\Rightarrow$  is weak convergence of measures.

Fix a consistent sequence of parameters  $\psi_n$ . For each  $n$ , we define the sketched Markov chain with transition kernel  $\kappa_{\text{pre}} \cdot \kappa_n \cdot \kappa_{\text{post}}$ , where  $\kappa_n$  is the kernel obtained by replacing the exact matrix of counts used in  $\kappa$  with a sketched matrix with parameters  $\psi_n$ :

$$\begin{aligned}\kappa_n((c, z), A) &= \int \kappa'(z, dc') \int T_{\psi_n}(c', dc'') 1_A(c'', z)\end{aligned}$$

(recall that  $T_{\psi_n}$  is the kernel induced by the combined sketching algorithm, as described in §3.1). We assume that the set  $S$  containing the union of the states of the exact chain and the sketched chains is some compact measurable subset of  $\mathbb{R}_{\geq 0}^{K \times V} \times Y$ . Assuming that each  $\kappa_{\text{pre}} \cdot \kappa_n \cdot \kappa_{\text{post}}$  has a stationary distribution  $\mu_n$ , we will show that they converge weakly to  $\mu$ . We use the following general result of Karr:

**Theorem 4.1** (Karr (1975, Theorems 4 and 6)). *Let  $E$  be a complete separable metric space with Borel sigma algebra  $\Sigma$ . Let  $\kappa$  and  $\kappa_1, \kappa_2, \dots$  be Markov kernels on  $(E, \Sigma)$ . Suppose  $\kappa$  has a unique stationary distribution  $\mu$  and  $\kappa_1, \dots$  have stationary distributions  $\mu_1, \dots$ . Assume the following hold*

1. *for all  $s$ ,  $\{\kappa_n(s, \cdot)\}_n$  is tight, and*
2.  *$s_n \rightarrow s$  implies  $\kappa_n(s_n, \cdot) \Rightarrow \kappa(s, \cdot)$ .*

*Then  $\mu_n \Rightarrow \mu$ .*

We now show that the assumptions of this theorem hold for our chains. The first condition is straightforward:

**Lemma 4.2.** *For all  $x$ , the family of measures  $\{(\kappa_{\text{pre}} \cdot \kappa_n \cdot \kappa_{\text{post}})(x, \cdot)\}_n$  is tight.*

*Proof.* This follows immediately from the assumption that the set of states  $S$  is a compact measurable set.  $\square$

To establish the second condition, we start with the following:

**Lemma 4.3.** *If  $s_n \rightarrow s$ , then  $(\kappa_{\text{pre}} \cdot \kappa_n)(s_n, \cdot) \Rightarrow (\kappa_{\text{pre}} \cdot \kappa)(s, \cdot)$ .*

*Proof.* To match up with the results in §3, it is helpful to rephrase this as a question of convergence of distribution of random variables with appropriate laws. By assumption  $\kappa_{\text{pre}} \cdot \kappa$  is Feller continuous, so we know that  $(\kappa_{\text{pre}} \cdot \kappa)(s_n, \cdot) \Rightarrow (\kappa_{\text{pre}} \cdot \kappa)(s, \cdot)$ , hence by Skorokhod's representation theorem, there exists random matrices  $C, C_1, \dots$ , and random  $Y$ -elements  $Z, Z_1, \dots$  such that the law of  $(C_n, Z_n)$  is  $(\kappa_{\text{pre}} \cdot \kappa)(s_n, \cdot)$ , that of  $(C, Z)$  is  $(\kappa_{\text{pre}} \cdot \kappa)(s, \cdot)$ , and  $(C_n, Z_n) \xrightarrow{P} (C, Z)$ . Then the distribution of  $(Q_{\psi_n}(C_n), Z_n)$  is that of  $(\kappa_{\text{pre}} \cdot \kappa_n)(s_n, \cdot)$ , so it suffices to show that  $Q_{\psi_n}(C_n) \xrightarrow{P} C$ .

Fix  $\delta, \epsilon > 0$ . Let  $U$  be the union of the supports of each  $C_n$ . Then  $U$  consists of integer matrices lying in some compact subset of real vectors (since  $S$  is compact and the counts returned by  $\kappa$  are exact integers), so  $U$  is finite. Moreover, by Theorem 3.2 we know that for all  $c$ , there exists  $n_c$  such that for all  $n > n_c$ ,  $\Pr[\|Q_{\psi_n}(C_n) - c\| > \epsilon/2 \mid C_n = c] < \delta/2$ . Let  $m_1$  be the maximum of the  $n_c$  for  $c \in U$ . We also know that there exists  $m_2$  such that for all  $n > m_2$ ,  $\Pr[\|C_n - C\| > \epsilon/2] < \delta/2$ . For  $n > \max(m_1, m_2)$ , we then have  $\Pr[\|Q_{\psi_n}(C_n) - C\| > \epsilon] < \delta$ .  $\square$

Continuity of  $\kappa_{\text{post}}$  then gives us:

**Lemma 4.4.** *If  $s_n \rightarrow s$ , then  $(\kappa_{\text{pre}} \cdot \kappa_n \cdot \kappa_{\text{post}})(s_n, \cdot) \Rightarrow (\kappa_{\text{pre}} \cdot \kappa \cdot \kappa_{\text{post}})(s, \cdot)$ .*

Thus by Karr's theorem we conclude:

**Theorem 4.5.**  $\mu_n \Rightarrow \mu$ .

In the above, we have assumed that there is a single sketched matrix of counts, and that each row of the matrix uses the same sketch parameters. However, the argument can be generalized to the case where there are several sketched matrices with different parameters. We now explain how this result can be applied to some Dirichlet-Categorical models:

**Example 1: SEM for LDA.** When using stochastic expectation maximization (SEM) for the LDA topic

model (Blei et al., 2003), the states of the Markov chain are matrices  $wpt$  and  $tpd$  giving the words per topic and topic per document counts. Within each round, estimates of the corresponding topic and document distributions  $\theta$  and  $\phi$  are computed from smoothed versions of these counts; new topic assignments are sampled according to this distribution, and the counts  $wpt$  and  $tpd$  are updated. We can replace the rows of either  $wpt$  or  $tpd$  with sketches. In this case  $\kappa_{\text{pre}}$  and  $\kappa_{\text{post}}$  are the identity, and the Feller continuity of  $\kappa$  follows from the fact that the estimates of  $\theta$  and  $\phi$  are continuous functions of the  $wpt$  and  $tpd$  counts. Compactness of the state space is a consequence of the fact that the set of documents (and hence maximum counts) are finite, and the maximum counter base is bounded. Finally, the sketched kernels still have unique stationary distributions because the smoothing of the  $\theta$  and  $\phi$  estimates guarantees that if a state is representable in the sketched chain, we can transition to it in a single step from any other state.

**Example 2: Gibbs for Pachinko Allocation.** The Pachinko Allocation Model (PAM) (Li and McCalum, 2006) is a generalization of LDA in which there is a hierarchy of topics with a directed acyclic structure connecting them. A blocked Gibbs sampler for this model can be implemented by first conditioning on topic distributions and sampling topic assignments for words, then conditioning on these topic assignments to update topic distributions – in the latter phase, one needs counts of the occurrences of words in the different topics and subtopics which can be collected using sketches. Since the priors for sampling topics based on these counts are smoothed, the sketched chains once again have unique stationary distributions for the same reason as in LDA.

## 5 Experimental Evaluation

We now examine the empirical performance of using these sketches. We implemented a sketched version of SCA (Zaheer et al., 2016), an optimized form of SEM which is used in state of the art scalable topic models (Zaheer et al., 2016; Zhao et al., 2015; Chen et al., 2016; Li et al., 2017), and apply it LDA. Full details of SCA can be found in the appendix.

**Setup** We fit LDA (100 topics,  $\alpha = 0.1$ ,  $\beta = 0.1$ , 291k-word vocabulary after removing rare and stop-words as is customary) to 6.7 million English Wikipedia articles using 60 iterations of SCA distributed across eight 8-core machines, and measure the perplexity of the model after each iteration on 10k randomly sampled Reuters documents. For all experiments, we report the mean and standard-deviation of perplexity and timing

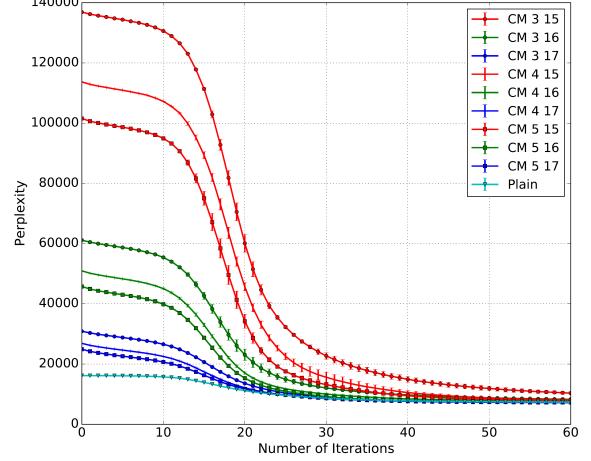


Figure 2: LDA perplexity with count-min sketch.

across three trials. Example topics from the various configurations are shown in the appendix. For more details, see Appendix C.1.

In this distributed setting, each machine must store a copy of the word-per-topic ( $wpt$ ) frequency counts, and at the end of an iteration, updated counts from different machines must be merged. However, each machine only needs to store the rows of the topics-per-document matrix ( $tpd$ ) pertaining to the documents it is processing. Hence, controlling the size of  $wpt$  is more important from a scalability perspective, so we will examine the effects of sketching  $wpt$ .

The data set and number of topics we are using for these tests are small enough that the non-sketched  $wpt$  matrix and documents can feasibly fit in each machine’s memory, so sketching is not strictly necessary in this setting. Our reason for using this data set is to be able to produce baselines of statistical performance for the non-sketched version to compare against the sketched versions.

**Experiment 1: Impact of the CM sketch.** In the first experiment, we evaluate the results of just using the CM sketch. We replace each row of the  $wpt$  matrix in baseline plain SCA with a count-min sketch. We vary the number of hash functions  $l \in \{3, 4, 5\}$  and the bits per hash from  $\{15, 16, 17\}$ . Figure 2 displays perplexity results for these configurations. While the more compressive variants of the sketches start at worse perplexities, by the final iterations, they converge to similar perplexities as the exact baseline with arrays. The range of the hash has a much larger effect than the number of hash functions in the earlier iterations

$l$	$\log_2(w)$	time (s)	size ( $10^5$ bytes)
NA	NA	$12.14 \pm 1.82$	1164.0
3	15	$22.75 \pm 4.30$	393.2
3	16	$23.90 \pm 4.41$	786.43
3	17	$25.32 \pm 4.68$	1572.9
4	15	$29.70 \pm 5.82$	524.3
4	16	$32.75 \pm 6.17$	1048.6
4	17	$33.35 \pm 5.89$	2097.2
5	15	$37.76 \pm 6.95$	655.4
5	16	$39.71 \pm 7.01$	1310.7
5	17	$42.33 \pm 7.75$	2621.4

Table 1: Time per iteration and size of  $wpt$  representation for LDA with CM sketch. The first row gives non-sketched baseline. 4-byte integers are used to store entries in the dense matrix and sketches.

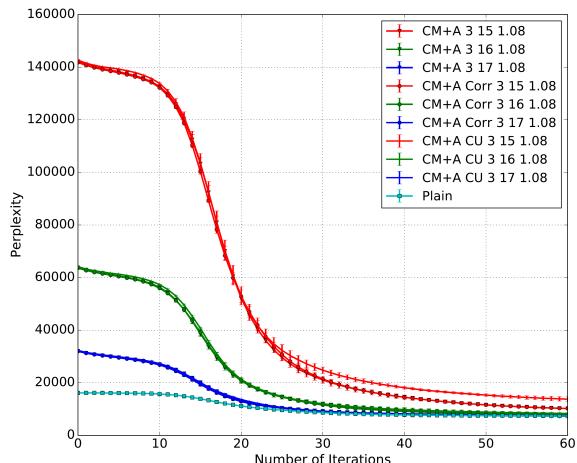


Figure 3: LDA perplexity with combined sketches.

of inference.

Table 1 gives timing and space usage (the first row corresponds to the baseline time and space). Recall that our main interest in sketching is to reduce space usage. Note that some of the parameter configurations here use more space than a dense array, so the purpose of including them is to better understand the statistical and timing effects of the parameters. Even though the smaller configurations do save space compared to the baseline, hashing the keys adds time overheads. Again, this is relative to the ideal case for the baseline, in which the documents and the full  $wpt$  matrix represented as a dense array can fit in main memory.

**Experiment 2: Combined Sketches** For the next experiment (Figure 3), we use the three variants of combined sketches with approximate counters de-

$l$	$\log_2(w)$	time (s)	size ( $10^5$ bytes)
NA	NA	$12.14 \pm 1.82$	1164.0
3	15	$12.58 \pm 2.00$	98.3
3	16	$17.57 \pm 2.78$	196.6
3	17	$22.69 \pm 3.72$	393.22

Table 2: Time per iteration results for LDA with combined sketch using 1-byte, base 1.08 independent independent counters. Timing for other update rules are similar.

scribed in Section 3 (sketch with independent counters (CM+A), sketch with correlated counters (CM+A Corr), and sketch with correlated counters and the conservative update rule (CM+A CU)). We use 1-byte base-1.08 approximate counters in order to represent a similar range as a 4-byte integer (but using 1/4 the memory). Given the results of the previous experiment, we just consider the case where 3 hash functions are used. In this particular benchmark, we do not see a large difference in perplexity between the various update rules, which again converge reasonably close to the perplexity of the baseline.

Table 2 gives timing and space usage for the combined sketches using the independent counter update rule. Each iteration runs faster than when just using the CM sketch with similar parameters. This is because the combined sketches are a quarter of the size of the CM sketch, so there is less communication complexity involved in sending the representation to other machines.

We explore a more comprehensive set of sketch and counter parameter effects on perplexity in Appendix E.3, run time in Appendix E.2, and example topics in Appendix E.1.

## 6 Conclusion

As machine learning models grow in complexity and datasets grow in size, it is becoming more and more common to use sketching algorithms to represent the data structures of learning algorithms. When used with MCMC algorithms, a primary question is what effect sketching will have on equilibrium distributions. In this paper we analyzed sketching algorithms that are commonly used to scale non-Bayesian NLP applications and proved that their use in various MCMC algorithms is justified by showing that sketch parameters can be tuned to reduce the distance between sketched and exact equilibrium distributions.

## References

- P. Alquier, N. Friel, R. Everitt, and A. Boland. Noisy Monte Carlo: Convergence of Markov chains with ap-

- proximate transition kernels. *ArXiv e-prints*, March 2014.
- Elaine Angelino, Matthew James Johnson, and Ryan P. Adams. Patterns of scalable bayesian inference. *Foundations and Trends in Machine Learning*, 9(2-3):119–247, 2016.
- Rémi Bardenet, Arnaud Doucet, and Christopher C. Holmes. Towards scaling up markov chain monte carlo: an adaptive subsampling approach. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 405–413, 2014.
- David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3: 993–1022, March 2003.
- Jianfei Chen, Kaiwei Li, Jun Zhu, and Wenguang Chen. Warplda: A cache efficient o(1) algorithm for latent dirichlet allocation. *Proc. VLDB Endow.*, 9(10):744–755, June 2016. ISSN 2150-8097.
- Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005. ISSN 0196-6774.
- Fan Deng and Davood Rafiei. New estimation algorithms for streaming data: Count-min can do more, 2007.
- Benjamin Van Durme and Ashwin Lall. Streaming pointwise mutual information. In *Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada*, pages 1892–1900, 2009a.
- Benjamin Van Durme and Ashwin Lall. Probabilistic counting with randomized storage. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1574–1579, 2009b.
- Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 19-23, 2002, Pittsburgh, PA, USA*, pages 323–336, 2002.
- Philippe Flajolet. Approximate counting: A detailed analysis. *BIT*, 25(1):113–134, June 1985.
- Leo N. Geppert, Katja Ickstadt, Alexander Munteanu, Jens Quedenfeld, and Christian Sohler. Random projections for bayesian regression. *Statistics and Computing*, 27(1):79–101, 2017.
- Amit Goyal and Hal Daumé III. Approximate scalable bounded space sketch for large data NLP. In *EMNLP*, pages 250–261, 2011.
- Amit Goyal, Hal Daumé III, and Graham Cormode. Sketch algorithms for estimating point queries in NLP. In *EMNLP-CoNLL*, pages 1093–1103, 2012.
- E. J. Gumbel. The maxima of the mean largest value and of the range. *The Annals of Mathematical Statistics*, 25(1):76–84, 1954. ISSN 00034851. URL <http://www.jstor.org/stable/2236513>.
- H. O. Hartley and H. A. David. Universal bounds for mean range and extreme observation. *Ann. Math. Statist.*, 25(1):85–99, 03 1954. doi: 10.1214/aoms/1177728848. URL <https://doi.org/10.1214/aoms/1177728848>.
- Jonathan H. Huggins, Trevor Campbell, and Tamara Broderick. Coresets for scalable bayesian logistic regression. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 4080–4088, 2016.
- Alan F. Karr. Weak convergence of a sequence of markov chains. *Wahrscheinlichkeitstheorie verw Gebiete*, 35(1):41–48, 1975.
- Kaiwei Li, Jianfei Chen, Wenguang Chen, and Jun Zhu. Saberlda: Sparsity-aware learning of topic models on gpus. In *ASPOLOS*, pages 497–509, 2017. ISBN 978-1-4503-4465-4.
- Wei Li and Andrew McCallum. Pachinko allocation: Dag-structured mixture models of topic correlations. In *International Conference on Machine Learning*, 2006.
- Torgny Lindvall. *Lectures on the Coupling Method*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002. ISBN 9780486421452.
- Robert Morris. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, October 1978. ISSN 0001-0782. doi: 10.1145/359619.359627. URL <http://doi.acm.org/10.1145/359619.359627>.
- J. Negrea and J. S. Rosenthal. Error Bounds for Approximations of Geometrically Ergodic Markov Chains. *ArXiv e-prints*, February 2017.
- N. S. Pillai and A. Smith. Ergodicity of Approximate MCMC Chains with Applications to Large Data Sets. *ArXiv e-prints*, May 2014.
- Guillaume Pitel and Geoffroy Fouquier. Count-min-log sketch: Approximately counting with approximate counters. *CoRR*, abs/1502.04885, 2015. URL <http://arxiv.org/abs/1502.04885>.
- Tim Roughgarden and Gregory Valiant. Approximate heavy hitters and the count-min sketch. Lecture

- notes., 2015. URL <http://theory.stanford.edu/~tim/s15/l/12.pdf>.
- Christopher De Sa, Christopher Ré, and Kunle Olukotun. Ensuring rapid mixing and low bias for asynchronous gibbs sampling. In *ICML*, pages 1567–1576, 2016.
- Guy L. Steele, Jr. and Jean-Baptiste Tristan. Adding approximate counters. In *PPoPP*, pages 15:1–15:12, 2016.
- Mikhail Yurochkin and XuanLong Nguyen. Geometric dirichlet means algorithm for topic inference. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2505–2513. 2016.
- Mikhail Yurochkin, Aritra Guha, and XuanLong Nguyen. Conic scan-and-cover algorithms for non-parametric topic modeling. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3881–3890. 2017.
- Manzil Zaheer, Michael Wick, Jean-Baptiste Tristan, Alex Smola, and Guy Steele. Exponential stochastic cellular automata for massively parallel inference. In *AISTATS*, volume 51 of *Proceedings of Machine Learning Research*, pages 966–975. PMLR, 09–11 May 2016.
- Bo Zhao, Hucheng Zhou, Guoqiang Li, and Yihua Huang. Zenlda: An efficient and scalable topic model training system on distributed data-parallel platform. *CoRR*, abs/1511.00440, 2015. URL <http://arxiv.org/abs/1511.00440>.

## A Consistency of CM sketch with Approximate Counters

In this section, we prove [Theorem 3.1](#) from [§3.1](#):

**Theorem 3.1.** *Let  $\psi_n = (b_n, l_n, w_n)$ . Suppose  $b_n \rightarrow 1$ ,  $w_n \rightarrow \infty$  and there exists some  $L$  such that  $l_n \leq L$  for all  $n$ . Let  $Q_{\psi_n}$  be a sketch with these parameters, using either independent counters or correlated counters. Then for all  $x$ ,  $Q_{\psi_n}(x)$  converges in probability to  $f_x$  as  $n \rightarrow \infty$ .*

*Proof.* Let  $N$  be the sum of the frequencies of all keys. Fix some key  $x$ , and write  $f_x$  for its true frequency. Now, for all  $\delta, \epsilon > 0$  we must show that there exists  $n_0$  such that for  $n' > n_0$ ,  $\Pr [|Q_{\psi_{n'}}(x) - f_x| > \epsilon] < \delta$ . Let  $X_{n,i}$  be the random variable for the  $i$ th counter for key  $x$  in sketch  $n$ , and let  $Z_{n,i}$  be the random variable indicating how many times this counter is incremented.

As in the regular analysis of the count-min sketch, we have that  $\mathbb{E}[Z_{n,i}] \leq f_x + \frac{N}{w_n}$  and  $Z_{n,i} - f_x \geq 0$ . Thus by Markov's inequality applied to this difference, we have that  $\Pr [Z_{n,i} - f_x > 1/2] \leq \frac{2N}{w_n}$ . Thus for large enough  $w_n$ , we can make this probability arbitrarily small. Since  $w_n \rightarrow \infty$ , there exists  $n_1$  such that for  $n > n_1$ ,  $\Pr [Z_{n,i} - f_x > 1/2] < \frac{\delta}{2L}$ . Since  $Z_{n,i}$  and  $f_x$  are both integers, this implies  $\Pr [Z_{n,i} \neq f_x] < \frac{\delta}{2L}$ . Now, applying the Chebyshev bound to  $X_{n,i}$  we have:

$$\Pr [|X_{n,i} - f_x| > \epsilon \mid Z_{n,i} = f_x] \leq \frac{b_n - 1}{2\epsilon^2} (f_x^2 - f_x)$$

So that once more, for  $b_n$  close enough to 1, this probability will be less than  $\frac{\delta}{2L}$ , and there exists some  $n_2$  such that for all  $n > n_2$ ,  $b_n$  will be sufficiently close to 1. Set  $n_0 = \max(n_1, n_2)$ , then we have that for  $n > n_0$ :

$$\begin{aligned} \Pr [|X_{n,i} - f_x| > \epsilon] &= \Pr [|X_{n,i} - f_x| > \epsilon \mid Z_{n,i} = f_x] \Pr [Z_{n,i} = f_x] \\ &\quad + \Pr [|X_{n,i} - f_x| > \epsilon \mid Z_{n,i} \neq f_x] \Pr [Z_{n,i} \neq f_x] \\ &\leq \frac{\delta}{L} \end{aligned}$$

Thus we have that for such  $n$ :

$$\begin{aligned} \Pr [|Q_{\psi_n}(x) - f_x| > \epsilon] &= \Pr \left[ \left| \min_i X_{n,i} - f_x \right| > \epsilon \right] \\ &\leq \sum_i^{l_n} \Pr [|X_{n,i} - f_x| > \epsilon] \\ &\leq l_n \cdot \frac{\delta}{L} \\ &\leq \delta \end{aligned}$$

where the second inequality follows from the union bound. Because we use the union bound, it does not matter whether the  $X_{n,i}$  are correlated for different  $i$ .  $\square$

The above argument can be adapted to handle the case when we add together a finite number of sketches using the [Steele and Tristan \(2016\)](#) rule for adding approximate counters. The idea is to argue that when the base of the counters of the two summands are close to 1, they will both with high probability be equal to the “true” number of increments that have been performed to each, and similarly so will the sum.

## B Bias from Merging Combined Sketches

As we have explained in the body of the paper, merging the combined sketches can be done using the addition routine of [Steele and Tristan \(2016\)](#). However, if these additions are done independently, this introduces potential for underestimation even in the case where correlated counters are used for incrementing within each sketch prior to merging. In this section we bound the resulting bias.

The expected values of maxima and minima of IID random variables is well studied. In particular, we have:

**Theorem B.1** ([Gumbel \(1954\)](#), [Hartley and David \(1954\)](#)). *Let  $Y_1, \dots, Y_l$  be a collection of iid random variables, such that  $\mathbb{E}[Y_1] = \mu$  and  $\mathbb{V}[Y_1] = \sigma^2$ .*

$$\mathbb{E}[\max_i Y_i] \leq \mu + \sigma \left( \frac{l-1}{\sqrt{2l-1}} \right)$$

Let  $X'_n$  be the value of an approximate counter obtained by adding together several independent base  $b$  counters that have been incremented a total of  $n$  times collectively.

Then, the results of [Steele and Tristan \(2016\)](#) show that:

$$\mathbb{V}[\phi(X'_n)] \leq \frac{b-1}{2}(n^2 - n) + \frac{1}{-2(b^2 - 4b + 1)}$$

Note that this bound holds regardless of how many counters were added together or what their relative sizes were.

As [Steele and Tristan \(2016\)](#) point out, for  $1 < b \leq 2$ , the right summand is less than  $\frac{1}{4}$ . Define  $\xi(n) = \sqrt{\frac{b-1}{2}(n^2 - n)} + \frac{1}{2}$ , then we have that  $\sqrt{\mathbb{V}[\phi(X'_n)]} \leq \xi(n)$  for  $b$  in this range.

### B.1 Merging Independent Counter Sketches

Let  $Y'_1, \dots, Y'_l$  be IID random variables such that each  $Y'_i$  is an approximate counter whose value is obtained

by adding together several independent base  $b$  counters that have been incremented a total of  $n$  times collectively. Let  $M'_{b,l}(n)$  be  $\min_i \phi(Y'_i)$ .

**Theorem B.2.** *If  $1 < b \leq 2$ , then*

$$\mathbb{E}[M'_{b,l}(n)] \geq n \left( 1 - \left( \sqrt{\frac{b-1}{2}} + \frac{1}{2n} \right) \left( \frac{l-1}{\sqrt{2l-1}} \right) \right)$$

*Proof.* Applying [Theorem B.1](#), we obtain:

$$\begin{aligned} \mathbb{E}[M'_{b,l}(n)] &\geq n - \xi(n) \left( \frac{l-1}{\sqrt{2l-1}} \right) \\ &\geq n \left( 1 - \left( \sqrt{\frac{b-1}{2}} \left( 1 - \frac{1}{n} \right) + \frac{1}{2n} \right) \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \\ &\geq n \left( 1 - \left( \sqrt{\frac{b-1}{2}} + \frac{1}{2n} \right) \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \end{aligned}$$

□

Then, if  $Q$  is a CM sketch obtained by merging several sketches using independent counters, and  $f_x$  is the total frequency of key  $x$  across the sketches, then  $\mathbb{E}[M'_{b,l}(f_x)] \leq \mathbb{E}[Q(x)]$

## B.2 Merging Correlated Counter Sketches

If we use the correlated increment rule of [Pitel and Fouquier \(2015\)](#), then the counters corresponding to a key within each sketch are not independent, hence they are not independent in the merged sketch either, so the results of [Theorem B.1](#) do not immediately apply. However, we can still obtain the same bound as in the independent case, as we will see.

To simplify the notation, we will just assume we are merging only two sketches; the proof generalizes to merging an arbitrary number of sketches. Let  $f_{x,1}$  be the frequency that  $x$  occurs in the data stream processed by the first sketch, and let  $f_{x,2}$  be the frequency in the second sketch, so that  $f_x = f_{x,1} + f_{x,2}$ . Since we seek a lower bound  $\mathbb{E}[Q(x)]$ , and hash collisions between  $x$  and other keys only possibly increase  $\mathbb{E}[Q(x)]$ , it suffices to bound the case when there are no collisions.

In that case, the correlated increment rule is the same whether we do conservative update or not, and the values of the counters for key  $x$  within sketch 1 are all equal, and similarly for all the counters within sketch 2. Let  $Y_1$  and  $Y_2$  denote these values. For each  $m$  and  $n$ , if we condition on  $Y_1 = m$  and  $Y_2 = n$ , then the results of adding the cells for key  $x$  are IID, and so [Theorem B.1](#) applies once more, so that we get:

$$\mathbb{E}[Q(x) \mid Y_1 = m, Y_2 = n] \tag{9}$$

$$\geq (n+m) - \xi(n+m) \left( \frac{l-1}{\sqrt{2l-1}} \right) \tag{10}$$

$$\geq (n+m) \left( 1 - \sqrt{\frac{b-1}{2}} \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \tag{11}$$

$$- \frac{1}{2} \left( \frac{l-1}{\sqrt{2l-1}} \right) \tag{12}$$

By the law of the total expectation, it follows that

$$\mathbb{E}[Q(x)] \tag{13}$$

$$\geq (\mathbb{E}[Y_1] + \mathbb{E}[Y_2]) \left( 1 - \sqrt{\frac{b-1}{2}} \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \tag{14}$$

$$- \frac{1}{2} \left( \frac{l-1}{\sqrt{2l-1}} \right) \tag{15}$$

$$= f_x \left( 1 - \sqrt{\frac{b-1}{2}} \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \tag{16}$$

$$- \frac{1}{2} \left( \frac{l-1}{\sqrt{2l-1}} \right) \tag{17}$$

$$= f_x \left( 1 - \left( \sqrt{\frac{b-1}{2}} + \frac{1}{2f_x} \right) \left( \frac{l-1}{\sqrt{2l-1}} \right) \right) \tag{18}$$

## C LDA

### C.1 Inference with SCA

SCA for LDA has the following parameters:  $I$  is the number of iterations to perform,  $M$  is the number of documents,  $V$  is the size of the vocabulary,  $K$  is the number of topics,  $N[M]$  is an integer array of size  $M$  that describes the shape of the data  $w$ ,  $\alpha$  is a parameter that controls how concentrated the distributions of topics per documents should be,  $\beta$  is a parameter that controls how concentrated the distributions of words per topics should be,  $w[M][N]$  is ragged array containing the document data (where subarray  $w[m]$  has length  $N[m]$ ),  $\theta[M][K]$  is an  $M \times K$  matrix where  $\theta[m][k]$  is the probability of topic  $k$  in document  $m$ , and  $\phi[V][K]$  is a  $V \times K$  matrix where  $\phi[v][k]$  is the probability of word  $v$  in topic  $k$ . Each element  $w[m][n]$  is a nonnegative integer less than  $V$ , indicating which word in the vocabulary is at position  $n$  in document  $m$ . The matrices  $\theta$  and  $\phi$  are typically initialized by the caller to randomly chosen distributions of topics for each document and words for each topic; these same arrays serve to deliver “improved” distributions back to the caller.

The algorithm uses three local data structures to store various statistics about the model (lines 2–4):

$tpd[M][K]$  is a  $M \times K$  matrix where  $tpd[m][k]$  is the number of times topic  $k$  is used in document  $m$ ,  $wpt[V][K]$  is an  $V \times K$  matrix where  $wpt[v][k]$  is the number of times word  $v$  is assigned to topic  $k$ , and  $wt[K]$  is an array of size  $K$  where  $wt[k]$  is the total number of time topic  $k$  is in use. We actually have two copies of each of these data structures because the algorithm alternates between reading one to write in the other, and vice versa.

The SCA algorithm iterates over the data to compute statistics for the topics (loop starting on line 9 and ending on line 32). The output of SCA are the two probability matrices  $\theta$  and  $\phi$ , which need to be computed in a post-processing phase that follows the iterative phase. This post-processing phase is similar to the one of a classic collapsed Gibbs sampler. In this post-processing phase, we compute the  $\theta$  and  $\phi$  distributions as the means of Dirichlet distributions induced by the statistics.

In the iterative phase of SCA, the values of  $\theta$  and  $\phi$ , which are necessary to compute the topic proportions, are computed on the fly (lines 21 and 22). Unlike the Gibbs algorithm, where in each iteration we have a back-and-forth between two phases, where one reads  $\theta$  and  $\phi$  in order to update the statistics and the other reads the statistics in order to update  $\theta$  and  $\phi$ ; SCA performs the back-and-forth between two copies of the statistics. Therefore, the number of iterations is halved (line 9), and each iteration has two subiterations (line 10), one that reads  $tpd[0]$ ,  $wpt[0]$ , and  $wt[0]$  in order to write  $tpd[1]$ ,  $wpt[1]$ , and  $wt[1]$ , then one that reads  $tpd[1]$ ,  $wpt[1]$ , and  $wt[1]$  in order to write  $tpd[0]$ ,  $wpt[0]$ , and  $wt[0]$ .

**Sketching and hashing** Modifying SCA to support feature hashing and the CM sketch is fairly simple. The read of  $wpt$  on line 22 and the write of  $wpt$  on line 27 are replaced by the read and write procedures of the CM sketch, respectively. Note that the input data  $w$  on line 1 is not of type int anymore but rather of type string. Consequently, the data needs to be hashed before the main iteration starts on line 9. We can replace the  $V$  used on line 22 with the size of the hash space.

**Implementation and Hardware** Our implementation is written in Java 8. To achieve good performance, we use only arrays of primitive types and preallocate all of the necessary structures before the learning starts. We implement multi-threaded parallelization within a node using the Fork/Join work-stealing framework, and distribute across multiple nodes using the Java binding to OpenMPI. Our implementation makes use of the Alias method to sample the topics and leverage

the document sparsity. We run our experiments on a small cluster of 8 nodes connected through 10Gb/s Ethernet. Each node has two 8-core Intel Xeon E5 processors (some nodes have Ivy Bridge processors while others have Sandy Bridge processors) for a total of 32 hardware threads per node and 512GB of memory. We use 4 MPI processes per node and set Java’s memory heap size to 10GB.

## D Microbenchmarks

In these microbenchmarks we measure mean relative error when estimating bigram frequencies using various CM sketches. The test set is a corpus of 2 million tokens drawn from a snapshot of Wikipedia with 769,722 unique bigrams.

Figure 5 shows a comparison between CM with and without approximate counters. We use  $w = 6666$  for the approximate counter sketch and  $w = 3333$  for the conventional sketch, with  $l = 3$  for both, so that the approximate counter sketches have twice as many total counters. Since the most frequent bigram in this sample occurs 19430 times, the conventional sketch requires at least 16 bit integers, while only 8 bits would suffice for each approximate counter, hence total space usage is approximately the same. We evaluate both kinds of sketches with and without conservative update. As expected, conservative update dramatically lowers error in both cases, particularly for less frequent words. The larger value of  $w$  enabled by approximate counters further lowers error on less frequent words, though for more frequent words there is some increased error. Additional benchmarks measuring the effect of the counter base and errors resulting from merging sketches are given in Appendix E.

Figure 6 and Figure 7 show the effects of varying the counter base  $b$  when using the independent counter and conservative update alternatives. There are two interesting phenomena in these plots. First, when using the independent counters, we see that the error for low frequency words is slightly but consistently lower when the base is larger. This makes sense because we know that the sketch overestimates such words, but when we take the minimum of independent counters the expected value of the result is smaller, which reduces the error for infrequent words – with a larger counter base, it is more likely that the minimum counter happens to underestimate. The second effect is that when using conservative update with a large base, there appear to be more instances with large error when estimating frequent words, compared to the independent case. Again, a plausible explanation is that these errors correspond to cases where the approximate counters occasionally take on a much larger value than the true count, and

$M$	(Number of documents)
$\forall m \in \{1..M\}, N_m$	(Length of document $m$ )
$V$	(Size of the vocabulary)
$K$	(Number of topics)
$\alpha \in \mathbb{R}^K$	(Hyperparameter controlling documents)
$\beta \in \mathbb{R}^V$	(Hyperparameter controlling topics)
$\forall m \in \{1..M\}, \theta_m \sim Dir(\alpha)$	(Distribution of topics in document $m$ )
$\forall k \in \{1..K\}, \phi_k \sim Dir(\beta)$	(Distribution of words in topic $k$ )
$\forall m \in \{1..M\}, \forall n \in \{1..N_m\}, z_{mn} \sim Cat(\theta_m)$	(Topic assignment)
$\forall m \in \{1..M\}, \forall n \in \{1..N_m\}, w_{mn} \sim Cat(\phi_{z_{mn}})$	(Corpus content)

```

1: procedure SCA(int  $I$ , int  $M$ , int  $K$ , int  $N[M]$ , float  $\alpha$ , float  $\beta$ , int  $w[M][N]$ )
2:   local array int  $tpd[2][M][K]$ 
3:   local array int  $wpt[2][H][R_2][K]$ 
4:   local array int  $wt[2][K]$ 
5:   initialize array  $tpd[0]$                                  $\triangleright$  Randomly chosen distributions
6:   initialize array  $wpt[0]$                                  $\triangleright$  Randomly chosen distributions
7:   initialize array  $wt[0]$                                  $\triangleright$  Randomly chosen distributions
8:    $\triangleright$  The main iteration
9:   for  $i$  from 0 through  $(I \div 2) - 1$  do
10:    for  $r$  from 0 through 1 do
11:      clear array  $tpd[1 - r]$                                  $\triangleright$  Set every element to 0
12:      clear array  $wpt[1 - r]$                                  $\triangleright$  Set every element to 0
13:      clear array  $wt[1 - r]$                                  $\triangleright$  Set every element to 0
14:       $\triangleright$  Compute new statistics by sampling distributions
15:       $\triangleright$  that are computed from old statistics
16:      for all  $0 \leq m < M$  do
17:        for all  $0 \leq n < N$  do
18:          local array float  $p[K]$ 
19:          let  $v \leftarrow w[m][n]$ 
20:          for all  $0 \leq k < K$  do
21:            let  $\theta \leftarrow (tpd[r][m][k] + \alpha)/(N[m] + K \times \alpha)$ 
22:            let  $\phi \leftarrow (wpt[r][v][k] + \beta)/(wt[r][k] + V \times \beta)$ 
23:             $p[k] \leftarrow \theta \times \phi$ 
24:          end for
25:          let  $z \leftarrow sample(p)$                                  $\triangleright$  Now  $0 \leq z < K$ 
26:          increment  $tpd[1 - r][m][z]$                                  $\triangleright$  Increment counters
27:          increment  $wpt[1 - r][v][z]$ 
28:          increment  $wt[1 - r][z]$ 
29:        end for
30:      end for
31:    end for
32:  end for

```

Figure 4: Pseudocode for Streaming SCA

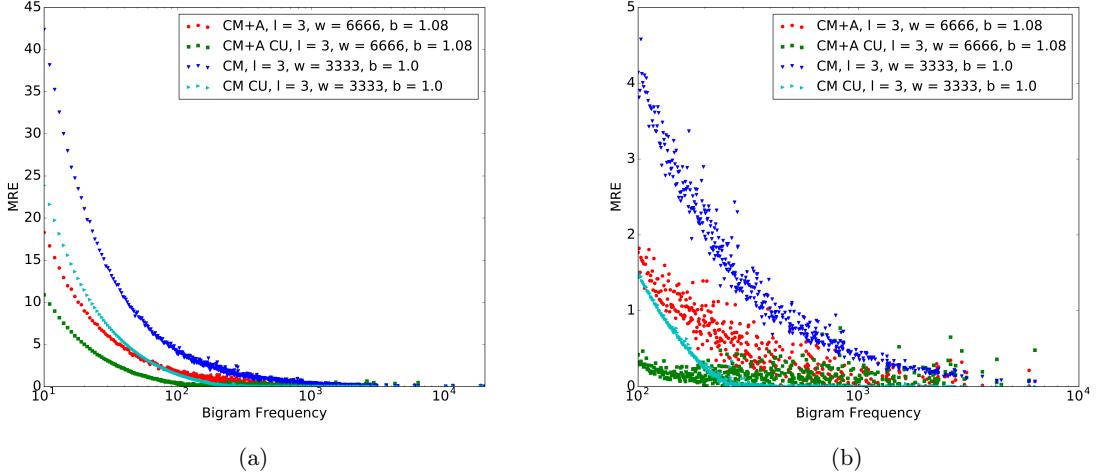


Figure 5: Experiment measuring mean relative error for bigram estimation. The  $y$ -value of each point on the plot is equal to the mean relative error for all bigrams whose true frequency is equal to the  $x$ -value of the point, averaged over 5 trials; the right figure is the same data zoomed to a subset of the  $x$ -axis. “CM+A” is a combined sketch using independent counters; “CM+A CU” is a combined sketch with correlated conservative update. “CM CU” and “CM” are conventional sketches with/without conservative update, respectively.

that when taking the minimum of several independent counters this is unlikely to happen.

**Figure 8** shows just the points for the  $b = 2$  conservative update case from these same experiments. There is a noticeable repeating “crisscross” pattern in the errors for the more frequent keys. This arises because the base is so large the counter can only represent counts of the form  $2^k - 1$ , so the mean relative error for these keys is largely dependent on how close they are to a value that can be represented in this form.

Finally, **Figure 9** shows the error when merging different sketches together. In these experiments, we split the corpus into  $k$  substreams of equal size, compute sketches on each, and then merge the  $k$  sketches together. We vary  $k$  from 1 to 4, and use conservative update when computing each of the subsketches. There does not appear to be a substantial difference between the errors when merging or not when using  $b = 1.08$ .

## E Experiments

In this section we present additional experiments, provide example topics and report timing results for the various algorithms.

### E.1 Example Topics

Since it is possible for a model to have good perplexity, but yield poor topics, we also manually inspect

the top-words per topics to ensure that the topics are reasonable in our LDA experiments. In general, we find no perceptible difference in quality between the models that employ sketching and hashing representations of the sufficient statistics and those that employ exact array-based representations. In Table 3 we provide example topics from three different models, all of which use representations that compress the sufficient statistics more than a traditional array of 4-byte integers. The first two systems combine count-min sketch with approximate counters while the third combines all three ideas: count-min sketch, feature hashing and approximate counters. As is typical of LDA and other topic-models, not all topics are perfect and some topics arise out of idiosyncrasies of the data like a topic full of dates from a certain century, but again, there did not appear to be noticeable differences in quality between the systems.

### E.2 Timing results

We report the timing results in this section.

**CM sketch timing** Although the CM sketch representation of the sufficient statistics behaves well statistically, there are some computational concerns. In particular, the CM sketch stores multiple counts per word (one per-hash function) and a distributed algorithm must then communicate the extra counts over the network. Therefore, to evaluate the effect on run-time performance, we also report the average per-iteration

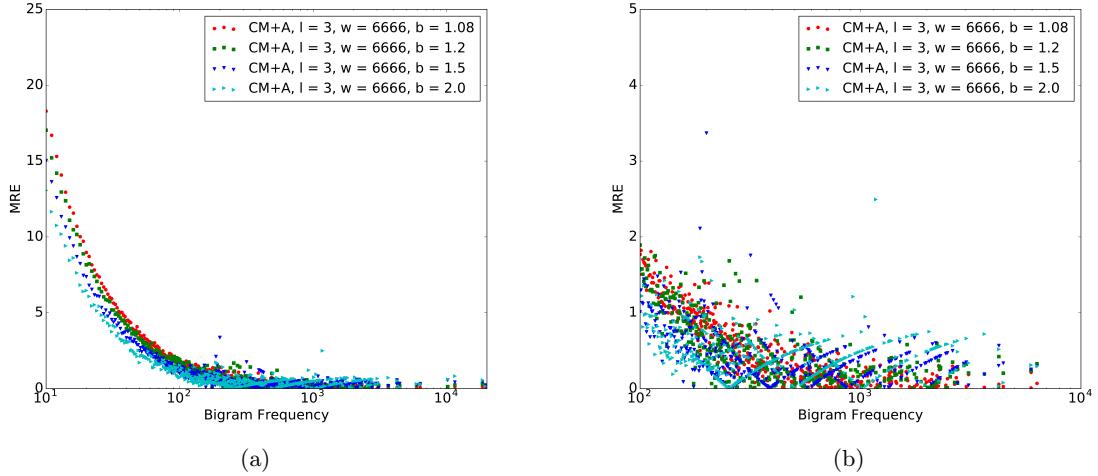


Figure 6: Effects of different approximate counter base  $b$  for bigram estimation benchmark using independent counters. The figure on the right is a zoomed in version of the same data.

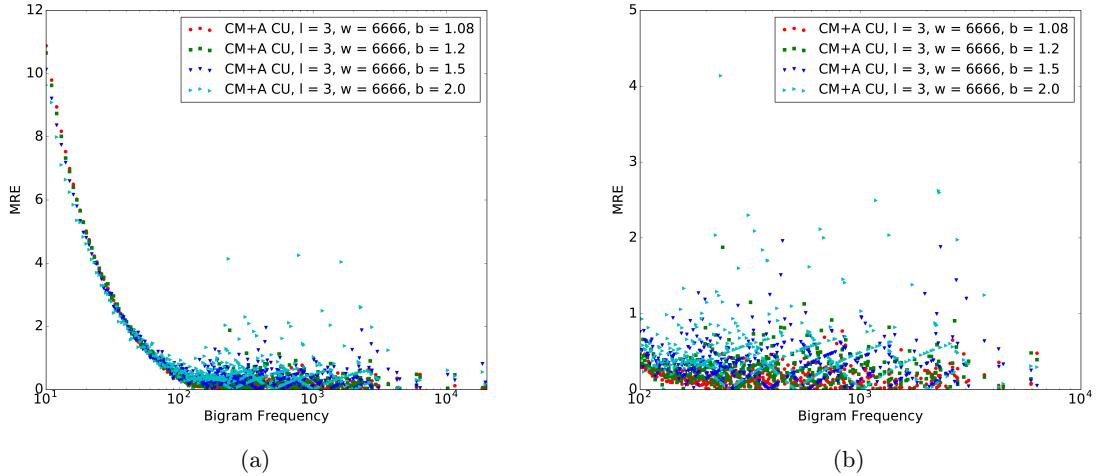


Figure 7: Effects of different approximate counter base  $b$  for bigram estimation benchmark using conservative update.

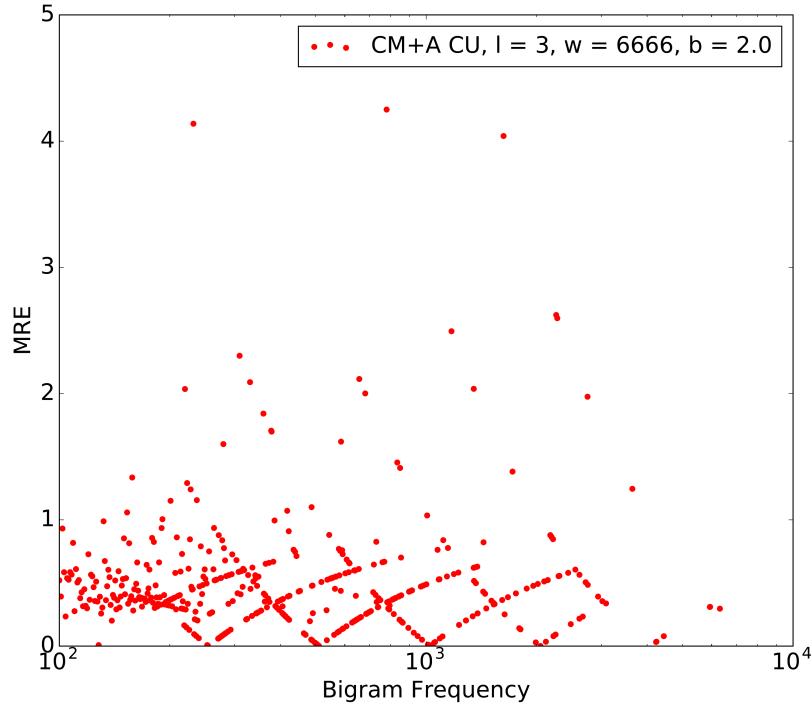


Figure 8: Results showing crisscross pattern for large frequency key errors with large bases.

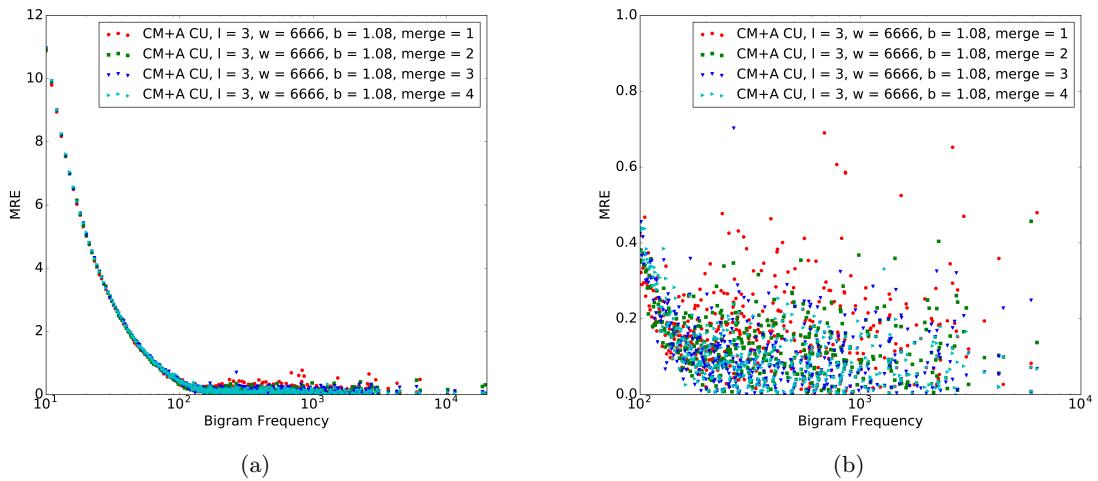


Figure 9: Effects of merging multiple sketches together using the addition procedure of Steele and Tristan (2016).

Topic A	Topic B	Topic C
CM+A 3 16 1.08		
space	episode	law
light	series	court
energy	season	act
system	show	states
earth	episodes	legal
CM+A 3 15 1.04		
russian	john	man
war	william	time
government	died	story
soviet	king	series
union	henry	back
CM+H+A 3 18 22 1.08		
system	band	court
high	album	police
power	released	case
systems	rock	law
device	records	prison

Table 3: Example topics. “CM+A 3 16 1.08” indicates CM-sketch (three 16-bit hashes) with approximate counters (8-bit base-1.08), “CM+A 3 15 1.04” indicates CM-sketch (three 15-bit hashes) with approximate counters (8-bit base-1.04), and “CM+H+A 3 16 22 1.08” indicates CM-sketch (three 18-bit hashes) with feature hashing (22-bit) and approximate counters (8-bit base-1.08). All three configurations are more compressive than the traditional array-based representation of the sufficient statistics that utilize 4-byte integers.

Method	# hashes	hash range	time (s)
SSCA	NA	NA	12.14 ± 1.82
SSCAS	3	15	22.75 ± 4.30
SSCAS	3	16	23.90 ± 4.41
SSCAS	3	17	25.32 ± 4.68
SSCAS	3	18	28.10 ± 5.18
SSCAS	4	15	29.70 ± 5.82
SSCAS	4	16	32.75 ± 6.17
SSCAS	4	17	33.35 ± 5.89
SSCAS	4	18	36.18 ± 5.97
SSCAS	5	15	37.76 ± 6.95
SSCAS	5	16	39.71 ± 7.01
SSCAS	5	17	42.33 ± 7.75
SSCAS	5	18	45.47 ± 7.70
SSCAS	6	15	45.04 ± 8.51
SSCAS	6	16	46.38 ± 8.18
SSCAS	6	17	48.70 ± 8.21
SSCAS	6	18	53.03 ± 8.89
SSCAS	7	15	51.66 ± 9.45
SSCAS	7	16	53.35 ± 9.38
SSCAS	7	17	56.44 ± 9.42
SSCAS	7	18	61.15 ± 9.93

Table 4: Time per iteration results for LDA with CM sketch.

wall-clock time for each system in Table 4. The method “SSCA” is the default implementation of SCA the employs arrays to represent the sufficient statistics and methods marked “SSCAS” employ the CM sketch.

As expected, the number of hash functions has a bigger impact on running time than the range of the hash functions. Fortunately, at least empirically, the range of the hash function is more important than the number of hash functions in that it has a greater effect on inference’s ability to achieve higher accuracy in fewer iterations. Thus, in terms of both the number of iterations and the wall-clock time, increasing the hash range is more beneficial than increasing the number of hashes.

**Timing with approximate counters** Although using the CM sketch with these dimensions increases the running time as demonstrated above, approximate counters effectively compensate for the increased communication overhead because the corresponding sketches are much smaller for a given number of hash functions and range. We report the results in Table 5. The first row, method “SSCA” is the default implementation of SSCA using array representation of the sufficient statistics and “SSCASA” is the version with sketching and approximate counters (8 bits and base 1.08).

Method	# hashes	hash range	time (s)
SSCA	NA	NA	$12.14 \pm 1.82$
SSCASA	3	15	$12.58 \pm 2.00$
SSCASA	3	16	$17.57 \pm 2.78$
SSCASA	3	17	$22.69 \pm 3.72$
SSCASA	3	18	$24.29 \pm 3.93$
SSCASA	4	15	$17.00 \pm 2.82$
SSCASA	4	16	$24.50 \pm 4.18$
SSCASA	4	17	$29.46 \pm 4.88$
SSCASA	4	18	$32.19 \pm 5.30$
SSCASA	5	15	$22.11 \pm 3.84$
SSCASA	5	16	$31.26 \pm 5.54$
SSCASA	5	17	$37.78 \pm 6.60$
SSCASA	5	18	$41.12 \pm 6.74$

Table 5: Time per iteration results for LDA with CM sketch and approximate counters.

### E.3 Exploring more sketching and hashing parameters

In this section, we report a wider range of settings to the parameters of the CM sketch. In particular, we report numbers for a sketch with a range of just 15-bits to one with 18, while also varying the number of hash functions from 3 to 7. To make the plots more readable, we depict curves for sketches with the same hash range in the same color (for example, all sketches with a 15-bit range are red). We also depict curves for sketches with the same number of hash functions with the same symbol (for example, sketches with three hash functions are all marked with circles).

In Figure 10(a) we report the results for just the CM sketch and begin to reach the limits of our ability to push the compressiveness of the sketch. We see that the final perplexity (after 60 iterations) is the same in all cases except for the most compressive variant of the sketch (with 15 bits and 3 hash functions). Since the vocabulary size is 291,561, this sketch is one-third of the size as the raw array-based representation for representing word counts per topic. This seems to be the point at which we begin to see worse final perplexity.

We can also see from this plot that the hash range (curves from the same hash-range are in the same color, and curves with different ranges are in different colors) has a bigger impact on initial performance than the number of hash functions. For example, if we wanted to double the size of the sketch, it would be better to add an extra bit to the range than to double the number of hash functions from 3 to six (as seen by the perplexity gaps in the early iterations between each of the hash ranges). This is in line with the earlier results of Goyal and Daumé III (2011); Goyal et al. (2012).

In Figure 10(b) we repeat the same experiment, but employ an 8-bit base-1.08 approximate counter to represent the counts in the sketch (instead of the usual 4-byte integers). We include up to 5 hashes for this plot. Note that despite using just one-quarter the amount of memory to represent the sufficient statistics, the results for these combined data-structures are similar to the CM sketch alone. Further, as noted earlier, the approximate counters are much faster in a distributed setting since they overcome the additional data that needs to be transmitted by the CM sketch. Thus, the combined data-structure is not only more compressive than the CM sketch alone, but it also runs much faster, achieving similar performance as the original algorithm depending on the setting to its parameters.

Finally, as we mentioned in Section 3, combining the CM sketch and the approximate counters is non-obvious due to the way the min operation interacts with the counters. We have proposed and discussed several alternatives: CM sketch with independent counters, CM sketch with correlated counters, and CM sketch with correlated counters and the conservative update rule to reduce the bias. We show the plots for these counters in Figures 11, 12, 13 respectively. We also vary the base of the counters while keeping the number of counter bits fixed at 8. Each color represents a different base (1.08, 1.09 and 1.10) to make it easier to interpret. The main takeaways from these plots is that the method in which you combine the counters and min-sketch does not matter as much for an application like LDA, which seems to be robust to the bias in the first two methods. We note that in some cases, increasing the base of the counter appears to improve perplexity. While we have not been able to find a satisfactory explanation for this phenomenon, previous work on the geometric aspects of topic modeling Yurochkin and Nguyen (2016); Yurochkin et al. (2017) has highlighted the subtle interaction between the geometry of the topic simplex and perplexity.

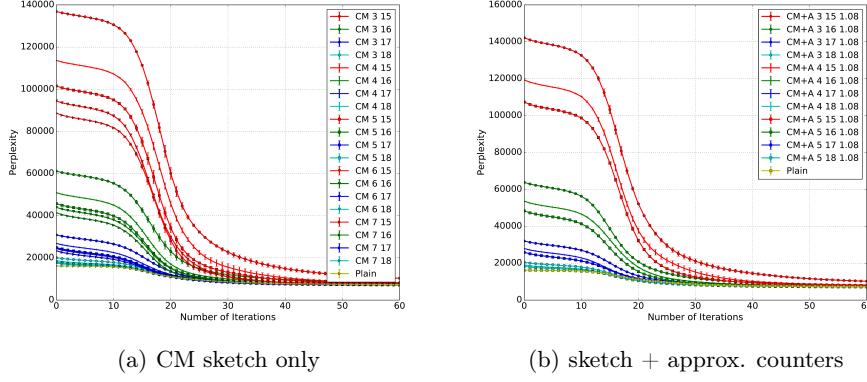


Figure 10: Experiments across a full range of parameter settings for the sketch.

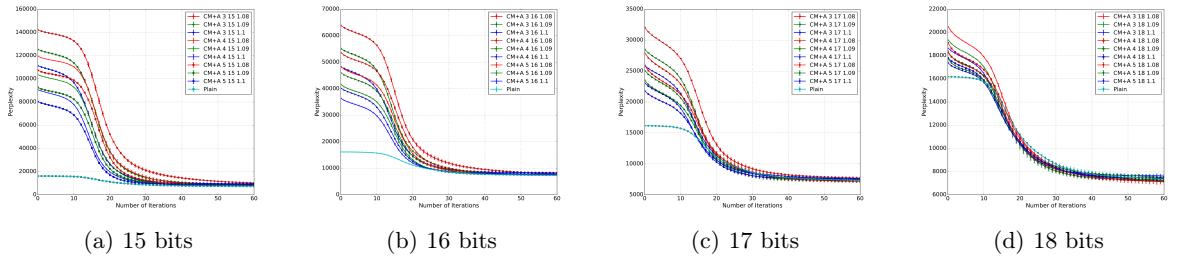


Figure 11: *Independent counter* variant of the combined CM sketch and approximate counter data-structure on LDA with a hash range of 15,16,17 and 18 as indicated in the respective captions.

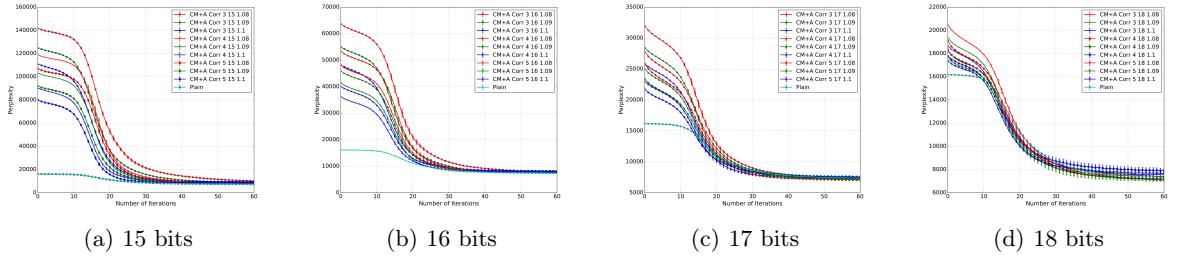


Figure 12: *Correlated counter* variant of the combined CM sketch and approximate counter data-structure on LDA with a hash range of 15,16,17 and 18 as indicated in the respective captions.

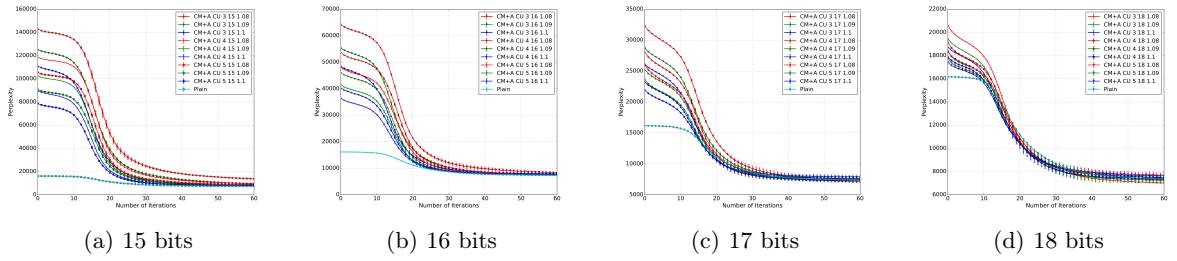


Figure 13: *Correlated counter + conservative update* variant of the combined CM sketch and approximate counter data-structure on LDA with a hash range of 15,16,17 and 18 as indicated in the respective captions.

# Scaling Hierarchical Coreference with Homomorphic Compression

**Michael Wick**

*Oracle Labs,  
Burlington, MA*

MICHAEL.WICK@ORACLE.COM

**Swetasudha Panda**

*Orable Labs,  
Burlington, MA*

SWETASUDHA.PANDA@ORACLE.COM

**Joseph Tassarotti**

*Massachusetts Institute of Technology  
Cambridge, MA*

JTASSARO@MIT.EDU

**Jean-Baptiste Tristan**

*Orable Labs,  
Burlington, MA*

JEAN.BAPTISTE.TRISTAN@ORACLE.COM

## Abstract

Locality sensitive hashing schemes such as simhash provide compact representations of multisets from which similarity can be estimated. However, in certain applications, we need to estimate the similarity of dynamically changing sets. In this case, we need the representation to be a homomorphism so that the hash of unions and differences of sets can be computed directly from the hashes of operands. We propose two representations that have this property for cosine similarity (an extension of simhash and angle-preserving random projections), and make substantial progress on a third representation for Jaccard similarity (an extension of minhash). We employ these hashes to compress the sufficient statistics of a conditional random field (CRF) coreference model and study how this compression affects our ability to compute similarities as entities are split and merged during inference. We also provide novel statistical analysis of simhash to help justify it as an estimator inside a CRF, showing that the bias and variance reduce quickly with the number of hash bits. On a problem of author coreference, we find that our simhash scheme allows scaling the hierarchical coreference algorithm by an order of magnitude without degrading its statistical performance or the model’s coreference accuracy, as long as we employ at least 128 or 256 bit hashes. Angle-preserving random projections further improve the coreference quality, potentially allowing even fewer dimensions to be used.

## 1. Introduction

Probabilistic models in machine learning, such as conditional random fields (CRFs), are widely successful at modeling many problems at the heart of knowledge base construction, including those in natural language processing, information extraction and data integration. However, when dealing with natural language data, the underlying feature representations are often sparse, high-dimensional and dynamic (*i.e.*, they change during inference). In this paper we consider the task of coreference resolution, in which the goal is to partition a set of mentions into the entities to which they refer. We might represent each mention with a

feature vector in which each dimension corresponds to a word or  $n$ -gram. Since only a small subset of the vocabulary is observed per mention, most elements of the vector are zero.

Given the model and these representations, inference entails making decisions about whether two entities should be coreferent. To make such decisions, the model estimates a probability that involves computing the similarities between the aggregate feature representations of the two entities' mentions. Since *all* the vectors are both sparse and high-dimensional, these similarity operations are computationally expensive because the sparse data structures supplant the dense arrays that would otherwise support fast look-ups. Moreover, as the inference algorithm makes decisions about whether or not two entities are coreferent, we may have to split or merge the entities and thus we must *update the feature vector* to reflect these changes. Maintaining such sparse-vector representations in the inner-loop of probabilistic inference is expensive, especially as the entities grow in size.

In order to cope with the computational problems associated with sparse, high dimensional dynamic feature representations, we propose using *homomorphic compression*, in which the compressed representations of intermediate inference results can be computed directly from their operands, allowing inference to run directly on the compressed representations of the data even as they change. In this paper, we consider several such schemes to scale hierarchical coreference. First, we propose a novel homomorphic cosine-preserving hashing scheme based on simhash [Charikar, 2002] that also supports addition and subtraction to more efficiently represent the data and the evolving intermediate results of probabilistic inference. Second, because various linear norm-preserving random projections also preserve angles [Magen, 2002], we can directly compute cosine similarity on projected data – linearity of the projections means that they too can be updated dynamically. The resulting angle estimates are superior to the homomorphic simhash representation, at the cost of reduced efficiency for certain operations. Third, we develop a homomorphic version of minhash [Broder, 1997b] to support Jaccard similarity. Our current algorithm is biased, but the bias appears small in practice for the situations we have considered. Although the minhash based set representations are not currently employed in hierarchical coreference, they might be useful in other models or applications that involve binary features over changing sets [Culotta et al., 2007b].

We provide error analysis for all three schemes, collating and extending known results, and in the case of simhash, we provide novel statistical analysis for its use as a direct estimator for  $\cos(\theta)$  that shows the bias and variance decrease rapidly with the number of bits, helping to justify its use in a CRF for a task like coreference. On a hierarchical model for coreference resolution, the proposed simhash scheme improves the speed of probabilistic inference by an order of magnitude while having little effect on model quality. Moreover, we find that the underlying random projection representation can provide even better cosine estimates than simhash, at the cost of not being able to use certain fast bitwise-operations to compute similarities. Finally, we briefly evaluate homomorphic minhash and show that even though there are possible pathological cases, as long as we employ enough hash functions, the estimates are reasonably close to the true Jaccard, albeit, biased.

## 2. Coreference Resolution: Background and Challenges at Scale

**Coreference resolution** Coreference resolution is the problem of determining whether different *mentions* refer to the same underlying *entity* [Getoor and Machanavajjhala, 2012].

For example, in the sentence “In a few years [McDaniels] would replace [Belichick] as the forty-two year old [quarterback], [Tom Brady] retires.” coreference must correctly determine that “quarterback” refers to Tom Brady and not Belichick or McDaniels. This type of coreference is sometimes referred to as noun-phrase or within document coreference and often relies upon linguistic and discourse motivated features [Raghunathan et al., 2010]. Coreference resolution arises in many other situations; for example, when merging two or more databases together it is desirable to remove duplicates that result from the merge, a problem sometimes termed record linkage or deduplication [Newcombe et al., 1959]. Coreference is also foundational to *knowledge base construction* which requires that we combine information about entities of interest from multiple sources that might mention them in different contexts. For example, if we were to build a knowledge base of all scientists in the world — similar to Google scholar — we would need to perform the task of *author coreference* to determine who authored what papers [Han et al., 2004, Culotta et al., 2007a]. Are the following two mentions of “J Smith” the same author?

*V Khachatryan, AM Sirunyan,..., J Smith.* Observation of the diphoton decay of the Higgs boson and measurement of its properties. *The European Physical Journal 2014*

*S Chatrchyan, V Khachatryan, AM Sirunyan, A Tumasyan, W Adam, J Smith.* Jet production rates in association with  $W$  and  $Z$  bosons in  $pp$  collisions. *J High Energy Physics 2012*

Although generally this is a difficult problem, it can be solved with machine learning since features of the mentions such as the words in the title (both have “Boson” in common), the topic of the title (both are about a similar subfield of physics), the journal (both are physics journals) and the co-authors (there appears to be a co-author in common) provide some evidence about whether or not the two “J Smith’s” might be the same person.

In order to solve the problem, it is thus common to extract such contextual features about each mention, such as in the above example, features from the title, co-author list, venue, year and author-name and employ them in a probabilistic model. These features are typically the raw words, character-ngrams and normalized variants thereof, sometimes with positive real-valued weights to indicate the importance (e.g., via TFIDF) of each feature. Then, given such features, a coreference model measures the similarities between mentions via functions such as cosine-similarity. In contrast to within document coreference discussed earlier, this type of coreference resolution problem is better suited for similarity based models, such as the ones we will use in the following. Moreover, since coreference decisions are not restricted by document-boundaries, the entities can grow unbounded in size, making compact representations of their growing feature sets especially important.

Typically, the model is a discriminative conditional random field (CRF) that measure the probability of an assignment of mentions to entities conditioned on the observed features [McCallum and Wellner, 2003]. The model factorizes into potentials that score local coreference decisions. Local search procedures such as greedy-agglomerative clustering or Markov-chain Monte Carlo (MCMC) find the most likely assignment of mentions to entities [McCallum and Wellner, 2003, Culotta et al., 2007a, Wick et al., 2012].

In *pairwise* models, potential functions measure the compatibility of two mentions being in the same cluster. An issue with such models is that the possible pairwise comparisons scales

quadratically with the number of mentions. An alternative class of models that avoids this quadratic blow-up are *entity-based*, in which entities are treated as first-class variables with their own set of inferred features, and potentials measure compatibility between mentions and entities. However, entity-based models come with their own scalability challenges. To illustrate the problem (and our solution), we focus on an entity-based model called *hierarchical coreference*, which recently won a challenge to disambiguate inventor names for the USPTO, due to its accuracy and scalability [Uni, 2016, Monath and McCallum, 2016].

**Hierarchical Coreference** In the hierarchical coreference model, mentions are organized into latent tree structures [Wick et al., 2012]. There is one tree per entity with mentions at the leaves and intermediate nodes as “subentities” that organize subsets of the entity’s mentions. Rather than modeling interactions between mention-pairs, the potential functions measure compatibility between child and parent nodes in the tree. The score of a given assignment of mentions into latent trees is the product of all model potentials which includes these child-parent compatibility scores as well as some additional priors on tree-shape and entities. These compatibility scores are parametrized cosine functions.

Each mention is endowed with multiple feature variables that each capture a subset of the total features. For example, in author coreference, one feature variable might capture features of the author’s name and another might capture features of the title and venue of the paper. Colloquially, we refer to these feature variables as “bags” since they inherit the usual bags-of-words assumption. We distinguish between different “bag-types” which each capture different types of features (e.g., the author name bag, title words bag, co-author bag, etc). The other nodes in the tree also contain feature variables (bags), but the values of these variables are determined by the current assignment of children to that parent node. In particular, *a parent’s bag is the sum of all its children’s bags*. In order to maintain this invariant, the bag representations must be updated to reflect the current state of inference, and hence for efficiency reasons, representations must be homomorphic with respect to operations that will be performed on the bags.

Interpreting bags as vectors, the cosine distance between them is used to calculate their compatibility. The primary potential functions measure the compatibility between each child’s bag and its parent’s bag. There is one potential for each bag-type. For example, to measure the compatibility between a node  $z_i$  and  $z_j$ , let  $y_{ij}$  be the binary variable that is 1 if and only if  $z_j$  is the parent of  $z_i$ , and let  $b_i^{(0)}$  and  $b_j^{(0)}$  be a bag for  $z_i$  and  $z_j$  respectively, then the potential  $\psi^{(0)}$  for “bag 0” scores a coreference decision as:

$$\psi^{(0)}(z_i, z_j, y_{ij}; w, t) = \begin{cases} 1 & y_{ij} = 0 \\ \exp\left(w(\cos(b_i^{(0)}, b_j^{(0)}) - b_i^{(0)}) - t\right) & y_{ij} = 1 \end{cases} \quad (1)$$

where  $w$  is a real-valued weight and  $t$  is a real-valued translation parameter for potential  $\psi^{(0)}$ . The potentials for each bag-type have parameters  $w, t$  that we can fit to data.

Because only a small handful of features are ever observed for a given mention, typical implementations of hierarchical coreference employ sparse-vector representations to represent bags (e.g., the implementation found in FACTORIE [McCallum et al., 2008, 2009]).

However, a key disadvantage of sparse vector representations is that they must store the indices and weights of the non-zero elements, which means that the data-structures

must dynamically change in size as MCMC splits and merges entities. As the sizes of the entities grow, these operations become increasingly expensive. Similar issues arise in other entity-based models where features of entities are aggregated from those of mentions. Thus, while entity-based models avoid the quadratic comparison issue of pairwise models, a straight-forward sparse representation of their feature vectors is no longer efficient.

Is there an alternative representation of feature vectors which (a) allows fast evaluation of cosine-similarity, and (b) can be efficiently dynamically updated? As we describe in the next section, the simhash hashing function [Charikar, 2002] provides a representation with property (a). However, the standard simhash cannot be updated as vectors are modified. We therefore develop a variant which we call *homomorphic* simhash, which has both properties (a) and (b). We also identify two other schemes that support these properties.

### 3. Homomorphic Representations for Measuring Similarity

We now discuss three different homomorphic representations that support addition and subtraction in the compressed space, while preserving similarity estimates. We propose homomorphic simhash and a related random projection for cosine similarity of high-dimensional vectors and multisets; and homomorphic minhash for Jacard similarity of sets.

#### 3.1 Homomorphic simhash and its Statistical Properties

**Background: simhash** A *locality-sensitive hash function* for a distance metric  $d$  on a set of objects  $S$  is a function  $H$  such that given  $x, y \in S$ , we can estimate  $d(x, y)$  from the hashed representations  $H(x)$  and  $H(y)$ . Simhash [Charikar, 2002] is a locality sensitive hash function for cosine similarity.

To understand simhash, it is first helpful to consider the following randomized process: Imagine that we have two vectors  $a$  and  $b$  on the unit hypersphere in the Euclidean space  $\mathcal{R}^d$  with angle  $\theta$  between them, and we want to produce an estimate of  $\cos(\theta)$ . Select a random  $d$ -dimensional vector  $u$  by sampling each of its coordinates independently from  $N(0, 1)$ . Let the random variable  $X$  have value 1 if  $a$  and  $b$  are on different sides of the hyperplane orthogonal to  $u$ , and 0 otherwise. Then  $X$  is a Bernoulli random variable with expectation:

$$\mathbb{E}[X] = 1 - \mathbb{P}(\text{sign}(a \cdot u) = \text{sign}(b \cdot u)) \quad (2)$$

$$= 1 - \frac{\theta}{\pi} \quad (3)$$

Let  $X_1, \dots, X_n$  be the result of independently repeating this process several times, and set  $\bar{X}_n = \frac{1}{n} \sum_{i=1}^n X_i$ . Then  $\mathbb{E}[\bar{X}_n] = 1 - \frac{\theta}{\pi}$ , and hence  $\mathbb{E}[\pi(1 - \bar{X}_n)] = \theta$ , so that we can use<sup>1</sup>  $\cos(\pi(1 - \bar{X}_n))$  as an estimator of  $\cos(\theta)$ .

The idea behind simhash is to come up with a hash representation which lets us reproduce this randomized estimator: to construct a function  $H$  which produces  $n$ -bit hashes, we first randomly sample  $n$  vectors  $u_1, \dots, u_n$  as above. Then, given a vector  $a$ , the hash  $H(a)$  is the length  $n$  bit sequence in which the  $i$ th bit is 1 if the sign of  $a \cdot u_i$  is positive and 0

---

1. Charikar [Charikar, 2002] notes that for some applications,  $1 - \frac{\theta}{\pi}$  may be a good enough approximation of  $\cos(\theta)$ , so that one can use  $\bar{X}_n$  directly as an estimator of  $\cos(\theta)$ .

otherwise. Now, from two hashed representations  $H(a)$  and  $H(b)$ , if the  $i$ th bit in the two hashes disagree, this is equivalent to  $X_i$  in the randomized process above being equal to 1. Thus, by counting the number of positions where the hashes are distinct and dividing by  $n$ , we get  $\bar{X}_n$ , thereby producing an estimate of  $\cos(\theta)$ .

Rather than constructing the hash function  $H$  by sampling the  $u_1, \dots, u_n$  vectors from the  $d$ -dimensional unit sphere uniformly, a common optimization is to instead sample them from  $\{-1, 1\}^d$ . This has two advantages. First, it is faster to compute the dot product since no floating point multiplication is involved. Second, rather than having to explicitly sample and store each  $u_i$  as a vector, we can replace each  $u_i$  by a 1-bit feature hash function  $h_i$ : the “value” of the vector represented by  $h_i$  is 1 at coordinate  $j$  if  $h_i(j) = 1$  and is  $-1$  if  $h_i(j) = 0$ . We write  $a \cdot h_i$  for the dot product of  $a$  with the vector corresponding to  $h_i$ .

By restricting only to test vectors with coordinates of the form 1 and  $-1$ , the corresponding expected value of  $\pi(1 - \bar{X}_n)$  is no longer exactly  $\theta$  (see [Leskovec et al., 2014, Section 3.7.3] for an example), but for high-dimensional spaces, this approximation is known to be effective in practice [Henzinger, 2006].

**Homomorphic simhash** If we want to use simhash as a representation of feature vectors for entities in coreference resolution, then we need to be able to update the simhash representation as entities are merged and split. In particular, if we join nodes with feature vectors  $a$  and  $b$ , then the vector of their new parent will be  $a + b$ . However, if we only store  $H(a)$  and  $H(b)$ , rather than the vectors  $a$  and  $b$  themselves, we cannot compute  $H(a + b)$ : the  $i$ th bit of  $H(a)$  and  $H(b)$  just records the sign of  $a \cdot h_i$  and  $b \cdot h_i$ , and if these are different, we do not know what the sign of  $(a + b) \cdot h_i$  should be. A similar problem occurs when we split a child with vector  $b$  from a parent with vector  $a$ , since the updated parent’s hash should be  $H(a - b)$ .

Our solution is instead to store the actual dot product of  $a \cdot h_i$  in the hash of  $a$ , rather than just the sign. That is,  $H(a)$  is now an array of length  $n$  instead of an  $n$ -bit sequence. And since

$$(a + b) \cdot h_i = a \cdot h_i + b \cdot h_i \quad \text{and} \quad (a - b) \cdot h_i = a \cdot h_i - b \cdot h_i$$

we can compute  $H(a + b)$  by adding component-wise the arrays for  $H(a)$  and  $H(b)$ , and similarly for  $H(a - b)$ . Finally, we can still efficiently compute the cosine distance between two vectors  $a$  and  $b$  by examining the signs of the entries of  $H(a)$  and  $H(b)$ . We call this representation *homomorphic* because  $H$  is a homomorphism with respect to the additive group structure on vectors.

Of course, storing each dot product instead of just the signs increases the size of our hashes. However, they are still small compared to the feature vectors and, more importantly, their sizes are fixed. We can also store both the dot product and the signs as a bit vector, making sure to update the sign vector after each operation based on the dot product. By storing the sign vector separately we can quickly count the signs in common between two vectors using bitwise operations.

**Statistical Properties** Recall that since  $\mathbb{E}[\pi(1 - \bar{X}_n)] = \theta$ , we can derive a plausible estimate of  $\cos(\theta)$  from  $\bar{X}_n$ . In particular, let  $g(x) = \cos(\pi(1 - x))$  and consider the estimator  $C_n = g(\bar{X}_n)$ . We now describe some statistical properties of  $C_n$ . Our emphasis here is somewhat different from the standard analyses found in related work. The reason is that

LSHs like simhash are most commonly used for duplicate detection [Henzinger, 2006] and for approximate nearest neighbor search [Leskovec et al., 2014], which is quite different from our use case. In those settings, one wants to show that if two items  $x$  and  $y$  are very similar, then the distance estimated from  $h(x)$  and  $h(y)$  will very likely be quite small, and conversely if  $x$  and  $y$  are very different, then their estimated distances will be large. In such cases, the linear approximation to cosine  $\bar{X}_n$  is sufficient. However, since we want to use the cosine distance estimates  $C_n$  as part of the potential function in our CRF, we are interested in additional statistical properties of the estimator.

**Lemma 3.1.**  $C_n$  is consistent. In particular,  $C_n \xrightarrow{\text{a.s.}} \cos(\theta)$ .

*Proof.* By the strong law of large numbers, we have that  $\bar{X}_n \xrightarrow{\text{a.s.}} 1 - \frac{\theta}{\pi}$ . Since  $g$  is continuous, by the continuous mapping theorem [Van Der Vaart, 1998],

$$g(\bar{X}_n) \xrightarrow{\text{a.s.}} g(1 - \frac{\theta}{\pi}) = \cos(\theta)$$

□

**Lemma 3.2.**  $\mathbb{E}[C_n] = \cos(\theta) + E_n$ , where  $|E_n| \leq \frac{\pi^2}{8n}$

*Proof Sketch.* Set  $\mu = \mathbb{E}[\bar{X}_n] = 1 - \frac{\theta}{\pi}$ . The first degree Taylor series for  $g(x)$  about  $\mu$  is:

$$g(x) = \cos(\theta) + \pi \sin(\theta)(x - \mu) + R(x)$$

where  $R$  is the remainder term. We have then that:

$$\mathbb{E}[g(\bar{X}_n)] = \cos(\theta) + \pi \sin(\theta)(\mathbb{E}[\bar{X}_n] - \mu) + \mathbb{E}[R(\bar{X}_n)] \quad (4)$$

$$= \cos(\theta) + \mathbb{E}[R(\bar{X}_n)] \quad (5)$$

Thus it suffices to bound  $|\mathbb{E}[R(\bar{X}_n)]|$ , which can be done using Lagrange's remainder formula (see appendix). □

**Lemma 3.3.**  $\mathbb{V}[C_n] = \frac{\pi^2 \sin^2(\theta)}{n} \cdot \frac{\theta}{\pi}(1 - \frac{\theta}{\pi}) + O(n^{-3/2})$

*Proof Sketch.* For intuition, note that the Taylor series above for  $g$  shows us that  $g(x) \approx \cos(\theta) + \pi \sin(\theta)(x - \mu)$ . So, recalling that  $\mathbb{V}[C_n] = \mathbb{V}[g(\bar{X}_n)] = \mathbb{E}[g(\bar{X}_n)^2] - \mathbb{E}[g(\bar{X}_n)]^2$ , and plugging in the approximation we get:

$$\mathbb{V}[g(\bar{X}_n)] \approx \mathbb{E}[(\cos(\theta) + \pi \sin(\theta)(\bar{X}_n - \mu))^2] - \cos(\theta)^2 \quad (6)$$

$$= 2\pi \sin(\theta) \mathbb{E}[\bar{X}_n - \mu] + \pi^2 \sin^2(\theta) \mathbb{E}[(\bar{X}_n - \mu)^2] \quad (7)$$

$$= \pi^2 \sin^2(\theta) \mathbb{E}[(\bar{X}_n - \mu)^2] \quad (8)$$

To obtain the actual error bound, we carry out the same process but without dropping  $R(x)$  from the Taylor approximation for  $g$ , and then once again use Lagrange's remainder formula to bound the remainder (see appendix). □

Finally, since  $C_n$  is equal to a Lipschitz continuous function of  $n$  independent Bernoulli random variables, we can use the the method of bounded differences [Dubhashi and Panconesi, 2009, Corollary 5.2], to derive the following Chernoff-like tail bound:

**Lemma 3.4.**  $\mathbb{P}(|C_n - \cos(\theta)| > \delta + \frac{\pi^2}{8n}) \leq 2e^{-2n\delta^2/\pi^2}$ .

### 3.2 Angle Preserving Random Projections for Cosine

The statistics underlying simhash, whether the hyperplanes are drawn from Gaussians or the  $d$ -dimensional hypercube (i.e., Rademacher distributions), are actually random projections for which the Johnson-Lindenstrauss lemma applies [Johnson and Lindenstrauss, 1982, Indyk and Motwani, 1998, Achlioptas, 2003]. The lemma states that any set of  $m$  points in  $d$ -dimensional Euclidean space can be embedded into  $n$ -dimensional Euclidean space, where  $n$  is logarithmic in  $m$  and independent of  $d$ ,  $n = O(\epsilon^{-2} \log m)$ , such that all pairwise distances are maintained within an arbitrarily small factor  $1 \pm \epsilon$ , where  $0 < \epsilon < 1$ . Since the projections are linear, they are homomorphic with respect to addition and subtraction. Moreover, previous work shows that the norm-preserving property of a linear random projection  $A$  implies an angle-preserving property [Magen, 2002]; that is, for vectors  $u, v$ , let  $\theta = \triangle(v, u)$  and  $\hat{\theta} = \triangle(Av, Au)$ . The result is the following:

$$\theta - \hat{\theta} \leq 2\sqrt{\epsilon} \text{ and } \frac{1}{1 + \epsilon'} \leq \hat{\theta}/\theta \leq 1 + \epsilon' \quad (9)$$

where  $\epsilon \leq \frac{1}{3}$ ,  $\epsilon' = \frac{8}{\pi}\sqrt{\epsilon}$  and  $n \geq 60\epsilon^{-2} \log m$ . Therefore, we are justified in using the statistics underlying simhash to directly estimate the cosine similarity; viz.,  $\cos \theta \approx \cos \hat{\theta}$ . More precisely, using a Taylor expansion,  $|\cos \theta - \cos \hat{\theta}| \leq |2\sqrt{\epsilon} \sin \hat{\theta} + \frac{(2\sqrt{\epsilon})^2}{2} \cos \hat{\theta}| \leq 2\sqrt{\epsilon}(1 + \epsilon)$ . Although we lose the bit-representation that supports the fast bit-operations that makes simhash so efficient in applications such as webpage deduplication that employ the linear estimator, we potentially gain an improvement in the quality of the cosine estimate. Certainly, the estimate will be smoother since simhash essentially quantizes the angles into a finite number of possibilities equal to the number of bits. Since the same representation supports both types of estimates, we could even alternate between the estimation strategies as dictated by the particular situation: if a large number of similarity comparisons are performed for each entity then simhash makes more sense, otherwise a direct computation of cosine makes more sense. Regardless, as we will later see, both schemes provide sufficiently fast and accurate cosine estimates for coreference resolution.

### 3.3 Homomorphic Minhash for Jaccard Similarity

Minhash [Broder, 1997a, Broder et al., 2000] is a locality-sensitive hash function for Jaccard similarity, defined for binary vectors (encoding sets). The Jaccard similarity between two sets  $S_1, S_2 \in \Omega$  is  $J = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$ . Minhash applies a random permutation (which in practice is accomplished using a random hash function, such as seeded 64-bit murmur hash)  $\pi : \Omega \rightarrow \Omega$ , on a given set  $S \subset \Omega$  and then extracts the minimum value  $h_\pi(S) = \min(\pi(S))$ . The probability that the minhash function for a random permutation of a set computes the same value for two sets is equal to the Jaccard similarity of the two sets [Rajaraman and Ullman, 2010]. In practice multiple ( $n$ ) hash functions are employed to reduce the variance, resulting in a vector  $v^S = \langle h_{\pi_0}(S), \dots, h_{\pi_{n-1}}(S) \rangle$  of  $n$  minimum values, one per hash function.

There are three methods we need to design for homomorphic minhash: **union**, **difference** and **score** (to compute the Jaccard estimate). However, unlike simhash for cosine similarity, the operations underlying the statistics for minhash are non-linear due to the elementwise minimum operation that produces the underlying vector of hash values. Moreover, the set semantics on which the minhash **score** method relies are problematic because the **union**

and **difference** methods need to maintain the invariance that a parent set is equal to the union of a collection of child sets: when we perform a difference operation between a parent set and a child set we cannot simply remove all of the child’s elements from the parent since a sibling might also (redundantly) contribute some of those elements to the parent. We sketch a solution to these problems and include details in Appendix E. However, we note that our solution is not complete because there are several corner cases we do not yet handle. Nevertheless, we find that the representation works well in practice.

First, we handle the problem of set-semantics by augmenting the  $n$ -dimensional minhash representation  $v^S$  of each set  $S$  with an  $n$ -dimensional vector of counts  $c^S$ , in which each dimension corresponds to a minimum hash value (essentially embracing multiset semantics, but in the hash space). The count indicates the number of child sets that contribute the associated hash value. For the union of two sets  $S = S_1 \cup S_2$ , we either keep the count associated with the smaller hash value if they are different (that is, since we set  $v^S = v^{S^*}$  we set  $c_i^S = c_i^{S^*}$  where  $S^* = \operatorname{argmin}_{S_j \in \{S_1, S_2\}} (v_i^{S_j})$ ), or sum the counts if they are the same (that is,  $c_i^S = c_i^{S_1} + c_i^{S_2}$ ). For difference we employ the same strategy except we subtract the counts instead of add them. The counts are appropriately ignored when estimating the Jaccard since we want the estimate to reflect sets rather than multisets.

Second, we must also address the fact that the minimum is a non-linear operator. The problem arises when the count associated with a hash value becomes zero. Then, how do we recompute what the new minimum value should be? Recomputing the minimum from scratch is a nonstarter because it would require keeping around the original sparse vector representations that we are trying to supplant. Rewriting the difference in terms of unions is also computationally expensive since it would require traversing the entire tree to check what the new minimum value should be. Instead, noting that minhash typically has hundreds of hash functions (e.g.,  $n = 256$ ) for each set, we propose to ignore the hash values associated with zero counts, and employ the remaining hashes to compute the Jaccard. This strategy has consequences for both bias and variance. First, since fewer hashes are employed, the variance increases, and if left unchecked may culminate in a worst case in which all counts are zero and Jaccard can no longer be estimated. However, we can periodically refresh the counts by traversing the trees and hopefully we do not have to do such refresh operations too often in practice. Second, since the hashes associated with zero counts are correlated, the estimate is no longer unbiased. Therefore, as described in Appendix E, we modify the Jaccard estimate to make better use of the zero-counts rather than simply ignore them, and this eliminates the bias in some cases. However, we do not have a solution that works in every case.

Finally, there is also the question of how to perform union and difference for cases in which the count is zero. For now, we ignore the zero counts during these operations, but are currently exploring strategies for incorporating them to further reduce bias and variance.

### 3.4 Additional Compression Schemes

Hierarchical coref involves other features, such as entropy and complexity-based penalties on the bag of words representations of context and topics associated with the entities. Although not a focus of this current study, we note that some of these representations depend on the ratios of  $p$ -norms that can be estimated with Johnson-Lindenstrauss style representations.

Moreover, there exist hashing schemes to enable fast estimation of entropy. We save the homomorphic versions of these hashes for future work.

## 4. Experiments

In this section we empirically study the two cosine-preserving homomorphic compression schemes, simhash and its related random projection, on a conditional random field (CRF) model of author coreference resolution, the problem introduced in Section 2. First we study simhash and we hypothesize that the this representation will substantially improve the speed of inference for the reasons outlined earlier, but it is less clear how the simhash representation will affect the quality of the model. On the one hand, our theoretical analysis shows that the variance and bias reduce rapidly with the number of bits, but on the other hand, the sheer number of vector comparisons that take place during inference is substantial, making it likely for errors to occur anyway. With this in mind, we study how simhash affects the model quality in our first set of experiments. In our second set of experiments, we study the extent to which it improves inference speed. In a third set of experiments, we compare simhash with the random projection method. Finally, we present initial results for homomorphic minhash to empirically assess how it compares to exact Jaccard, which is important given the increased bias and variance of our method.

**Data** We employ the REXA author coreference dataset,<sup>2</sup> which comprises 1404 author mentions of 289 authors with ambiguous first-initial last-name combinations: *D. Allen, A. Blum, S. Jones, H. Robinson, S. Young, L. Lee, J. McGuire, A. Moore*. We split the data such that training set contains mentions of the first five ambiguous names (950 mentions) while the testing set comprises the remaining three ambiguous names (454 mentions). The dataset is highly ambiguous since there are multiple entities with each name. In addition, we also employ the DBLP dataset which contains over one million paper citations from which we extract about five million unlabeled author mentions [Ley, 2002].

**Model** We investigate homomorphic simhash in the context of the hierarchical coreference model presented in Section 2. We employ two types of feature variables, a “name bag” that represents the features of the author’s name and a “context bag” that represents the remaining features in the citation from which the author mention is extracted (co-authors, title, venue, topics, keywords). For more details about the features please see Appendix B.

We employ the implementation of hierarchical coreference available in the FACTORIE toolkit, using FACTORIE’s implementation of the variables, the model and the inference algorithm [McCallum et al., 2008]. We additionally implement the simhash variables and potential functions inside this framework. We employ FACTORIE’s default inference algorithm for hierarchical coreference which is essentially a greedy variant of multi-try Metropolis-Hastings in which the proposals make modifications to the sub-trees (e.g., move a subtree from one entity to another, or merge two trees under a common root node). More details are in previous work, and the implementation is publicly available in FACTORIE [McCallum et al., 2009, Wick et al., 2012]. We estimate the parameters with hyper-parameter search on the training-set (Appendix B).

---

2. <http://www.iesl.cs.umass.edu/datasets.html>

## HOMOMORPHIC HASHING

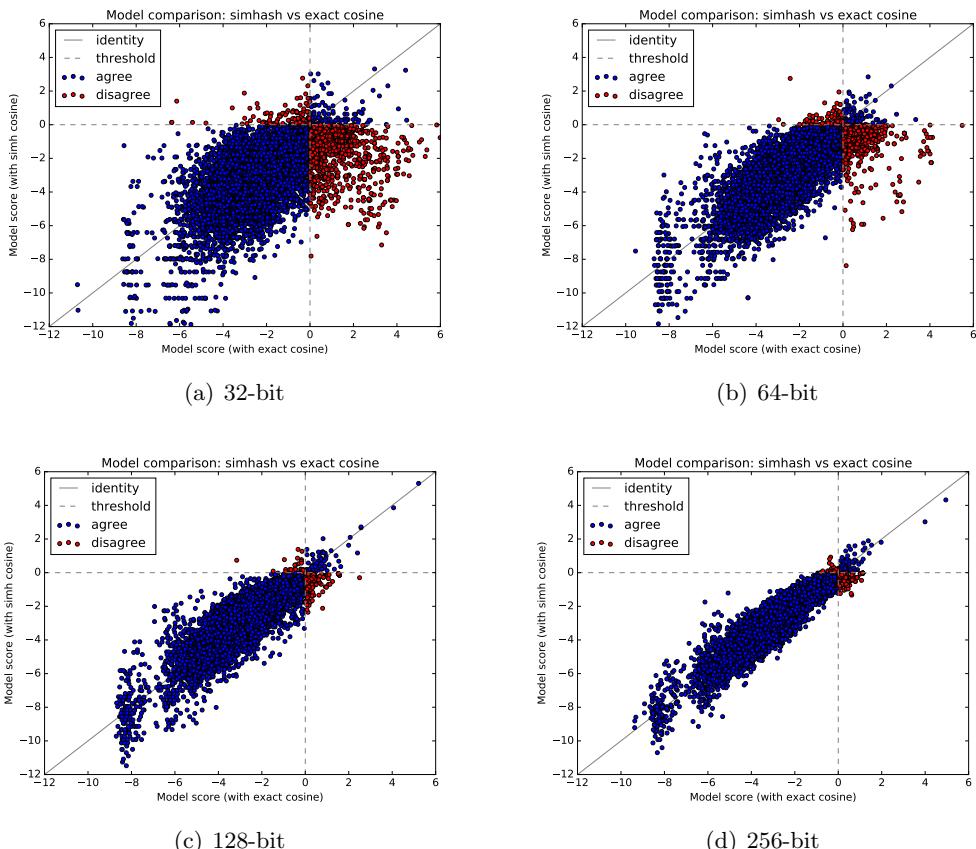


Figure 1: Model score comparison with homomorphic simhash and exact sparse-vector representations. The closer points are to the identity reference line, the better. The dotted horizontal and vertical lines represent the decision threshold for inference. Points for which the two models agree are in blue, the agreements rates are: 88.6, 83.6, 97.0, 97.8 percent (respectively 32, 64, 128, 256 bits).

**Experiment 1: Simhash Estimation Quality** Here we study the quality of models with simhash representations by comparing them to models with exact representations. First, we compare how these models evaluate and score the intermediate results of MCMC inference. We run MCMC for 100,000 steps to optimize simhash-based models on the REXA test set (with either 256, 128, 64 and 32 bit hashes). The chains begin with the singleton configuration (all mentions in their own entity tree of size one), and at each step proposes changes to the current state which the model decides to either accept or reject. This process gradually produces larger and larger trees. For each proposed state change (sample), we record the log model ratio of both the simhash model and the exact model. We present every 10th sample in a scatter plot in Figure 1. The closer the points are to the identity reference line  $y = x$ , the more accurate the simhash model is for those points. Varying the number of bits has a pronounced effect on the model’s ability to judge MCMC states.

For each step, we also record if the simhash model and exact model agree upon whether to accept the stochastic proposal (blue points) or not (red points).<sup>3</sup> The agreement rates are 97.8, 97.0, 93.6, 88.6 percent (respectively 256, 128, 64, 32 bits). We also plot the decision boundaries for this agreement as dotted lines. The upper-left and lower-right quadrants contain all the points for which the two models disagree, while the other two quadrants contain points for which they agree. In particular, the upper-right quadrants contain the points that both the simhash model and exact model believe should be accepted (true positives), while the lower-right quadrants contain the points that both models think should be rejected (true negatives). Most points lie in this quadrant since the chance of proposing a fruitful move is relatively low. Visually, there appears to be a qualitative gap between 64 bits and 128 bits on this data, leading to a recommendation of using at least 128 bits.

We also compare the models in terms of their final coreference performance (according to B-cubed F1 [[Bagga and Baldwin, 1998](#)]). The exact model achieves an F1 of 78.6, while the simhash variants achieve F1 scores of 77.6, 75.6, 62.8, 55.6 for 256, 128, 64, 32 bits respectively. For more detailed results, with precision, recall and other types of F1, see Table 1 in the appendix. Overall, the accuracy of the 128 and 256-bit models are reasonable with 256 being competitive with the performance of the exact model. When using fewer bits, again, the performance decreases precipitously.

**Experiment 2: Simhash Speed** In this experiment we study the extent to which the compressed simhash representation improves inference speed. As described before, we tune the models on the REXA training set. Then, to test the models on a larger dataset, we supplement the 454 labeled REXA test mentions with five million unlabeled mentions from DBLP. We run each model on this combined dataset, initializing to the singleton configuration and then running one billion samples. We record coreference quality on the labeled subset every 10,000 samples and plot how it changes over time in Figures 2(a),2(b).

Although Experiment 1 shows that the simhash models are slightly worse in terms of their final F1 accuracy on the REXA test set, we see a striking computational advantage for simhash. For each F1 value, we compute how much faster the simhash model achieves that value than the exact model and plot this speed-improvement as a function of F1-level in Figures 2(c),2(d). As we can see, the speed improvement is more than a base-ten order of magnitude for most accuracy levels, and the speed difference increases with accuracy. This

---

3. These decisions, and hence agreement upon them, are deterministic with our 0-temperature sampler.

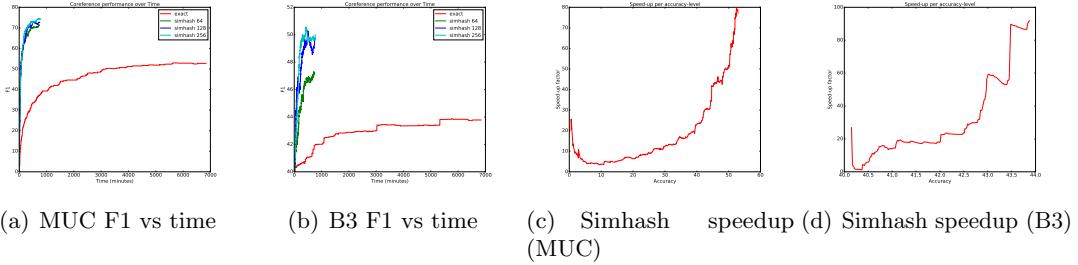


Figure 2: Comparison of hierarchical coreference models with either simhash or exact sparse-vector representations of the features. Simhash representations result in large speedups and have little affect on accuracy.

is because the exact representation slows down over time as the number of features in the representation grows during inference. Indeed if we look at the sampling rate over time for each model, we find that the simhash models run at about 20,000-25,000 samples per second the entire time, while the model with the exact representation starts at a rate of 3000-4000 samples per second, but then drops to under 1000 samples per second as the size of the entities get larger. This raises the question: can we improve the speed of the exact model by reducing the number of features? We address this question in Appendix C.2, but summarize here briefly: it is possible to improve the speed and reduce the gap, but simhash is still faster and there is a trade-off with coreference quality. In addition, whether feature ablation meaningfully improves performance depends on particular aspects of the data set, whereas the simhash representation can be applied generically.

**Experiment 3: JL Random Projections** In this experiment, we compare simhash with an approach that directly computes cosine on the statistics that underly the bit representation. As argued previously in Section 3.2, this approach is justified because the statistics are random projections that are homomorphic and cosine-preserving. Intuitively, we expect this approach to work even better because simhash further compresses these statistics into single bits, whereas in the random projection approach, we compute cosine directly on the real-valued statistics. However, our current theoretical analysis does not allow us to compare one to another; therefore, we turn to the empirical studies in this section.

We perform an experiment analogous to Experiment 1, except we employ the exact model to make decisions about what MCMC proposals to accept. This allows us to compare both approximation schemes on the same set of samples. For each accepted sample, we again ask the question how the approximate methods, simhash and JL, would have judged them. We find that JL does indeed perform better than simhash for the same number of bits.<sup>4</sup> In particular, the Spearman’s rho for JL increases from 93.2 to 97.2 over simhash when employing 256 bits (dimensions). For all but one case (128 bits), we find a similar improvement. Moreover, the coreference accuracy also increases in each case; for example,

4. Dimension might be a better term for JL since the floats are never converted into bits.

from 77.6 B3 F1 to 78.3. More detailed results are in Table 3 in Appendix D, along with the associated scatter plots (Figure 6).

## 5. Experiment 4: MinHash

We also investigate our homomorphic MinHash representation for Jaccard similarity. For lack of space, and because Jaccard is not employed by the hierarchical coreference, we relegate most of the evaluation to Appendix E.2. To briefly summarize, we find that 128 and 256 hash functions are sufficient for reasonable estimates and that the representation rarely needs to be refreshed to ensure that all the counts are above zero, enabling the computational efficiency we desire. However, there is room for improvement because only 16% of the hash functions on average have a non-zero entry after 100,000 samples. Thus, after 100,000 samples, a 256 hash version will begin to behave like a 40 hash version.

## 6. Related Work

Homomorphic representations have many possible uses in machine learning; for example, *homomorphic encryption* allows models to run directly on *encrypted* data for cases in which privacy is a concern [Graepel et al., 2012, Dowlin et al., 206]. Instead, our method is similar to *homomorphic compression* [McGregor, 2013] which allows our model to run directly on *compressed* data for cases in which computational efficiency is a concern. Our approach is based on simhash, a locality-sensitive hash function. Locality sensitive hashes such as simhash and minhash are useful in large scale streaming settings; for example, to detect duplicate webpages for web-crawlers or in nearest neighbor search [Broder, 1997b, Manku et al., 2007, Leskovec et al., 2014]. They are sometimes employed in search and machine-learning applications including coreference for “blocking” the data in order to reduce the search space [Getoor and Machanavajjhala, 2012]. Note that this application of locality sensitive hash functions for coreference is complementary to ours, and does not require it to be homomorphic. Other hashing and sketching algorithms are also employed as a strategy to compress the sufficient statistics in Bayesian models so that they can scale better as the number of parameters increase with the dataset. For example, feature hashing, count-min sketches and approximate counters have been employed in scaling latent Dirichlet categorical models such as LDA by compressing, for example, the counts that assign latent variables to mixture components [Zaheer et al., 2016, Tassarotti et al., 2018].

Our paper focuses on three homomorphic compression schemes including one for minhash. Our solution of furnishing the minhash values with counts resembles a strategy that similarly employs counts in a  $k$  minimum values (KMV) sketch for estimating the number of distinct values in a set [Beyer et al., 2007, 2009]. A key difference is that that work develops an unbiased estimator for the number of *distinct values* that explicitly involves the counts as part of the estimate itself, whereas we develop a biased estimator that directly estimates *Jaccard similarity* and that employs the counts to bound our knowledge about possible minimum hash values when they vanish during difference operations.

Our schemes are related to random projections, which are commonly used to improve computational efficiency in machine learning. Examples include feature-hashing [Weinberger et al., 2009], fast matrix decomposition [Halko et al., 2009] and fast kernel computations

[Rahimi and Recht, 2007]. However, while some of these random projections happen to be homomorphic, this property is often not exploited. The reason is that they are typically used to compress static objects that remain fixed, such as the Gram matrix for kernel methods. In contrast, our setting requires compressing dynamic sets that change during inference.

Word embeddings [Mikolov et al., 2013] also provide low-dimensional dense representations that are useful for capturing context, but in practice, these embeddings are too smooth to be useful as the sole representation for disambiguating the names of peoples, places or organizations, as is necessary in coreference (e.g., the names “Brady” and “Belichick” would be highly similar according to a word-to-vec style embedding, even though they are unlikely to be coreferent). Deep learning might allow for more suitable representations to be learnt and its application to coreference is promising. However, the current research focus is on within-document noun-phrase coreference and entity linking, while emphasizing improvements in accuracy rather than inference speed and scalability [Globerson et al., 2016, Wiseman et al., 2016, Lee et al., 2017, Raiman and Raiman, 2018]. Combining deep learning and conditional random fields for hierarchical coreference remains an open problem.

Finally, in addition to the work mentioned throughout the paper, there is an abundance of related work on coreference resolution including noun-phrase coreference, cross-document coreference, record linking, entity linking and author coreference. While not possible to cover the breadth of the space here, we refer the reader to a few useful surveys and tutorials on the subject [Ng, 2010, Getoor and Machanavajjhala, 2012]. More recently, as mentioned in the foregoing paragraph, deep learning approaches are beginning to show promise. There has also been promising recent work on scalable hierarchical clustering known as PERCH [Kobren et al., 2017]. Since PERCH employs Euclidean distance rather than cosine similarity, it currently falls outside the purview of our study. However, the Johnson-Lindenstrauss applies to Euclidean distance making random projections a possibility for PERCH.

## 7. Conclusion

In this paper we presented several homomorphic compression schemes for representing the sparse, high dimensional features in a graphical model for coreference resolution, even as these features change during inference. Our primary concern was cosine similarity for which we investigated simhash and angle-preserving random projections. We also proposed a homomorphic version of minhash for Jaccard similarity. In the particular case of simhash, we presented a new variant of the original estimator and analyzed its statistical properties — including variance, bias and consistency — to help justify its use in a probabilistic model. We found that both simhash and angle-preserving random projections were sufficiently accurate, given enough dimensions, to represent features for coreference, and that the random projections produce slightly better estimates. Moreover, these representations were an order of magnitude faster than conventional sparse data structures, laying the foundation for greater scalability.

## References

- Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66:671–687, 2003.
- Amit Bagga and Breck Baldwin. Algorithms for scoring coreference chains. In *Conference on Language Resources and Evaluation Workshop on Linguistics Coreference*, 1998.
- Kevin Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’07, pages 199–210, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8.
- Kevin Beyer, Rainer Gemulla, Peter J. Haas, Berthold Reinwald, and Yannis Sismanis. Distinct-value synopses for multiset operations. *Commun. ACM*, 52(10):87–95, October 2009. ISSN 0001-0782.
- Andrei Z Broder. On the resemblance and containment of documents. In *Compression and complexity of sequences 1997. proceedings*, pages 21–29. IEEE, 1997a.
- Andrei Z. Broder. On the resemblance and containment of documents. In *IEEE SEQUENCES*, 1997b.
- Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.
- Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.
- Aron Culotta, Pallika Kanani, Robert Hall, Michael Wick, and Andrew McCallum. Author disambiguation using error-driven machine learning with a ranking loss function. In *Sixth International Workshop on Information Integration on the Web (IIWeb-07), Vancouver, Canada*, 2007a.
- Aron Culotta, Michael L. Wick, and Andrew McCallum. First-order probabilistic models for coreference resolution. In *HLT-NAACL*, 2007b.
- Nathan Dowlin, ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 206.
- Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. ISBN 978-0-521-88427-3.
- Lise Getoor and Ashwin Machanavajjhala. Entity resolution: Tutorial. In *VLDB Tutorial*, 2012.
- Amir Globerson, Nevena Lazic, Soumen Chakrabarti, Amarnag Subramanya, Michael Ringgaard, and Fernando Pereira. Collective entity resolution with multi-focal attention. In *ACL*, 2016.

Thore Graepel, Kristin Lauter, and Michael Naehrig. MI confidential: Machine learning on encrypted data, 2012.

N. Halko, P.G. Martinsson, and J.A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev*, 53(2), 2009.

Hui Han, Lee Giles, Hongyuan Zha, Cheng Li, and Kostas Tsoutsouliklis. Two supervised learning approaches for name disambiguation in author citations. In *Proceedings of the 4th ACM/IEEE-CS Joint Conference on Digital Libraries, JCDL '04*, pages 296–305, New York, NY, USA, 2004. ACM.

Monika Rauch Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Seattle, Washington, USA, August 6-11, 2006*, pages 284–291, 2006. doi: 10.1145/1148170.1148222. URL <http://doi.acm.org/10.1145/1148170.1148222>.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC '98*, pages 604–613, New York, NY, USA, 1998. ACM.

William B. Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. In *Conference in modern analysis and probability*, pages 189–206. Amer. Math. Soc., 1982.

Ari Kobren, Nicholas Monath, Akshay Krishnamurthy, and Andrew McCallum. A hierarchical algorithm for extreme clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17*, pages 255–264, New York, NY, USA, 2017. ACM.

Kenton Lee, Luheng He, Mike Lewis, and Luke Zettlemoyer. End-to-end neural coreference resolution. In *EMNLP*, 2017.

J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014. ISBN 9781107077232.

Michael Ley. The DBLP computer science bibliography: Evolution, research issues, perspectives. In *String Processing and Information Retrieval, 9th International Symposium, SPIRE 2002, Lisbon, Portugal, September 11-13, 2002, Proceedings*, pages 1–10, 2002.

Avner Magen. Dimensionality reductions that preserve volumes and distance to affine spaces, and their algorithmic applications. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 239–253. Springer, 2002.

Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

Andrew McCallum and Ben Wellner. Toward conditional models of identity uncertainty with application to proper noun coreference. In *Proceedings of the 2003 International*

*Conference on Information Integration on the Web*, IIWEB'03, pages 79–84. AAAI Press, 2003.

Andrew McCallum, Kashayar Rohanemanesh, Michael Wick, Karl Schultz, and Sameer Singh. FACTORIE: efficient probabilistic programming for relational factor graphs via imperative declarations of structure, inference and learning. In *NIPS workshop on Probabilistic Programming*, 2008.

Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. In *Neural Information Processing Systems (NIPS)*, 2009.

Andrew McGregor. Towards a thoery of homomorphic compression. In P. Bonizzoni, V. Brattka, and B. Lowe, editors, *The nature of computational. Logic, algorithms, applications.*, pages 316–319. Springer, 2013.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

Nicolas Monath and Andrew McCallum. University of massachusetts's entry in the uspto patentsview workshop. In *PatentsView Inventor Disambiguation Workshop*, 2016.

H.B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, 1959.

Vincent Ng. Supervised noun phrase coreference research: The first fifteen years. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1396–1411, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nathanael Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher Manning. A multi-pass sieve for coreference resolution. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP '10, pages 492–501, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

Ali Rahimi and Ben Recht. Random features for large-scale kernel machines. In *NIPS*, 2007.

Jonathan Raiman and Olivier Raiman. Deeptype: Multilingual entity linking by neural type system evolution. In *AAAI*, 2018.

A Rajaraman and JD Ullman. Finding similar items. *Mining of Massive Datasets*, 77:73–80, 2010.

Joseph Tassarotti, Jean-Baptiste Tristan, and Michael Wick. Sketching for latent dirichlet-categorical models, 2018.

*PatentsView Inventor Disambiguation Workshop*, 2016. United States Patent and Trademark Office (USPTO). URL <https://www.uspto.gov/about-us/organizational-offices/office-policy-and-international-affairs/patentsview-inventor>.

A.W. Van Der Vaart. *Asymptotic Statistics*. Cambridge Series in Statistical and Probabilistic Mathematics, 3. Cambridge University Press, 1998. ISBN 9780521496032. URL <https://books.google.com/books?id=udhfQgAACAAJ>.

Kilian Weinberger, Anibarn Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *ICML*, 2009.

Michael Wick, Sameer Singh, and Andrew McCallum. A discriminative hierarchical model for fast coreference at scale. In *ACL*, 2012.

Sam Wiseman, Alexander M. Rush, and Stuart M. Shieber. Learning global features for coreference resolution. In *NAACL-HLT*, 2016.

Manzil Zaheer, Michael L. Wick, Jean-Baptiste Tristan, Alex Smola, and Guy L. Steele Jr. Exponential stochastic cellular automata for massively parallel inference. In *ICML*, 2016.

# Online Post-Processing In Rankings For Fair Utility Maximization

Ananya Gupta\*  
UMass Amherst

Ari Kobren  
Oracle Labs

Eric Johnson\*  
UMass Amherst

Swetasudha Panda  
Oracle Labs

Justin Payan†  
UMass Amherst

Jean-Baptiste Tristan  
Boston College

Aditya Kumar Roy  
UMass Amherst

Michael Wick  
Oracle Labs

## ABSTRACT

We consider the problem of utility maximization in online ranking applications while also satisfying a pre-defined fairness constraint. We consider batches of items which arrive over time, already ranked using an existing ranking model. We propose online post-processing for re-ranking these batches to enforce adherence to the pre-defined fairness constraint, while maximizing a specific notion of utility. To achieve this goal, we propose two deterministic re-ranking policies. In addition, we learn a re-ranking policy based on a novel variation of learning to search. Extensive experiments on real world and synthetic datasets demonstrate the effectiveness of our proposed policies both in terms of adherence to the fairness constraint and utility maximization. Furthermore, our analysis shows that the performance of the proposed policies depends on the original data distribution w.r.t the fairness constraint and the notion of utility.

### ACM Reference Format:

Ananya Gupta, Eric Johnson, Justin Payan, Aditya Kumar Roy, Ari Kobren, Swetasudha Panda, Jean-Baptiste Tristan, and Michael Wick. 2021. Online Post-Processing In Rankings For Fair Utility Maximization . In *Proceedings of the Fourteenth ACM International Conference on Web Search and Data Mining (WSDM '21), March 8–12, 2021, Virtual Event, Israel*. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3437963.3441724>

## 1 INTRODUCTION

Ranking models are ubiquitous and support high stakes decisions in a variety of application contexts, e.g., online marketing, job search and candidate screening, loan applications, etc. Depending on the application, these models are used to rank products, job candidates, credit profiles, etc. Ultimately, these models facilitate selection of specific items from the ranked list.

Many practical instantiations of ranking applications are online processes where there is an incoming stream of batches of items to be ranked. For example, if we consider a hiring application, a job advertisement elicits applicants which naturally arrive over time.

\*These authors contributed equally to this research.

†Work completed prior to author's internship with Amazon.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*WSDM '21, March 8–12, 2021, Virtual Event, Israel*

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8297-7/21/03... \$15.00

<https://doi.org/10.1145/3437963.3441724>

The hiring entity processes these applications in batches to screen and select candidates for job interviews. Unlike a static ranking application, such an online system necessitates proactive decision making so as to maximize long-term utility. In the context of hiring, this translates to the selection of qualified candidates given an unknown distribution over the batches of future applicants.

The position of an item in a ranking directly influences its visibility or *exposure* [23], thereby directly affecting whether or not the item is eventually selected. Standard techniques for ranking often involve ordering items in descending order of *relevance* (as defined in information retrieval literature [22]). For example, in hiring, relevance can be quantified as the degree to which an applicant's qualifications match the job requirements. These standard techniques are referred to as *utility* maximizing ranking algorithms. However, recent literature has shown that utility maximization can lead to representation disparities in the generated rankings [14, 23], either in the static or online environment.

Recent work in machine learning fairness attempts to alleviate discrimination by enforcing adherence to specific fairness criteria or equivalently, fairness constraints [8, 9]. While a large proportion of this research focuses on supervised classification [27, 28], the idea of fairness in ranking is relatively less explored [3, 6, 23, 29]. In previous work, algorithms have been proposed to satisfy a variety of such fairness constraints (e.g., parity of exposure). However, these algorithms are primarily aimed at static rather than online ranking problems. One technique for ensuring adherence to a fairness constraint in classification is to post-process the decisions from a trained (black-box) classifier [11]. In the offline variant of this technique, a learned classifier is modified offline to generate a derived classifier which is then deployed to make the predictions in real time. One issue with this approach is that adherence to the fairness constraints hold in expectation with respect to the training data distribution. Consequently, this might lead to sporadic violations of the fairness constraints on the test data distribution.

In this work we consider the problem of satisfying fairness constraints while maximizing utility in online ranking applications. We do not advocate for a specific fairness constraint. Instead, we assume that a constraint is assigned prior to deployment, and our goal is to ensure that it is always satisfied. While our approach can incorporate general constraints, in this paper, we define the *demographic disparity* criterion to ensure parity of pairwise *exposures* of the different groups of items over an aggregate of observed ranking batches. Parity constraints are important in ranking applications beyond fairness considerations, e.g., to incorporate diversity in rankings.

We consider an incoming stream of batches of items that need to be ranked for a specific application. An existing ranking model generates a ranking from each batch at a given timestep. The goal is to decide whether (and how) to re-rank the batch in order to maximize cumulative utility while enforcing the fairness criteria. We post-process or re-rank the decisions generated by the initial (fixed) ranking model by deploying a *re-ranking* algorithm that guarantees that the fairness constraints are satisfied. Unlike a static intervention, an online post-processor is better equipped to handle concept drift in the test data distribution. It can address instantaneous fairness constraint violations at any given timestep, thereby satisfying the fairness constraint proactively through the time steps, while maximizing a predefined utility notion.

We begin by proposing two different deterministic policies: Fair Queues and Greedy Fair Swap. Since our framework involves an unknown distribution over ranking batches, we also propose to learn a re-ranking policy via a novel variation of learning to search [7], dubbed *locally optimal learning to search with queues* (L2SQ). L2SQ creates a priority queue for each group in a batch, within which items are ordered according to relevance scores from the initial ranking model. The learned policy creates a re-ranking by repeatedly deciding which queue to pull from until all queues are empty. The learned policy's action space is defined at each position in the re-ranking to be the non-empty queues which, upon being pulled from, can result in a ranking that satisfies the fairness constraint.

In summary, we make the following contributions: a) we present an algorithmic framework for online post-processing of ranking batches according to a given fairness criteria, b) we propose two deterministic policies, and a novel approach to learn a re-ranking policy which maximizes utility while respecting the fairness criteria, and c) we present extensive experimental results on various real world (German Credit, AirBnb, StackExchange and Resume) as well as a synthetic dataset to demonstrate the effectiveness of the proposed approaches in terms of utility and adherence to the fairness constraint. We release our processed versions of the datasets for the online re-ranking setting. In addition, our experiments demonstrate that the performance of the policies depends on the original data distribution w.r.t the given fairness constraint and the notion of utility. For concreteness and based on available datasets, we consider specific instances of ranking applications, but our methodology is generally applicable.

## 2 RELATED WORK

There are several directions of ongoing research at the intersection of algorithmic fairness and rankings. Yang and Stoyanovich [26] propose various logarithmic discounting based measures for fairness and extend the approach on learning fair representations in [31] by redefining the loss function to be appropriate for ranked outputs. Several novel metrics for fairness auditing on rankings are proposed in [15, 21]. There is previous work on mitigating bias in the relevance scores, which are often used for generating ranking models. This includes work on inverse propensity scoring to estimate true relevance scores produced using click data [13] and learning a fair relevance model [30]. Kulshrestha et al. investigate and distinguish between sources of bias arising from the data and the ranking model respectively [16]. Asudeh et al. and Guan et al.

break the ranking score into a linear combination of component scores, with weights on each component given by the user. If the user-provided weights produce an unfair ranking, their algorithm proposes the closest weight vector which produces a fair ranking [1, 10]. There is previous research on learning ranking models which satisfy specific fairness constraints. Beutel et al. introduce a set of novel metrics for fairness auditing in recommendation systems and improve fairness criteria during model training using pairwise regularization [2]. Singh et al. propose a learning to rank approach for utility maximization with fairness constraints [24].

Unlike the above previous work, in our problem setting, an existing ranking model generates a ranking of items (or a sequence of rankings). Our approach performs a re-ranking of the items and constructs a new ranking so as to maximize a given utility notion while satisfying a fairness constraint. Consequently, we do not directly learn a ranking model or intervene during the training process of a ranking model.

One direction of previous research most relevant to our approach is utility maximization in rankings subject to fairness constraints. Singh et al. define the concept of exposure and optimize for fair probabilistic rankings [23]. While their algorithm satisfies the fairness constraints in expectation, the constraints might not hold on the individual rankings sampled from the optimized probability distribution. Zehlike et al. [29] present a statistical test based approach which operates on a series of rankings in an online fashion. However, they consider a specific fairness criterion based on the proportion of certain group members at each position in the ranking. Biega et al. [3] propose the idea of amortized fairness, but the analysis is specifically for individual fairness criteria. Celis et al. [5] propose a constrained maximum weight matching algorithm for utility maximization with a fairness constraint. However, the approach does not consider aggregate of rankings in an online setting. Panda et al. [18] explore audit and control of fairness measures in ranking batches. We present a concrete formulation of the online post-processing problem for ranking batches and analyze deterministic as well as learned policies as solution approaches.

There is significant previous research on diversity in information retrieval [4, 17] where the goal is to not present similar items in the rankings. Stoyanovich et al. study the online, diverse top-k set selection problem which is different from our problem setting [25]. Sakai and Song present a metric for auditing rankings for diversity measures [20]. In contrast, our approach reconstructs rankings to satisfy given diversity criteria. Moreover, unlike the majority of previous work on ranking diversity which are based on similarity of items, our approach operates on a discrete set of items as groups.

## 3 PROBLEM DEFINITION

In this section we describe the problem setting of online fair utility maximization in ranked batches. We begin with a brief review of standard definitions from information retrieval. Consider a *batch* of  $n$  items  $i \in 1, 2, \dots, n$ . Let  $r(i)$  denote the rank of item  $i$  in ranking  $r$ . The exposure [23] of  $i$  under ranking  $r$  is defined as the following

$$\text{Exposure}(i|r) = \frac{1}{\log(r(i) + 1)} \quad (1)$$

Let  $q(i)$  be the relevance of item  $i$ . The discounted cumulative gain (DCG) of a ranking  $r$  is defined as  $DCG(r) = \sum_{i=1}^n \frac{2^{q(i)} - 1}{\log(r(i)+1)}$  and the normalized DCG (nDCG) of  $r$  as  $\frac{DCG(r)}{DCG(r^\star)}$ , where  $r^\star$  ranks items in decreasing relevance order.

We assume that each item  $i$  in the batch is a member of a pre-defined group,  $g(i)$ . In this work, as a concrete example of a fairness constraint, we aim to generate rankings so as to equalize exposure across groups [23, 30]. Specifically, let  $G_j = \{i \in [n] \mid g(i) = j\}$  be a group of items. Then the exposure of  $G_j$  in ranking  $r$  is given by  $Exposure(G_j|r) = \sum_{i \in G_j} Exposure(i|r)$ .

We denote our fairness constraint as the *demographic disparity* (DDP) constraint that bounds the difference in mean exposures between all pairs of groups. We define the DDP of a ranking  $r$  as

$$DDP(r) = \max_{\{G_j, G_{j'}\}} \frac{Exposure(G_j|r)}{|G_j|} - \frac{Exposure(G_{j'}|r)}{|G_{j'}|}. \quad (2)$$

Our fairness constraint ensures that the DDP is less than a predetermined threshold  $\alpha$ . Note that DDP is analogous to the demographic parity constraint in classification. It also relaxes a previously proposed demographic parity constraint on rankings [23]. Although we focus on DDP here, our approach can be adapted to account for any general fairness constraint.

### 3.1 Online Post-Processing For Rankings

Unlike previous work, we focus on the online setting where the batches of items arrive over time. Specifically, at each timestep  $t \in \{1, \dots, T\}$  we receive a batch of items initially ranked by a fixed ranking model in descending score order (ranking denoted  $r_{\text{init}}^{(t)}$ ).

We will define a post-processing policy  $\pi$  to re-rank the items in each batch according to a new ranking  $r^{(t)}$  containing group populations  $G_j^{(t)}$ , such that nDCG is maximized and the fairness constraint is satisfied. However, in this setting, the nDCG and the fairness constraint apply in aggregate over batches. In particular, at a given timestep  $t$ , we define the nDCG at  $t$  for some sequence of rankings  $R = \{r^{(1)}, \dots, r^{(t)}\}$ ,  $nDCG(R, t)$ , as

$$\frac{1}{t} \sum_{s=1}^t nDCG(r^{(s)}) \quad (3)$$

The DDP at  $t$ ,  $DDP(R, t)$ , is

$$\max_{\{G_j, G_{j'}\}} \frac{\sum_{s=1}^t Exposure(G_j^{(s)}|r^{(s)})}{\sum_{s=1}^t |G_j^{(s)}|} - \frac{\sum_{s=1}^t Exposure(G_{j'}^{(s)}|r^{(s)})}{\sum_{s=1}^t |G_{j'}^{(s)}|} \quad (4)$$

$\pi$  uses all rankings  $\{r^{(s)}\}_{s=1}^{t-1} \cup \{r_{\text{init}}^{(t)}\}$  to compute the utility and the constraints in aggregate. However, it can only re-rank the current batch and not any of the previous batches. By re-ranking the current batch, our post-processing policy aims to satisfy the fairness constraint (in aggregate) while maximizing cumulative utility over the batches observed so far. Consequently, our goal in online post-processing is given by

$$\begin{aligned} & \text{maximize } nDCG(R, T) \\ & \text{subject to } \max_{1 \leq t \leq T} DDP(R, t) \leq \alpha \end{aligned} \quad (5)$$

Henceforth, we use the term fair ranking to denote an aggregate of ranking batches up to a given (current) time step which satisfy our DDP fairness constraint (and the term unfair ranking otherwise).

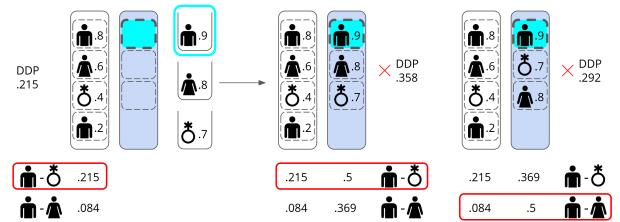
## 4 DETERMINISTIC POLICIES

A deterministic policy re-ranks a batch at a given time-step so as to obtain a solution to the objective in Equation 5. In this section, we describe two deterministic re-ranking policies based on different heuristics: Fair Queues and Greedy Fair Swap.

### 4.1 Fair Queues

We begin by modifying the FA\*IR algorithm proposed in [29]. The original FA\*IR algorithm creates a priority queue for each group, sorted in decreasing order of relevance. This is followed by construction of a new ranking as follows. To fill each position in the new ranking, it identifies the queue with the most-qualified (top) item and pops from that queue. If that selection results in a sub-ranking which violates the fairness constraint, it identifies the queue with the next most-qualified top item and pops from that queue instead.

We denote our modification of FA\*IR as Fair Queues. This algorithm works in a similar fashion to FA\*IR, but there are two key differences. FA\*IR models each sub-ranking on  $n'$  items using a binomial distribution  $p(k; n', p)$  and checks that  $p(k; n', p) > \alpha$ , while Fair Queues applies our non-probabilistic DDP constraint on full rankings. Second, our fairness definition is based on group exposure in aggregate across multiple time steps and applies to multi-group settings, while the fairness definition in [29] only applies to single rankings with two groups (usually denoted as the protected and non-protected groups).



**Figure 1: An example illustrating the can be fair pruning for the action space.** There are 3 groups: male, female, and non-binary. The DDP threshold is 0.25. The model checks if it can select from the male queue and still create a fair ranking. Both possible ranking completions are unfair after selecting a male item, indicating that selection from the male queue is not a valid action for this time step. The red highlights indicate the group pairs which attain the maximum average exposure difference (DDP).

We denote the subroutine that checks if a ranking can be completed while satisfying the fairness constraint as *can be fair* (illustrated in Figure 1). A naive approach would examine every completion of the ranking until it finds a fair completion or until it exhaustively examines all enumerations of ranking completions. Since there are  $n!$  rankings on  $n$  elements, we use a heuristic (Section 4.1.1) to find a single ranking completion. If that heuristic-based

ranking completion is unfair, *can be fair* fails, and we restrict the algorithm from selecting from the queue in consideration. In case all queues are eliminated, we select from the queue of the group with the minimum exposure.

Using the heuristic for *can be fair*, Fair Queues has a worst-case complexity of  $\Theta(gn^2)$  for a ranking with  $n$  items and  $g$  groups. The use of a heuristic rather than an exact method for *can be fair* implies that we might sometimes over-restrict the action space. However, even with the heuristic, *can be fair* never incorrectly allows for selecting a queue which precludes a final fair ranking, as long as there is a queue which allows for a final fair ranking. Therefore, while the reconstructed ranking might be sub-optimal with respect to nDCG, it will be fair whenever possible.

**4.1.1 Can Be Fair Heuristic.** Our heuristic completes a ranking using the same basic framework as Fair Queues - it selects a queue to draw from at each step. However, rather than selecting from the queue with the most relevant top item, it selects from the queue with the least expected exposure if we fill each remaining position by selecting from a *random* queue. To calculate the expected exposure, we first calculate the average exposure for all remaining open slots in the ranking. We then compute the expected exposure for a group by assuming that each remaining item in the group's queue receives the average remaining exposure. We can then average the exposures for each group under this assumption, and select from the queue with the lowest expected exposure. We do not claim this heuristic creates the optimally fair ranking completion, but rather treat it as a reasonable approximation.

## 4.2 Greedy Fair Swap

Our second deterministic policy denoted as Greedy Fair Swap aims to promote members of groups with lower exposure within a single ranking  $r^{(t)}$ . The algorithm (Algorithm 1) iteratively selects the most highly ranked member (highest relevance score) of a lower-exposure group which is still below a member of a higher-exposure group, and swaps them. Note that this swap is greedy because it minimally lowers nDCG while guaranteeing a lower DDP. The algorithm terminates when the rankings up to time  $t$  meet the DDP threshold  $\alpha$ . Since there are  $\binom{n}{2}$  possible swaps, Greedy Fair Swap is  $O(n^2)$  for re-ranking a batch with  $n$  items. This algorithm does not necessarily produce a ranking with optimal nDCG under the fairness constraint.

## 5 POLICY BASED ON LEARNING TO SEARCH

Both Greedy Fair Swap and Fair Queues have a potentially undesirable property by definition, which is they only act when it is absolutely required. Consequently, the DDP measure stays very close to the threshold at all times. If these policies suddenly have to re-rank a batch with highly relevant items from the group with the highest exposure, these will be forced to take a large penalty on nDCG to maintain the DDP under the fairness threshold. This weakness motivates a more pro-active learned policy. In this section, we describe a learned policy based on a variation of learning to search.

---

### Algorithm 1: Greedy Fair Swap

---

```

input :Initial ranking  $r_{\text{init}}$  on items  $\{i_1, \dots, i_n\}$ , group membership function  $g$ , threshold  $\alpha'$ 
output :Ranking  $r$  on  $\{i_1, \dots, i_n\}$ , with  $DDP(r) \leq \alpha'$ 
1 Initialize  $r = r_{\text{init}}$ 
2 while  $DDP(r) > \alpha'$  do
3   Identify the group with highest exposure  $G_h$ 
4   Identify the group with lowest exposure  $G_l$ 
5   Set  $l = \arg \min_{i_j \in G_l \mid \exists i_{j'} \in G_h, r(i_{j'}) < r(i_j)} r(i_j)$ 
6   Set  $h = \arg \max_{i_{j'} \in G_h \mid r(i_{j'}) < r(l)} r(i_{j'})$ 
7   Swap  $l$  and  $h$  in  $r$ 
8 return  $r$ 

```

---

## 5.1 Background

**Locally optimal learning to search (LOLS)** [7] learns a policy by imitating and extending a *reference policy*. Since the learned policy provably has low regret on deviations from the reference, it is possible to improve upon the performance of the reference [7]. The learned policy can be trained so as to predict an action from features derived from the *state space* at a given timestep. Below we summarize the concept at a very high level. LOLS constructs a training example by “rolling in” up to a given number of time steps according to the learned policy. For every action in the action space, LOLS “rolls out” using the reference policy (or possibly a mixture of the reference and the learned policy). This roll out terminates at an end state, and a score can be assigned to that end state. Using these scores, the model learns to prioritize actions which led to high scoring end states at a given time step.

## 5.2 LOLS With Queues (L2SQ)

Our proposed approach, Locally Optimal Learning to Search with Queues, merges the learning to search algorithm reviewed in Section 5.1 with the queue-based ranking procedure described in Section 4.1. A detailed description of L2SQ can be found in Algorithm 2. Concretely, we create a scoring model (a feedforward neural network) that maps from a partial ranking and a collection of queues (one per group) to a score for each queue. We select from the queue with the top score from the model, rather than the queue with the most-relevant item. Intuitively, we would like the L2SQ model to learn to maintain a fairness buffer well below the DDP threshold, allowing the model to take advantage of incoming batches with highly relevant items from a high-exposure group.

To implement the LOLS framework, we must define a reference policy, a parametrization of the state and action spaces, and a cost function to be applied at the end of roll-outs. At training time, we construct training examples where each example consists of a rolled-in set of rankings up to some timestep (described below) and a choice of queues from which to select the next element of the current ranking. We then roll-out for each possible choice of queue to obtain costs for each queue. From this pairing of state and costs, we construct multiple training examples to update the scoring model. To construct a set of rankings at inference time, we apply the scoring model for each slot of each ranking, filling in slots with the top item from the highest-scoring queue at each step.

**Algorithm 2:** L2SQ Training

---

```

input : Sets of initial rankings  $\{R_{\text{init}}^{(n)}\}_{n=1}^N$ , mixture parameter
 $\beta \geq 0$ , and roll-out horizon  $h$ 
1 for  $n \in \{1, 2, \dots, N\}$  do
2    $R_{\text{init}} \leftarrow R_{\text{init}}^{(n)}$ 
3   for  $t \in \{1, 2, \dots, T\}$  do
4     Roll-in  $t - 1$  rounds to reach  $r_{\text{init}}^{(t)} \in R_{\text{init}}$ 
5     Create priority queue  $Q_g$  (ordered by decreasing relevance)
          for all groups  $g$ 
6     Initialize  $r^{(t)} = \emptyset$ 
7     while  $|Q_g| > 0$  for at least 2 groups  $g$  do
8       for  $g \in \{1, 2, \dots, G\}$  do
9         if  $|Q_g| > 0$  and can_be_fair( $r^{(t)}, Q_g$ ) then
10           Copy  $r'^{(t)} \leftarrow r^{(t)}$ 
11           Insert  $Q_g.pop()$  into  $r'^{(t)}$ 
12           Apply roll-out policy to fill  $r'^{(t)}$  and the next
               $h$  batches  $r'^{(t+1)}, \dots, r'^{(t+h)}$ 
13           Compute cumulative nDCG after roll-out for
              group  $g$ 
14           For all rolled-out  $g$ , compute cost = max nDCG for any
              group minus nDCG of  $g$ 
15           Construct training example from groups  $g$ 
16           Compute BPR loss
17           Apply roll-out policy to insert  $Q_g.pop()$  into  $r^{(t)}$ 
18   Update model with total BPR loss

```

---

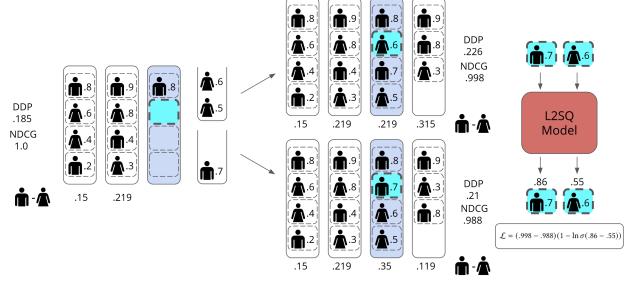
We parametrize the search space over queues (rather than over items) because DDP is based on groups and is agnostic to the choice of individual elements within a group.

**Reference Policy** Any ranking policy can be used as the reference policy. All our results use Fair Queues as the reference policy, since L2SQ did better with Fair Queues as a reference than with Greedy Fair Swap in early experiments.

**Parametrization of State and Action Spaces** We encode the state space using 17 features per group: mean exposure and percentage of the group in previous batches, total number of items in current ranking, statistics of relevance scores and ranks for items which have already been ranked (min, max, mean, standard deviation), the relevance score of the top item in the queue, size of the queue, and statistics of relevance scores for the queue (min, max, mean, standard deviation). We parametrize the model using a feedforward neural network, which takes as input all features for all groups and outputs a vector of scores, one per group.

The action space consists of all selections from non-empty queues which can result in a ranking that satisfies our constraint. We use the *can be fair* subroutine described in Section 4.1 to restrict the action space for L2SQ.

**Roll-out and Cost Computation** To create training examples, we roll-in up to a certain time step, simulate selecting from each non-restricted queue, then roll-out from each simulated choice to compute a loss function. The policy used for roll-out is a mixture of the learned policy and the reference policy, where the reference policy is selected with probability  $\beta$ . We calculate the score of each queue using the average nDCG over all batches after roll-out. An illustration of roll-out with two groups (male/female), four timesteps, and a DDP threshold of 0.25 is shown in Figure 2.



**Figure 2: Roll-out and loss function computation at a single time step.** We display the relevance of each item, as well as the difference in male and female exposures for each completed batch. We roll-out after selecting from each queue to calculate the post-roll-out nDCGs. The loss is a function of the post-roll-out nDCGs and the model’s scores. Note the model’s scores encode the model’s preferences for selecting from each group’s queue, not the relevances of particular items.

**Training Examples and Loss Function** We create multiple pairwise examples per state, comparing each queue to the queue with the highest post-roll-out nDCG. The model assigns each queue in the pair a score, and we compute the Bayesian Personalized Ranking loss [19] based on the pair of scores and final nDCGs for the two queues. If  $Q_1$  is the queue with the highest final nDCG, then for every non-restricted  $Q_2$  at a given timestep  $t$ , we calculate the loss as  $l(Q_2, t) = (nDCG(Q_1) - nDCG(Q_2))(1 - \ln \sigma(f(Q_1) - f(Q_2)))$ , where  $\sigma$  is the logistic function and  $f(\cdot)$  is the score of the model for a queue. Note that we do not calculate any losses for actions which are restricted by *can be fair*. An example of loss function computation is in Figure 2 (right).

**Inference** At inference time, we apply the scoring model for each slot of each ranking, filling in slots with the top item from the highest-scoring queue at each step. We apply the *can be fair* restriction on the action space at inference time as well, to ensure that the generated rankings are fair (if possible). Because of the *can be fair* restriction on the action space, the L2SQ model has a worst-case complexity of  $\Theta(gn^2)$  at inference time for a ranking with  $n$  items and  $g$  groups.

## 6 EXPERIMENTS

### 6.1 Data

We evaluate our proposed approach on the following four real world datasets: UCI German Credit, StackExchange, AirBnB, and a new dataset Resume. In addition, we analyse our algorithms on a synthetic dataset described below. In each case, we construct training examples with 10 batches each, and validation and test examples with 25 batches each.

**Synthetic Data** We construct a synthetic dataset to study the behavior of our proposed algorithms. We define four groups for generating this synthetic dataset. We generate the data by sampling an initial component  $\sim U(0, 1)$  and adding a value sampled from a Gaussian random variable with standard deviation  $\sigma = 0.1$  to each

group. We set the Gaussian random variable to have a negative mean for two of the four groups, where  $\mu \sim U(-.75, -.25)$  is sampled for each batch, and zero mean for the remaining two groups. For each batch, we select a random number of elements from each group (specifically, between 3 and 7 items per group). We sample 100 training examples, 50 validation examples, and 50 test examples.

**German Credit** Hans Hofmann [12] introduced the German Credit dataset which maps applicants to credit rating/scores generated by the private German credit agency Shufa. Each applicant has features such as age, gender, marital status, etc. There are 1,000 applicants overall. Our data was created similar to [29] who assigned a score to each applicant as the sum of the relevant numeric or ordered attributes<sup>1</sup>. We rank the applicants by this score. For our experiments we take the cross-product of gender and age to create four groups. To convert these rankings we shuffle the data and then split into batches of size 20. We sample 100 training examples, 50 validation examples, and 50 test examples.

**AirBnB** Although their setting is different, we take inspiration from [3] to create a ranking dataset from AirBnB listings. Note that our results are not directly comparable to [3], because they consider individual fairness criteria while we consider DDP which is defined for groups. We create a dataset using AirBnB data<sup>2</sup> for the cities Boston, Seattle, and New York. There are 37,171 listings in our dataset. Each listing is considered an item, and we obtain item scores by summing ratings for each of 7 attributes: cleanliness, check-in, communication, location, review score, value, and accuracy. Ratings are generally very close to the maximum of 10. Batches (of size 20) are uniformly sampled at random from the total population. We create four groups of listings based on price per night: \$0-\$50, \$51-\$100, \$101-\$200, and \$201-\$1000. Although listing price is not a sensitive attribute, our fairness constraint will ensure that some listings from each price range appear in the results, providing cheaper listings more opportunity to be seen while giving the user diversity of choice. From this construction, we sample 300 training examples, 50 validation examples, and 50 test examples.

**Stack Exchange** Also inspired by [3], we create a realistic query-answer dataset from the StackExchange online archive<sup>3</sup>. As with AirBnB, our results on this dataset are not directly comparable to their results due to the difference in problem setting. Each ranking is composed of answers to a question asked on the Unix, AskUbuntu, and Academia StackExchange websites. We restrict ourselves to questions with at least 4 answers. There are 28,026 questions that meet this criterion. For each question, we rank the question’s answers using cosine similarity between Latent Semantic Indexing vectors<sup>4</sup>. Answers are assigned a group based on the reputation of the posting user. We split users into 3 bins: low reputation (1-50), medium reputation (51-2,500), and high reputation (>2,500), so each item falls into one of these three groups. User reputation is not typically a sensitive attribute, but applying our fairness constraint improves diversity of results and gives less-experienced users more chance to gain reputation. As with other datasets, a single “example” consists of 10 questions for training or 25 questions for

validation/test. We sample 1000 training examples, 50 validation examples, and 50 test examples.

**Resume** Motivated by the setting of screening candidates for a job in an online, batched setting, we construct a dataset of batched resumes from a freely available list of 14,800 parsed resumes for software engineers, data scientists, and other computer science related professionals across India<sup>5</sup>. We trained a simple model to predict, from the body of the resume, whether the applicant should be considered for a software developer position. As a simple approximation of such a signal, we train a logistic regression model to predict (using only the resume’s body) whether or not the resume’s title contains the word “developer.” We use the scores output by the logistic regression model as the relevance scores. To sort resumes into groups, we use the resume’s “state” field to map each resume to one of four regions of the country: North, South, East, and West. We use a batch size of 20 items each. We sample 200 training examples, 50 validation, and 50 test examples.

## 6.2 Experimental Setup

For each dataset we split the items into two populations, one for training and validation and the other for test. We sample sets of rankings in a way that ensures no item is repeated within a single ranking, but may be repeated across rankings. For training we use 10 timesteps for each set of rankings, and for validation and test we use 25 timesteps. We use Fair Queues as a reference policy for L2SQ, rolling out for three timesteps before calculating the loss for each deviation. The DDP threshold  $\alpha$  for all datasets is set to 0.1. We optimize the model with Adam for 20 epochs. The model is a feed forward neural network with 17 features for each group (listed in Section 5.2), 2 hidden layers of size  $\lfloor \frac{17}{2} \rfloor$  and  $\lfloor \frac{17}{4} \rfloor$  features per group, and an output layer of size equal to the number of groups. The model for each dataset was trained on a Xeon Gold 6240 CPU @ 2.60GHz with 192GB RAM, though in practice we found the model did not use more than 10 GB of memory.

We tune the mixture parameter  $\beta$  on the validation set and keep the best model for test. We set the learning rate at 0.001 for all datasets. The test data consists of 50 sets of rankings for each dataset. We evaluate each algorithm on the test data and calculate, for each timestep, the mean and 95% confidence interval for nDCG and DDP across all 50 sets of rankings.

We compare the initial rankings to re-rankings from Greedy Fair Swap, Fair Queues, and L2SQ. Although there is a large amount of prior work on fair ranking, we are specifically focused on post-processing approaches that guarantee fairness over time in an online, multi-batch setting. We are not aware of prior work that directly fulfills these criteria, and thus do not make any further comparisons.

Our code is available online<sup>6</sup>.

## 6.3 Results

We begin with the synthetic data. In this setting L2SQ attains the highest nDCG on the test data while still meeting the fairness threshold (top row of Figure 3).

<sup>1</sup>[https://www.github.com/MilkaLichtblau/FA-IR\\_Ranking](https://www.github.com/MilkaLichtblau/FA-IR_Ranking)

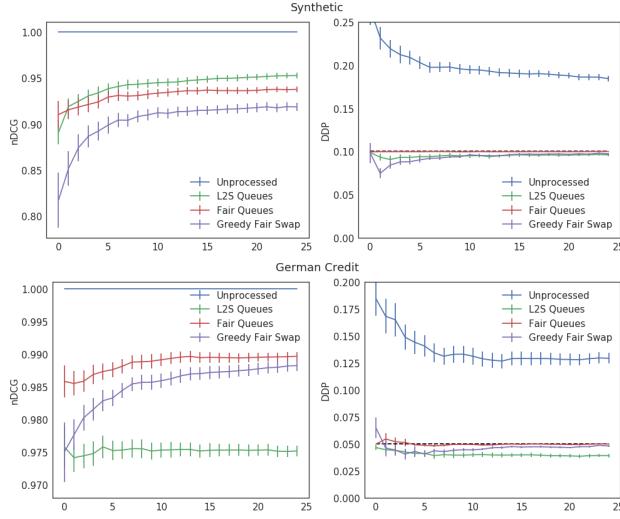
<sup>2</sup><http://insideairbnb.com/>

<sup>3</sup><https://archive.org/details/stackexchange>

<sup>4</sup><https://radimrehurek.com/gensim/models/lisimodel.html>

<sup>5</sup><https://www.kaggle.com/avaniisiddhapura27/resume-dataset>

<sup>6</sup>[https://github.com/ejohnson0430/fair\\_online\\_ranking](https://github.com/ejohnson0430/fair_online_ranking)



**Figure 3: Results on synthetic data and German Credit data over 25 timesteps.**

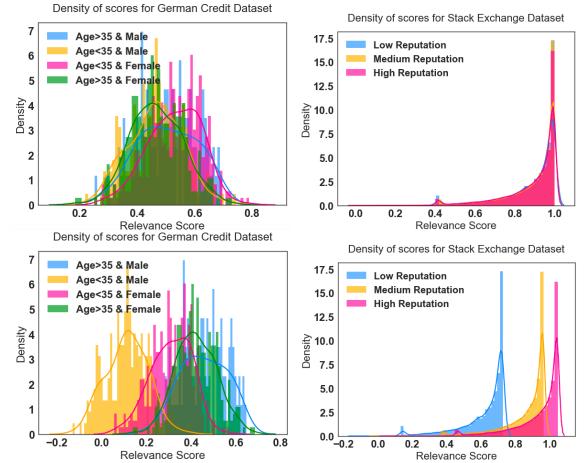
On the real datasets, we find that our two deterministic approaches are effective at enforcing fairness while maintaining high nDCG. L2SQ enforces fairness, but its nDCG lags behind the nDCG of the deterministic algorithms. The results on the German Credit dataset are shown in the bottom row of Figure 3 (note that we lower the DDP threshold to  $\alpha = 0.05$  since the unprocessed rankings already have low DDP). Results are similar on the other 3 real datasets and are omitted.

All our approaches produce fair rankings. However, our results suggest that the deterministic algorithms outperform L2SQ on the real datasets in terms of nDCG, but L2SQ excels on the synthetic dataset. By construction, the groups in the synthetic data have significant differences in mean relevance scores across groups, potentially requiring disruptive re-ranking to maintain fairness. The real datasets exhibit a low difference in mean relevance scores in the unprocessed rankings. We hypothesize that this discrepancy drives the behavior seen in the initial results.

We consequently run additional experiments with a wider spread of mean relevance scores for the real datasets, discussed in Sections 6.3.1 and 6.3.2. Section 6.4 provides further discussion on the algorithms’ sensitivity to the difference in mean relevance scores across groups.

**6.3.1 Score Distributions Across Groups.** We find that for the real datasets, the distribution of scores is not significantly different across groups. We display the relevance score distributions for the German Credit and Stack Exchange datasets in the top row of Figure 4, though Airbnb and Resume show a similar pattern.

In the previous section, we hypothesized that L2SQ outperforms our deterministic approaches when the initial rankings are very unfair. To test this on real data, we sample new relevance scores for all 4 real datasets by adding positive values to some groups and negative values to others. We then evaluate the L2SQ model and our baselines on these datasets.



**Figure 4: Distribution on GermanCredit dataset (left column) and StackExchange dataset (right column) with noise (top row) and without noise (bottom row).**

While constructing the rankings for each dataset, we add values sampled from a Gaussian random variable with zero mean for groups with higher exposure and a negative mean for groups with lower exposure. These values are drawn from  $\mathcal{N}(0, 0.1)$  and  $\mathcal{N}(-0.25, 0.1)$  respectively for German Credit, Resume, and Stack Exchange, where relevance scores are between 0 and 1. The values are drawn from  $\mathcal{N}(0, 0.25)$  and  $\mathcal{N}(-1, 0.25)$  respectively for AirBnB, where the relevance scores are between 0 and 7.

As an example, we show the new distribution of scores on German Credit and StackExchange in the bottom row of Figure 4. The groups form separated modes in the space of scores, indicating that any rankings sampled from the perturbed data will likely not satisfy the fairness constraint without further processing.

**6.3.2 Results With Modified Score Distributions.** We show results for all datasets in Figure 5. We see that L2SQ typically outperforms Fair Queues and Greedy Fair Swaps in terms of nDCG, and that all methods are successful at maintaining the DDP below the threshold at all steps, despite the fact that the unprocessed rankings nearly always exceed the threshold.

On Resume and StackExchange, L2SQ also visibly maintains a DDP significantly below the threshold and much lower than the other policies. L2SQ also displays a higher advantage in terms of nDCG over the other policies on these two datasets. As previously indicated, we believe that L2SQ’s main advantage is its ability to provide a buffer well under the threshold, allowing it to take advantage of batches which provide high nDCG only if items with already high exposure are chosen. We find that on the German Credit and AirBnB datasets, L2SQ is difficult to distinguish from the other policies.

## 6.4 Sensitivity to Relevance Distributions

To determine how the L2SQ model performs with respect to the degree of inherent bias, we perform an experiment systematically varying the degree of inherent bias in the synthetic dataset. We train

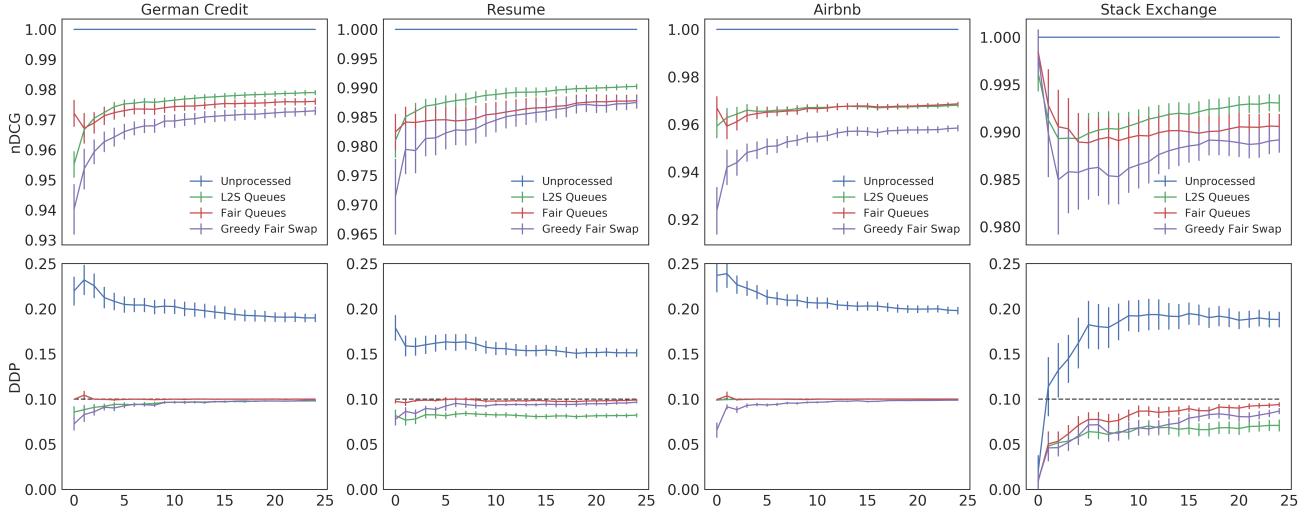


Figure 5: Results on all modified datasets over 25 timesteps.

L2SQ on 36 different synthetic datasets corresponding to six values (0.0 to 0.5 by 0.1) for the mean of the Gaussian random variable for groups 0 and 1, and the same six values for the negative mean of Gaussian random variable for groups 2 and 3. After training each model for 20 epochs, we take the model which scores best on the validation data and compare it to the other algorithms. Figure 6 visualizes the difference in the sum of nDCG scores of the batches re-ranked by each algorithm.

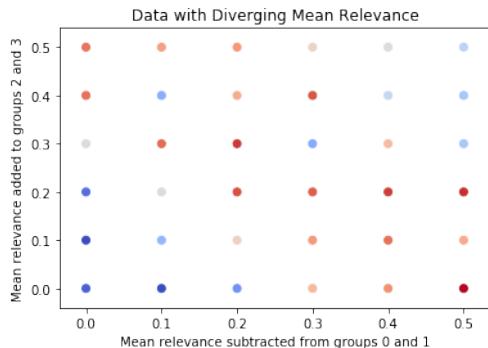


Figure 6: Red indicates L2S performed better, while blue indicates that either Greedy Fair Swap or Fair Queues performed better. The opacity of the color indicates degree of improvement.

L2SQ performs best when the groups have some spread between them, but its performance is not as good when the groups have very similar or very dissimilar scores. If we denote the total difference in means as the “spread,” it seems to perform best with a spread of 0.4 - 0.7. We hypothesized that L2SQ would perform better with a wider spread, but it seems that as the spread becomes too large it struggles. In this range the model may have a limited state space to

explore during training, which would inhibit its potential to learn a better policy.

## 7 CONCLUSION

We propose an algorithmic framework for post-processing of ranking batches while satisfying a given fairness constraint, when operating in an online environment. We evaluate two deterministic policies as well as a novel learning to search based policy L2SQ. Extensive experiments on synthetic as well as real world data sets demonstrate the effectiveness of our proposed approaches. The experiments also demonstrate that our approach can be generalized to consider multiple groups i.e., the synthetic dataset with different group sizes, StackExchange with three groups, and other datasets with four groups. Our approach is also robust to the batch size which varies among the different datasets. While we present our results using the DDP criterion, our approach can incorporate general fairness criteria and can operate on ranking models for a variety of applications.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback, and Andrew McCallum for introducing us as part of the DS696: Industry Mentorship Independent Study at UMass. This work was supported in part by the Chan Zuckerberg Initiative. The work reported here was performed in part using high performance computing equipment obtained under a grant from the Collaborative RD Fund managed by the Massachusetts Technology Collaborative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsor.

## REFERENCES

- [1] Abolfazl Asudeh, HV Jagadish, Julia Stoyanovich, and Gautam Das. 2019. Designing fair ranking schemes. In *Proceedings of the 2019 International Conference on Management of Data*. 1259–1276.
- [2] Alex Beutel, Jilin Chen, Tulsee Doshi, Hai Qian, Li Wei, Yi Wu, Lukasz Heldt, Zhe Zhao, Lichan Hong, Ed H. Chi, and Cristos Goodrow. 2019. Fairness in Recommendation Ranking through Pairwise Comparisons. arXiv:1903.00780 [cs.CY]
- [3] Asia J Biega, Krishna P Gummadi, and Gerhard Weikum. 2018. Equity of attention: Amortizing individual fairness in rankings. In *The 41st international acm sigir conference on research & development in information retrieval*. 405–414.
- [4] Jaime Carbonell and Jade Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 335–336.
- [5] L Elisa Celis, Anay Mehrotra, and Nisheeth K Vishnoi. 2020. Interventions for ranking in the presence of implicit bias. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. 369–380.
- [6] L Elisa Celis, Damian Straszak, and Nisheeth K Vishnoi. 2017. Ranking with fairness constraints. *arXiv preprint arXiv:1704.06840* (2017).
- [7] Kai-Wei Chang, Akshay Krishnamurthy, Alekh Agarwal, John Langford, and Hal Daumé III. 2015. Learning to search better than your teacher. In *International Conference on Machine Learning*.
- [8] Alexandra Chouldechova and Aaron Roth. 2018. The frontiers of fairness in machine learning. *arXiv preprint arXiv:1810.08810* (2018).
- [9] Sam Corbett-Davies and Sharad Goel. 2018. The measure and mismeasure of fairness: A critical review of fair machine learning. *arXiv preprint arXiv:1808.00023* (2018).
- [10] Yifan Guan, Abolfazl Asudeh, Pranav Mayuram, HV Jagadish, Julia Stoyanovich, Gerome Miklau, and Gautam Das. 2019. Mithrranking: A system for responsible ranking design. In *Proceedings of the 2019 International Conference on Management of Data*. 1913–1916.
- [11] Moritz Hardt, Eric Price, and Nathan Srebro. 2016. Equality of Opportunity in Supervised Learning. *CoRR* abs/1610.02413 (2016).
- [12] Hans Hofmann. 1994. Statlog (german credit data) data set. *UCI Repository of Machine Learning Databases* (1994).
- [13] Thorsten Joachims, Adith Swaminathan, and Tobias Schnabel. 2016. Unbiased Learning-to-Rank with Biased Feedback. arXiv:1608.04468 [cs.IR]
- [14] Matthew Kay, Cynthia Matuszek, and Sean A Munson. 2015. Unequal representation and gender stereotypes in image search results for occupations. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. 3819–3828.
- [15] Caitlin Kuhlman, MaryAnn VanValkenburg, and Elke Rundensteiner. 2019. FARE: Diagnostics for Fair Ranking using Pairwise Error Metrics. In *The World Wide Web Conference*. 2936–2942.
- [16] Juhi Kulshrestha, Motahare Eslami, Johnnatan Messias, Muhammad Bilal Zafar, Saptarshi Ghosh, Krishna P Gummadi, and Karrie Karahalios. 2017. Quantifying search bias: Investigating sources of bias for political searches in social media. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. 417–432.
- [17] Matevž Kunaver and Tomaž Požrl. 2017. Diversity in recommender systems—A survey. *Knowledge-Based Systems* 123 (2017), 154–162.
- [18] Swetasudha Panda, J. Tristan, M. Wick, Haniyeh Mahmoudian, and P. Kanani. 2019. Using Bayes Factors to Control for Fairness Case Study on Learning To Rank.
- [19] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2012. BPR: Bayesian personalized ranking from implicit feedback. *arXiv preprint arXiv:1205.2618* (2012).
- [20] Tetsuya Sakai and Ruihua Song. 2011. Evaluating diversified search results using per-intent graded relevance. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. 1043–1052.
- [21] Piotr Sapiezynski, Wesley Zeng, Ronald E Robertson, Alan Mislove, and Christo Wilson. 2019. Quantifying the Impact of User Attention on Fair Group Representation in Ranked Lists. In *Companion Proceedings of The 2019 World Wide Web Conference*. 553–562.
- [22] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.
- [23] Ashudeep Singh and Thorsten Joachims. 2018. Fairness of exposure in rankings. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2219–2228.
- [24] Ashudeep Singh and Thorsten Joachims. 2019. Policy learning for fairness in ranking. In *Advances in Neural Information Processing Systems*. 5426–5436.
- [25] Julia Stoyanovich, Ke Yang, and HV Jagadish. 2018. Online set selection with fairness and diversity constraints. In *Proceedings of the EDBT Conference*.
- [26] Ke Yang and Julia Stoyanovich. 2017. Measuring fairness in ranked outputs. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. 1–6.
- [27] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez-Rodriguez, and Krishna P Gummadi. 2019. Fairness Constraints: A Flexible Approach for Fair Classification. *J. Mach. Learn. Res.* 20, 75 (2019), 1–42.
- [28] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. 2017. Fairness constraints: Mechanisms for fair classification. In *Artificial Intelligence and Statistics*. PMLR, 962–970.
- [29] Meike Zehlike, Francesco Bonchi, Carlos Castillo, Sara Hajian, Mohamed Maged, and Ricardo Baeza-Yates. 2017. Fa\*ir: A fair top-k ranking algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 1569–1578.
- [30] Meike Zehlike and Carlos Castillo. 2020. Reducing disparate exposure in ranking: A learning to rank approach. In *Proceedings of The Web Conference 2020*. 2849–2855.
- [31] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. 2013. Learning fair representations. In *International Conference on Machine Learning*. 325–333.

---

# Conjugate Energy-Based Models

---

Hao Wu <sup>\*1</sup> Babak Esmaili <sup>\*1</sup> Michael Wick <sup>2</sup> Jean-Baptiste Tristan <sup>3</sup> Jan-Willem van de Meent <sup>1</sup>

## Abstract

In this paper, we propose conjugate energy-based models (CEBMs), a new class of energy-based models that define a joint density over data and latent variables. The joint density of a CEBM decomposes into an intractable distribution over data and a tractable posterior over latent variables. CEBMs have similar use cases as variational autoencoders, in the sense that they learn an unsupervised mapping from data to latent variables. However, these models omit a generator network, which allows them to learn more flexible notions of similarity between data points. Our experiments demonstrate that conjugate EBMs achieve competitive results in terms of image modelling, predictive power of latent space, and out-of-domain detection on a variety of datasets.

## 1. Introduction

Deep generative models approximate a data distribution by combining a prior over latent variables with a neural generator, which maps latent variables to points on a data manifold. It is common to evaluate these models in terms of their ability to generate realistic examples, or their estimated densities for unseen data. However, an arguably more important use case for these models is unsupervised representation learning. If a generator can faithfully represent the data in terms of a lower-dimensional set of latent variables, then we hope that these variables will encode a set of semantically meaningful factors of variation that will be relevant to a broad range of downstream tasks.

Guiding a model towards a semantically meaningful representation requires some form of inductive bias. A large body of work on variational autoencoders (VAEs, (Kingma & Welling, 2013; Rezende et al., 2014)) has explored the use of priors as inductive biases. Relatively mild biases in

the form of conditional independence are common in the literature on disentangled representations (Higgins et al., 2016; Kim & Mnih, 2018; Chen et al., 2018; Esmaili et al., 2019). More generally, recent work has shown that defining priors that reflect the structure of the underlying data will lead to representations that are easier to interpret and generalize better. Examples include priors that represent objects in an image (Eslami et al., 2016; Lin et al., 2020b; Engelcke et al., 2019; Crawford & Pineau, 2019b), or moving objects in video (Crawford & Pineau, 2019a; Kosiorek et al., 2018; Wu et al., 2020; Lin et al., 2020a).

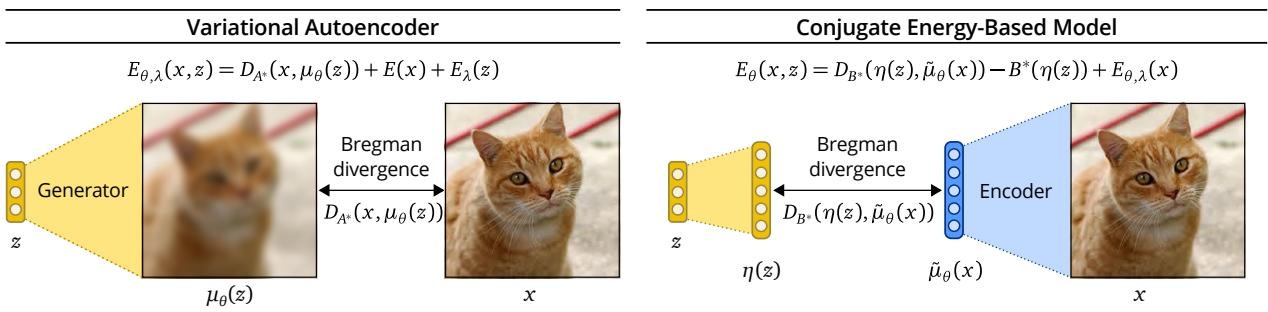
Despite steady progress, work on disentangled representations and structured VAEs still predominantly considers synthetic data. VAEs employ a neural generator that is optimized to reconstruct examples in the training set. For complex natural scenes, learning a generator that can produce pixel-perfect reconstructions poses fundamental challenges, given the combinatorial explosion of possible inputs. This is not only a problem for generation, but also from the perspective of the learned representation; a VAE must encode all factors of variation that give rise to large deviations in pixel space, regardless of whether these factors are semantically meaningful (e.g. presence and locations of objects) or not (e.g. shadows of objects in the background of the image).

The motivating question that we consider in this paper is whether it is possible to train latent-variable models without minimizing pixel-level discrepancies between an image and its reconstruction. Instead, we would like to design an objective that minimizes the discrepancy between the encoding of an image and the latent variables, which will in general be in a lower-dimensional space compared to the input. Our hope is that doing so will allow a model to learn more abstract representations, in the sense that it becomes easier to discard factors of variation that give rise to variation in pixel space, but should be considered noise.

In this paper, we consider energy-based models (EBMs) with latent variables as a particular instantiation of this general idea. EBMs with latent variables are by no means new; they have a long history in the context of restricted Boltzmann machines (RBMs) and related models (Smolensky, 1986; Hinton, 2002; Welling et al., 2004). Our motivation in the present work is to design a class of EBMs that retain the desirable features of VAEs, but employ a discriminative

<sup>\*</sup>Equal contribution <sup>1</sup>Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA <sup>2</sup>Oracle Labs, MA, USA <sup>3</sup>Computer Science department, Boston College, MA, USA. Correspondence to: Hao Wu <wu.hao10@northeastern.edu>, Babak Esmaili <esmaili.b@northeastern.edu>.

## Conjugate Energy-Based Models



**Figure 1.** Comparison between a VAE and a CEBM. A variational autoencoder with a Gaussian or Bernoulli likelihood has an energy that can be expressed in terms of a Bregman divergence in the data space  $D_{A^*}(x, \mu_\theta(z))$  between an image  $x$  and the reconstruction from the generator network  $\mu_\theta(z)$ . The energy function in a CEBM can be expressed in terms of a Bregman divergence in the latent space  $D_{B^*}(\eta(z), \tilde{\mu}_\theta(x))$  between a vector of natural parameters  $\eta(z)$  and the output of an encoder network  $\tilde{\mu}_\theta(x)$ . See main text for details.

energy function to model data at an intermediate level of representation that does not necessarily encode all features of an image at the pixel level.

Concretely, we propose conjugate EBMs (CEBMs), a new family of energy-based latent-variable models in which the energy function defines a neural exponential family. While the normalizer of CEBMs is intractable, we can nonetheless compute the posterior in closed form when we pair the likelihood with an appropriate conjugate bias term. As a result, the neural sufficient statistics in a CEBM fully determine both the marginal likelihood and the encoder, thereby side-stepping the need for a generator (Figure 1).

Our contributions can be summarized as follows:

1. We propose CEBMs, a class of energy-based models for unsupervised representation learning. The density of a CEBM factorizes into a tractable posterior and an energy-based marginal over data. This means that CEBMs can be trained using existing methods for EBMs, whilst inference is tractable at test time.
2. Unlike VAEs, CEBMs model data not at the pixel level, but at the level of the latent representation. We interpret the energy function of CEBMs in terms of a Bregman divergence in the latent space, and show that the density of a VAE can similarly be expressed in terms of a Bregman divergence in the data space.
3. We show that two of the most common inductive biases in VAEs can be incorporated in CEBMs: a spherical Gaussian and a mixture of Gaussians.
4. We evaluate how well CEBMs learned representations agree with class labels (which are not used during training). We show that neighbors are more likely to belong to the same class, which translates to increased performance in downstream classification tasks. Moreover, CEBMs perform competitively in out-of-domain detection. We do also note limitations; in particular we observe that CEBMs suffer from posterior collapse.

## 2. Background

### 2.1. Energy-Based Models

An EBM (LeCun et al., 2006) defines a probability density for  $x \in \mathbb{R}^D$  via the Gibbs-Boltzmann distribution

$$p_\theta(x) = \frac{\exp\{-E_\theta(x)\}}{Z_\theta}, \quad Z_\theta = \int dx \exp\{-E_\theta(x)\}.$$

The function  $E_\theta : \mathbb{R}^D \rightarrow \mathbb{R}$  is called the energy function which maps each configuration to a scalar value, the energy of the configuration. This type of model is widely used in statistical physics, for example in Ising models. The distribution can only be evaluated up to an unknown constant of proportionality, since computing the normalizing constant  $Z_\theta$  (also known as the partition function) requires an intractable integral with respect to all possible inputs  $x$ .

Our goal is to learn a model  $p_\theta(x)$  that is close to the true data distribution  $p_{\text{data}}(x)$ . A common strategy is to minimize the Kullback-Leibler divergence between the data distribution and the model, which is equivalent to maximizing the expected log-likelihood

$$\begin{aligned} \mathcal{L}(\theta) &= \mathbb{E}_{p_{\text{data}}(x)} [\log p_\theta(x)], \\ &= \mathbb{E}_{p_{\text{data}}(x)} [-E_\theta(x)] - \log Z_\theta. \end{aligned} \tag{1}$$

The key difficulty when performing maximum likelihood estimation is that computing the gradient of  $\log Z_\theta$  is intractable. This gradient can be expressed as an expectation with respect to  $p_\theta(x)$ ,

$$\nabla \log Z_\theta = \mathbb{E}_{p_\theta(x')} [-\nabla_\theta E_\theta(x')], \tag{2}$$

which means that the gradient of  $\mathcal{L}(\theta)$  has the form:

$$\nabla_\theta \mathcal{L}(\theta) = -\mathbb{E}_{p_{\text{data}}(x)} [\nabla_\theta E_\theta(x)] + \mathbb{E}_{p_\theta(x')} [\nabla_\theta E_\theta(x')].$$

This corresponds to *maximizing* the probability of samples  $x \sim p_{\text{data}}(x)$  from the data distribution and *minimizing* the probability of samples  $x' \sim p_\theta(x')$  from the learned model.

Contrastive divergence methods (Hinton, 2002) compute a Monte Carlo estimate of this gradient, which requires a method for approximate inference to generate samples  $x' \sim p_\theta(x')$ . A common method for generating samples from EBMs is Stochastic Gradient Langevin Dynamics (SGLD, (Welling & Teh, 2011)), which initializes a sample  $x'_0 \sim p_0(x')$  and performs a sequence of gradient updates with additional injected noise  $\epsilon$ ,

$$x'_{i+1} = x'_i - \frac{\alpha}{2} \frac{\partial E_\theta(x')}{\partial x'} + \epsilon, \quad \epsilon \sim N(0, \alpha). \quad (3)$$

SGLD is motivated as a discretization of a stochastic differential equation whose stationary distribution is equal to the target distribution. It is correct in the limit  $i \rightarrow \infty$  and  $\alpha \rightarrow 0$ , but in practice will have a bias.

The initialization  $x'_0$  is crucial because it determines the number of steps needed to converge to a high-quality sample. For this reason, EBMs are commonly trained using persistent contrastive divergence (PCD, (Du & Mordatch, 2019; Tielemans, 2008)), which initializes some samples from a replay buffer  $\mathcal{B}$  of previously generated samples (Nijkamp et al., 2019a; Du & Mordatch, 2019; Xie et al., 2016).

## 2.2. Energy-Based Latent-Variable Models

Energy-based latent-variable models are a subclass of EBMs where the energy function defines joint density on observed data  $x \in \mathbb{R}^D$  and latent variable  $z \in \mathbb{R}^K$ ,

$$p_\theta(x, z) = \frac{\exp\{-E_\theta(x, z)\}}{Z_\theta}. \quad (4)$$

Some of the most well-known examples of this family of models include restricted Boltzmann machines (RBMs, (Smolensky, 1986; Hinton, 2002)), deep belief nets (DBNs, (Hinton et al., 2006)), and deep Boltzmann machines (DBMs, (Salakhutdinov & Hinton, 2009)).

Similar to standard EBMs, energy-based latent-variable models can also be trained using contrastive divergence methods, where the gradient of  $\mathcal{L}(\theta)$  can be expressed as:

$$-\mathbb{E}_{p_{\text{data}}(x)p_\theta(z|x)}[\nabla_\theta E_\theta(x, z)] + \mathbb{E}_{p_\theta(x', z')}[\nabla_\theta E_\theta(x', z')].$$

Estimating this gradient has the additional problem of requiring samples from the posterior  $p_\theta(z|x)$  which is also intractable in general.

## 2.3. Conjugate Exponential Families

An exponential family is a set of distributions whose probability density can be expressed in the form

$$p(x | \eta) = h(x) \exp\{\langle t(x), \eta \rangle - A(\eta)\}, \quad (5)$$

where  $h : \mathcal{X} \rightarrow \mathbb{R}^+$  is a base measure,  $\eta \in \mathcal{H} \subseteq \mathbb{R}^K$  is a vector of natural parameters,  $t : \mathcal{X} \rightarrow \mathbb{R}^K$  is a vector of

sufficient statistics, and  $A : \mathcal{H} \rightarrow \mathbb{R}$  is the log normalizer (or cumulant function),

$$A(\eta) = \log Z(\eta) = \int dx h(x) \exp\{\langle t(x), \eta \rangle\}. \quad (6)$$

If a likelihood belongs to an exponential family, then there exists a conjugate prior that is itself an exponential family

$$p(\eta | \lambda, \nu) = \exp\{\langle \eta, \lambda \rangle - A(\eta)\nu - B(\lambda, \nu)\}. \quad (7)$$

The convenient property of conjugate exponential families is that both the marginal likelihood  $p(x | \lambda, \nu)$  and the posterior  $p(\eta | x, \lambda, \nu)$  are tractable. If we define

$$\tilde{\lambda}(x) = \lambda + t(x), \quad \tilde{\nu} = \nu + 1, \quad (8)$$

then the posterior and marginal likelihood are

$$\begin{aligned} p(\eta | x, \lambda, \nu) &= p(\eta | \tilde{\lambda}(x), \tilde{\nu}), \\ p(x | \lambda, \nu) &= h(x) \exp\{B(\tilde{\lambda}(x), \tilde{\nu}) - B(\lambda, \nu)\}. \end{aligned} \quad (9)$$

## 2.4. Legendre Duality in Exponential Families

Two convex functions  $A : \mathcal{H} \rightarrow \mathbb{R}^+$  and  $A^* : \mathcal{M} \rightarrow \mathbb{R}^+$  on spaces  $\mathcal{H} \subseteq \mathbb{R}^K$  and  $\mathcal{M} \subseteq \mathbb{R}^K$  are conjugate duals when

$$A^*(\mu) := \sup_{\eta \in \mathcal{H}} \{\langle \mu, \eta \rangle - A(\eta)\}. \quad (10)$$

When  $A$  is a function of Legendre type (see Rockafellar (1970) for details), the gradients of these functions define a bijection between conjugate spaces by mapping points to their corresponding suprema

$$\eta(\mu) = \nabla A^*(\mu), \quad \mu(\eta) = \nabla A(\eta), \quad (11)$$

such that we can express  $A^*(\mu)$  at the supremum as

$$A^*(\mu) = \langle \mu, \eta(\mu) \rangle - A(\eta(\mu)) \quad (12)$$

The log normalizer  $A(\eta)$  of an exponential family is of Legendre type when the family is regular and minimal ( $\mathcal{H}$  is an open set and sufficient statistics  $t(x)$  are linearly independent; see Wainwright & Jordan (2008) for details). We refer to  $\mathcal{M}$  as the mean parameter space, since we can express any  $\mu \in \mathcal{M}$  as the expected value of the sufficient statistics

$$\mu(\eta) = \mathbb{E}_{p(x|\eta)}[t(x)]. \quad (13)$$

## 2.5. Bregman Divergences and Exponential Families

A Bregman divergence for a function  $F : \mathcal{M} \rightarrow \mathbb{R}$  that is continuously-differentiable and strictly convex on a closed set  $\mathcal{M}$  has the form

$$D_F(\mu', \mu) = F(\mu') - F(\mu) - \langle \mu' - \mu, \nabla F(\mu) \rangle. \quad (14)$$

Well-known special cases of Bregman divergences include the squared distance ( $F(\mu) = \langle \mu, \mu \rangle$ ) and the Kullback-Leiber (KL) divergence ( $F(\mu) = \sum_k \mu_k \log \mu_k$ ).

Any Bregman divergence can be associated with an exponential family and vice versa, where  $F(\mu) = A^*(\mu)$  is the conjugate dual of  $A(\eta)$  (see Banerjee et al. (2005)). To see this, we re-express the log density of a (regular and minimal) exponential family using the substitution  $\mu = \nabla A(\eta)$ <sup>1</sup>,

$$\begin{aligned} \log p(x | \eta) &= \langle t(x), \eta \rangle - A(\eta), \\ &= (\langle \mu, \eta \rangle - A(\eta)) + \langle t(x) - \mu, \eta \rangle, \\ &= A^*(\mu) + \langle t(x) - \mu, \nabla A^*(\mu) \rangle, \\ &= -D_{A^*}(t(x), \mu) + A^*(t(x)). \end{aligned} \quad (15)$$

In other words, the log density of an exponential family can be expressed in terms of a bias term  $A^*(t(x))$ <sup>2</sup>, and a notion of agreement in the form of a Bregman divergence  $D_{A^*}(t(x), \mu)$  between the sufficient statistics  $t(x)$  and the mean parameters  $\mu$ . We will make use of this property of exponential families to provide an interpretation of both CEBMs and VAEs in terms of Bregman divergences.

### 3. Conjugate Energy-Based Models

We are interested in learning a probabilistic model that defines a joint density  $p_{\theta, \lambda}(x, z)$  over high-dimensional data  $x \in \mathbb{R}^D$  and a lower-dimensional set of latent variables  $z \in \mathbb{R}^K$ . The intuition that guides our work is that we would like to measure agreement between latent variables and data at a high level of representation, rather than at the level of individual pixels, where it may be more difficult to distinguish informative features from noise. To this end, we will explore energy-based models as an alternative to VAEs.

Concretely, we propose to consider models of the form

$$p_{\theta, \lambda}(x, z) = \frac{1}{Z_{\theta, \lambda}} \exp \{ -E_{\theta, \lambda}(x, z) \}, \quad (16)$$

where the energy function takes a form that is inspired by exponential family distributions

$$E_{\theta, \lambda}(x, z) = -\langle t_\theta(x), \eta(z) \rangle + E_\lambda(z). \quad (17)$$

In this energy function,  $\theta$  are the weights of a network  $t_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^H$ , which plays the role of an encoder by mapping high-dimensional data to a lower-dimensional vector of neural sufficient statistics. The function  $\eta : \mathbb{R}^K \rightarrow \mathbb{R}^H$  maps latent variables to a vector of natural parameters in the same space as the neural sufficient statistics. The function  $E_\lambda : \mathbb{R}^K \rightarrow \mathbb{R}$  serves as an inductive bias, with hyperparameters  $\lambda$ , that plays a role analogous to the prior.

<sup>1</sup>We here omit the base measure  $h(x)$  for notational simplicity.

<sup>2</sup>Or  $A^*(t(x)) + \log h(x)$  when we include  $h(x)$  the density.

We will consider a bias  $E_\lambda(z)$  in form of a tractable exponential family with sufficient statistics  $\eta(z)$

$$E_\lambda(z) = -\log p_\lambda(z) = -\langle \eta(z), \lambda \rangle + B(\lambda). \quad (18)$$

We can then express the energy function as

$$E_{\theta, \lambda}(x, z) = -\langle \lambda + t_\theta(x), \eta(z) \rangle + B(\lambda). \quad (19)$$

This form of the energy function has a convenient property: It corresponds to a model  $p_{\theta, \lambda}(x, z)$  in which the posterior  $p_{\theta, \lambda}(z | x)$  is tractable. To see this, we make a substitution  $\tilde{\lambda}_\theta(x) = \lambda + t_\theta(x)$  analogous to the one in Equation 8, which allows us to express the energy as

$$E_{\theta, \lambda}(x, z) = -\langle \eta(z), \tilde{\lambda}_\theta(x) \rangle + B(\tilde{\lambda}_\theta(x)) + E_{\theta, \lambda}(x), \quad (20)$$

$$E_{\theta, \lambda}(x) = -B(\tilde{\lambda}_\theta(x)) + B(\lambda). \quad (21)$$

We see that we can factorize the corresponding density

$$p_{\theta, \lambda}(x, z) = p_{\theta, \lambda}(x) p_{\theta, \lambda}(z | x), \quad (22)$$

which yields a posterior and marginal that are analogous the distributions in Equation 9

$$p_{\theta, \lambda}(z | x) = p(z | \tilde{\lambda}_\theta(x)), \quad (23)$$

$$\begin{aligned} p_{\theta, \lambda}(x) &= \frac{1}{Z_{\theta, \lambda}} \exp \{ -E_{\theta, \lambda}(x) \}, \\ &= \frac{1}{Z_{\theta, \lambda}} \exp \{ B(\tilde{\lambda}_\theta(x)) - B(\lambda) \}. \end{aligned} \quad (24)$$

In other words, the joint density of this model factorizes into a tractable posterior  $p_{\theta, \lambda}(z | x)$  and an intractable energy-based marginal likelihood  $p_{\theta, \lambda}(x)$ . This posterior is conjugate, in the sense that it is in the same exponential family as the bias. For this reason, we refer to this class of models as conjugate energy-based models (CEBMs).

### 4. Relationship to VAEs

CEBMs differ from VAEs in that they lack a generator network. Instead, the density is fully specified by the encoder network  $t_\theta(x)$ , which defines a notion of agreement  $\langle \tilde{\lambda}_\theta(x), \eta(z) \rangle$  between data and latent variables in the latent space. As with other exponential families, we can make this notion of agreement explicit by expressing the conjugate posterior in terms of a Bregman divergence using the decomposition in Equation 15

$$\begin{aligned} E_{\theta, \lambda}(x, z) &= D_{B^*}(\eta(z), \tilde{\mu}_\theta(x)) \\ &\quad - B^*(\eta(z)) + E_{\theta, \lambda}(x). \end{aligned} \quad (25)$$

Here  $B^*(\mu)$  is the conjugate dual of the the log normalizer  $B(\lambda)$ , and we use  $\tilde{\mu}_\theta(x) = \mu(\tilde{\lambda}_\theta(x))$  as a shorthand for the mean-space posterior parameters. We see that maximizing the density corresponds to minimizing a Bregman divergence in the space of sufficient statistics of the bias.

In Figure 1, we compare CEBMs to VAE in terms of the energy function for the log density of the generative model. In making this comparison, we have to keep in mind that these models are trained using different methods, and that VAEs have a tractable density  $p_\theta(x, z)$ . That said, the objectives in both models maximize the marginal likelihood, so we believe that it is instructive to write down the corresponding Bregman divergence in the VAE likelihood. This likelihood is typically a Gaussian with known variance, or a Bernoulli distribution (when modeling binarized images). Both distributions have sufficient statistics  $t(x) = x$ . Once again omitting the base measure  $h(x)$  for expediency, we can express the log density of a VAE as an energy

$$\begin{aligned} E_{\theta, \lambda}(x, z) &= -\log p_\theta(x|z) - \log p_\lambda(z), \\ &= -\langle x, \eta_\theta(z) \rangle + A(\eta_\theta(z)) - \log p_\lambda(z). \quad (26) \\ &= D_{A^*}(x, \mu_\theta(z)) - A^*(x) - \log p_\lambda(z) \end{aligned}$$

Here  $A^*(x)$  is the conjugate dual of the log normalizer  $A(\eta)$ , and we use  $\eta_\theta(z)$  and  $\mu_\theta(z)$  to refer to the output of the generator network in the natural-parameter and the mean-parameter space respectively. To reduce clutter and accommodate the case where a base measure  $h(x)$  is needed (e.g. that of a Gaussian likelihood with known variance), we will introduce the additional shorthands

$$E(x) = -A(x) - \log h(x), \quad E_\lambda(z) = -\log p_\lambda(z). \quad (27)$$

We then see that the energy function of a VAE has the form

$$E_{\theta, \lambda}(x, z) = D_{A^*}(x, \mu_\theta(z)) + E(x) + E_\lambda(z). \quad (28)$$

Like that of a CEBM, the energy function of a VAE contains a Bregman divergence, as well as two terms that depend only on  $x$  and  $z$ . However, whereas the Bregman divergence in CEBM is defined in the mean-parameter space of the latent variables, that of a VAE is computed in the data space.

## 5. Inductive Biases

CEBMs have a property that is somewhat counter-intuitive. While the posterior  $p_{\theta, \lambda}(z | x)$  in this class of models is tractable, the prior is in general not tractable. In particular, although the bias  $-E_\lambda(z)$  is the logarithm of a tractable exponential family, it is not the case that  $p_{\theta, \lambda}(z) = p_\lambda(z)$ . Rather the prior  $p_{\theta, \lambda}(z)$  has the form,

$$p_{\theta, \lambda}(z) = \frac{\exp\{-E_\lambda(z)\}}{Z_{\theta, \lambda}} \int dx \exp\{\langle t_\theta(x), \eta(z) \rangle\}.$$

In other words,  $E_\lambda(z)$  defines an inductive bias, but this bias is different from the tractable prior in a VAE<sup>3</sup>, in the

<sup>3</sup>The bias in a VAE contains the log prior  $\log p_\lambda(z)$  and the log normalizer  $A(\eta_\theta(z))$  of the likelihood. In a CEBM, by contrast, we omit the term  $A_\theta(\eta(z)) = \log \int dx \exp\{\langle t_\theta(x), \eta(z) \rangle\}$ , which is intractable, and hereby implicitly absorb it into its prior.

sense that it imposes only a soft constraint on the geometry of the latent space.

In principle, the bias in a CEBM can take the form of any exponential family distribution. Since products of exponential families are also in the exponential family, this covers a broad range of possible biases. For purposes of evaluation in this paper, we will constrain ourselves to two cases:

**1. Spherical Gaussian.** As a bias that is analogous to the standard prior in VAEs, we consider a spherical Gaussian with fixed hyperparameters  $(\mu, \sigma) = (0, 1)$  for each dimension of  $z \in \mathbb{R}^K$ ,

$$E_\lambda(z) = -\sum_k (\langle \eta(z_k), \lambda \rangle - B(\lambda)).$$

Each term has sufficient statistics  $\eta(z_k) = (z_k, z_k^2)$ , natural parameters  $\lambda$ , and log normalizer  $B(\lambda)$  as

$$\begin{aligned} \lambda &= \left( \frac{\mu}{\sigma^2}, -\frac{1}{2\sigma^2} \right) = \left( 0, -\frac{1}{2} \right), \\ B(\lambda) &= -\frac{\lambda_1^2}{4\lambda_2} - \frac{1}{2} \log(-2\lambda_2). \end{aligned}$$

The marginal likelihood of the CEBM is then

$$p_{\theta, \lambda}(x) = \frac{1}{Z_{\theta, \lambda}} \exp \left\{ \sum_k (B(\tilde{\lambda}_{\theta, k}(x)) - B(\lambda)) \right\},$$

where  $\tilde{\lambda}_{\theta, k}(x) = \lambda + t_{\theta, k}(x)$  and  $t_{\theta, k}(x)$  is the sufficient statistics that corresponds to  $z_k$ .

**2. Mixture of Gaussians.** In our experiments, we will consider datasets that are normally used for classification. These datasets, by design, exhibit multimodal structure that we would like to see reflected in the learned representation. In order to design a model that is amenable to uncovering this structure, we will extend the energy function in Equation 17 to contain a mixture component  $y$

$$E_{\theta, \lambda}(x, y, z) = -\langle t_\theta(x), \eta(y, z) \rangle + E_\lambda(y, z).$$

As an inductive bias, we will consider a bias in the form of a mixture of  $L$  Gaussians,

$$E_\lambda(y, z) = -\sum_{k, l} I[y = l] (\langle \eta(z_k), \lambda_{l, k} \rangle - B(\lambda_{l, k})).$$

Here  $z \in \mathbb{R}^K$  is a vector of features and  $y \in \{1, \dots, L\}$  is a categorical assignment variable. The bias for each component  $l$  is a spherical Gaussian with hyperparameters  $\lambda_{l, k}$  for each dimension  $k$ . Again, using the notation  $\tilde{\lambda}_{\theta, l, k} = \lambda_{l, k} + t_{\theta, k}(x)$  to refer to the posterior parameters, then we obtain an energy

$$E_{\theta, \lambda}(x, y, z) = -\sum_{k, l} I[y = l] (\langle \eta(z_k), \tilde{\lambda}_{\theta, l, k} \rangle - B(\lambda_{l, k})).$$

We can then define a joint probability over data  $x$  and the assignment  $y$  in terms the log normalizer  $B(\cdot)$ ,

$$p_{\theta,\lambda}(x, y) = \frac{1}{Z_{\theta,\lambda}} \exp \left\{ \sum_{k,l} I[y = l] (B(\tilde{\lambda}_{\theta,l,k}) - B(\lambda_{l,k})) \right\},$$

which then allows us to compute the marginal  $p_{\theta,\lambda}(x)$  by summing over  $y$ . We optimize this marginal with respect hyperparameters  $\lambda_{l,k}$  as well as the weights  $\theta$ .

## 6. Related Work

**Energy-Based Latent-Variable Models.** The idea of using EBMs to jointly model data and latent variables has a long history in the machine learning literature. Examples of this class of models include restricted Boltzmann machines (RBMs, (Smolensky, 1986; Hinton, 2002)), deep belief nets (DBNs, (Hinton et al., 2006)), and deep Boltzmann machines (DBMs, (Salakhutdinov & Hinton, 2009)). The idea of extending RBMs in exponential families and exploiting conjugacy to yield a tractable posterior is also not new and has been explored in Exponential Family Harmoniums (EFHs; (Welling et al., 2004)). These models differ from CEBMs in that they employ a bilinear interaction term  $x^T W z$ , which ensures that both the likelihood  $p(x | z)$  and  $p(z | x)$  are tractable. In CEBMs, the corresponding term  $t_\theta(x)^T z$  is nonlinear, which means that the posterior is tractable, but the likelihood is not. We provide a more detailed discussion regarding the connection of our work to this class of models in Appendix A.

**EBMs for Image Modelling.** Recent work has shown that EBMs with convolutional energy functions can accurately model distributions over images (Xie et al., 2016; Nijkamp et al., 2019a;b; Du & Mordatch, 2019; Xie et al., 2021a). This line of work typically focuses on generation and not on unsupervised representation learning as we do here. A line of work, which is similar to ours in spirit, employs EBMs as priors on the latent space of deep generative models (Pang et al., 2020; Aneja et al., 2020). These approaches, unlike our work, require a generator.

**Interpretation of other models as EBMs.** Grathwohl et al. (2019); Liu & Abbeel (2020); Xie et al. (2016) have proposed to interpret a classifier as an EBM that defines a joint energy function on the data and labels. CEBMs with a discrete bias can interpreted as the unsupervised variant of this model class. Che et al. (2020) interpret a GAN as an EBM defined by both the generator and discriminator.

**Training EBMs.** A commonly used training method is PCD (Tieleman, 2008), where the MCMC is initialized from a replay buffer that stores the previously generated samples (Du & Mordatch, 2019), or from a generator (Xie

et al., 2018; 2020; 2021a). Nijkamp et al. (2019a;b) comprehensively investigate the convergence of PCD based on a variety of factors such as MCMC initialization, network architecture, and the optimizer. They find that the difference between the energy of the data and model samples is a good diagnostic of training stability. Many of these findings were helpful during the training and evaluation in our work.

There is a large literature on alternative training methods. Gao et al. (2020) propose to use the noise contrastive estimation (NCE, (Gutmann & Hyvärinen, 2010)), where they pretrain a flow-based noise model and then train the EBM to discriminate between the real data examples and the ones generated from the noise model. Another popular approach is the score matching (SM, Vértes et al. (2016); Hyvärinen & Dayan (2005); Vincent (2011); Song et al. (2020); Bao et al. (2020)), which learns EBMs by matching the gradient of the log probability density of the model distribution to that of the data distribution. Bao et al. (2020) propose a bi-level version of this method where it is also applicable to latent-variable models. To sidestep the need of MCMC sampling, Han et al. (2019; 2020); Xie et al. (2021b) jointly train an EBM with a VAE in an adversarial manner; Grathwohl et al. (2021) learn a generator by entropy regularization. We refer the readers to Song & Kingma (2021) for a more comprehensive discussion on training methods for EBMs.

## 7. Experiments

Our experiments evaluate to what extent CEBMs can learn representations that encode meaningful factors of variation, whilst discarding details about the input that we would consider noise. This question is difficult to answer in generality, and in some sense not well-posed; whether a factor of variation should be considered signal or noise can depend on context. For this reason, our experiments primarily focus on the extent to which representations in CEBMs can recover the multimodal structure in datasets that are normally used for classification. While class labels are an imperfect proxy, in the sense that they do not reflect all factors of variation that we may want to encode in a representation, they provide a means of quantifying differences between representations that were learned in an unsupervised manner.

We begin with a qualitative evaluation by visualizing samples and latent representation. We then demonstrate that learned representations align with class structure, in the sense that nearest neighbors in the latent space are more likely to belong to the same class (section 7.2). Next, we evaluate performance on out-of-distribution detection (OOD) tasks which, although not our primary focus in this paper, are a common use case for EBMs (Section 7.3). We then quantify the extent to which the learned representations can improve performance in downstream task, we measure few-label classification accuracy for representations that



Figure 2. Samples generated from a CEBM trained on MNIST, Fashion-MNIST, SVHN and CIFAR-10.

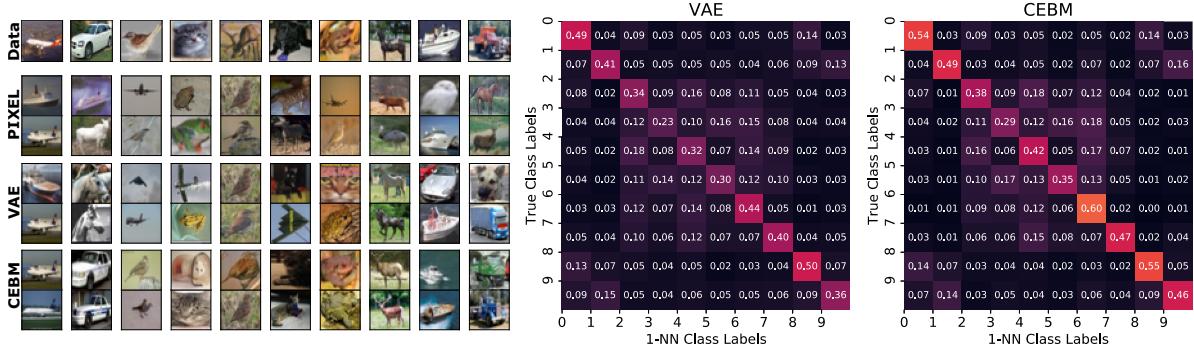


Figure 3. (Left) Samples from CIFAR-10 along with the top 2-nearest-neighbors in pixel space, the latent space of a VAE, and the latent space of a CEBM. (Right) Confusion matrices of 1-nearest-neighbor classification on CIFAR-10 based on L2 distance in the latent space. On average, CEBM representations more closely align with class labels compared to VAE.

were pre-trained without supervision (Section 7.4). Finally, we perform a more in-depth study of the latent space where we investigate to what extend the aggregate posterior distribution is close to the inductive bias as well how vulnerable CEBMs are to posterior collapse (Section 7.5).

### 7.1. Network Architectures and Training

**Architectures & Optimization.** The CEBMs in our experiments employ an encoder network  $t_\theta(x)$  in the form of 4-layer CNN (as proposed by Nijkamp et al. (2019a)), followed by an MLP output layer. We choose the dimension of latent variables to be 128. We found that the optimization becomes difficult with smaller dimensions. We train our models using 60 SGLD steps, 90k gradient steps, batch size 128, Adam optimizer with learning rate 1e-4. For training stability, we L2 regularize energy magnitudes (proposed by Du & Mordatch (2019)). See Appendix C for details.

**Hyperparameter Sensitivity.** As observed in previous work (Du & Mordatch, 2019; Grathwohl et al., 2019), training EBMs is challenging and often requires a thorough hyperparameters search. We found that the choices of activation function, learning rate, number of SGLD steps, and regularization will all affect training stability. Models regularly diverge during training, and it is difficult to perform diagnostics given that  $\log p_{\theta,\lambda}(x)$  cannot be computed. As suggested by (Nijkamp et al., 2019a), we found checking the difference in energy between data and model samples

can help to verify training stability. In general we also observed a trade-off between sample quality and the predictive power of latent variables in our experiments. We leave investigation of the source of this trade-off to future work, but we suspect that this is because SGLD has more difficulty converging when the latent space is more disjoint.

### 7.2. Samples and Latent Space

We begin with a qualitative evaluation by visualizing samples from the model. While generation is not our intended use case in this paper, such samples do serve as a diagnostic that allows us to visually inspect what characteristics of the input data are captured by the learned representation.

Figure 2 shows samples from CEBMs trained on MNIST, Fashion-MNIST, SVHN, and CIFAR-10. We initialize the samples with uniform noise and run 500 SGLD steps. We observe that the distribution over images is diverse and captures the main characteristics of the dataset. Sample quality is roughly on par with samples from other EBMs (Nijkamp et al., 2019a), although it is possible to generate samples with higher visual quality using class-conditional EBMs (Du & Mordatch, 2019; Grathwohl et al., 2019; Liu & Abbeel, 2020) (which assume access to labels).

To assess to whether the representation in CEBMs aligns with classes in each dataset, we look at the agreement between the label of an input and that of its nearest neighbor in the latent space. The latent representations are inferred by

Table 1. AUROC scores in OOD Detection. We use  $\log p_\theta(\mathbf{x})$  and  $\|\nabla_{\mathbf{x}} \log p_\theta(\mathbf{x})\|$  as score functions. The left block shows results of the models trained on F-MNIST and tested on MNIST, E-MNIST, Constant (C); The right block shows results of the models trained on CIFAR-10 and tested on SVHN, Texture and Constant (C).

	Fashion-MNIST						CIFAR-10					
	$\log p_\theta(\mathbf{x})$			$\ \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x})\ $			$\log p_\theta(\mathbf{x})$			$\ \nabla_{\mathbf{x}} \log p_\theta(\mathbf{x})\ $		
	MNIST	E-MNIST	C	MNIST	E-MNIST	C	SVHN	Texture	C	SVHN	Texture	C
VAE	.50	.39	.09	.61	.57	.01	.42	<b>.58</b>	.41	.38	<b>.51</b>	.37
IGEBM	.35	.36	.90	.78	.82	.96	.45	.31	.64	.33	.17	<b>.62</b>
CEBM	.37	.34	.90	<b>.82</b>	<b>.89</b>	<b>.98</b>	.47	.32	<b>.66</b>	.31	.17	.54
GMM-CEBM	<b>.56</b>	<b>.56</b>	<b>.92</b>	.56	.80	.95	<b>.55</b>	.30	.62	<b>.40</b>	.23	<b>.62</b>

Table 2. Average classification accuracy on the test set. We train a variety of deep generative models on MNIST, Fashion-MNIST, CIFAR-10, and SVHN in an unsupervised way. Then we use the learned latent representations to train logistic classifiers with 1, 10, 100 training examples per class, and the full training set. We train each classifier 10 times on randomly drawn training examples.

Models	MNIST				Fashion-MNIST				CIFAR-10				SVHN			
	1	10	100	full	1	10	100	full	1	10	100	full	1	10	100	full
VAE	42	85	92	95	41	63	72	81	16	22	31	38	<b>13</b>	13	16	36
GMM-VAE	53	86	93	97	49	68	79	84	<b>19</b>	23	33	39	<b>13</b>	14	23	56
BIGAN	33	67	85	91	46	65	75	81	18	<b>30</b>	<b>43</b>	52	11	20	42	56
IGEBM	63	89	95	97	50	<b>70</b>	79	83	16	26	33	42	10	16	35	49
CEBM	<b>67</b>	89	95	97	<b>52</b>	<b>70</b>	77	83	<b>19</b>	<b>30</b>	42	<b>53</b>	12	<b>25</b>	<b>48</b>	<b>70</b>
GMM-CEBM	<b>67</b>	<b>91</b>	<b>97</b>	<b>98</b>	<b>52</b>	<b>70</b>	<b>80</b>	<b>85</b>	16	29	42	52	10	17	39	60

computing the mean of the posterior  $p_{\theta,\lambda}(z|x)$ . In Figure 3, we show samples from CIFAR-10, along with the images that correspond to the nearest neighbors in pixel space, the latent space of a VAE, and the latent space of a CEBM. The distance in pixel space is a poor measure of similarity in this dataset, whereas proximity in the latent space is more likely to agree with class labels in both VAEs and CEBMs. We additionally show visualization of the latent space with UMAP (McInnes et al., 2018) in Figure 5.

In Figure 3 (right), we quantify this agreement by computing the fraction of neighbors in each class conditioned on the class of the original image. We see a stronger alignment between classes and the latent representation in CEBMs, which is reflected in higher numbers on the diagonal of the matrix. On average, a fraction of 0.38 of the nearest neighbors are in the same class in the VAE, whereas 0.45 of the neighbors are in the same class in the CEBM. This suggest that the representation in CEBMs should lead to higher performance in downstream classification tasks. We will evaluate this performance in Section 7.4.

### 7.3. Out-of-Distribution Detection

EBMs have formed the basis for encouraging results in out-of-distribution (OOD) detection (Du & Mordatch, 2019; Grathwohl et al., 2019). While not our focus in this paper, OOD detection is a benchmark that helps evaluate whether a

learned model accurately characterizes the data distribution. In Table 1, we report results in terms of two metrics. The first is the area under the receiver-operator curve (AUROC) when thresholding the log marginal  $\log p_{\theta,\lambda}(x)$ . The second is the gradient-based score function proposed by Grathwohl et al. (2019). We observe that in most cases, CEBM yields a similar score to the VAE and IGEBM baselines.

### 7.4. Few-label Classification

To evaluate performance in settings where few labels are available, we use pre-trained representations (which were learned without supervision) to train logistic classifiers with 1, 10, 100 training examples per class, as well as the full training set. We evaluate classification performance for a spherical Gaussian bias (CEBM) and the mixture of Gaussians bias (GMM-CEBM). We compare our models against the IGEBM (Du & Mordatch, 2019)<sup>4</sup>, a standard VAE with the spherical Gaussian prior, GMM-VAE (Tomczak & Welling, 2018) where the prior is a mixture of Gaussians (GMM), and BIGAN (Donahue et al., 2016).

We report the classification accuracy on the test set in Table 2. CEBMs overall achieve a higher accuracy compared to VAEs in particular for CIFAR-10 and SVHN where the

<sup>4</sup>Since the IGEBM does not explicitly have latent representations, we extract features from the last layer of the energy function.

Table 3. KL divergence between aggregate posterior and prior and the mutual information between data and latent variables.

	VAE		CEBM		GMM-CEBM	
	KL	MI	KL	MI	KL	MI
MNIST	11.5	9.1	0.9	0.3	18.7	4.7
FMNIST	3.5	9.0	0.6	0.4	8.1	3.9
CIFAR10	21.5	9.2	0.1	0.2	4.5	2.7
SVHN	8.6	10.1	0.1	0.1	5.6	2.2

pixel distance is not good measure for similarity. Moreover, we observe that CEBMs outperform the IGEBM. This suggests that the inductive biases in CEBMs can lead to increased performance in downstream tasks. The performance between BIGANs and CEBMs is not as distinguishable which we suspect is due the fact BIGANs, just like CEBMs, do not define a likelihood that measure similarity at the pixel level. We also observe that the CEBM with the GMM inductive bias does not always outperform the one with the Gaussian inductive bias, which we suspect is due to GMM-CEBM having more difficulty to converge.

### 7.5. Limitations: Posterior Collapse

While our experiments demonstrate that CEBMs are able to reasonably approximate the data distribution and learn latent representations that are in closer agreement with class labels, they do not evaluate the learned notation of posterior uncertainty, and more generally the role of inductive bias. In this subsection, we ask the following two questions: (1) Does the aggregate posterior distribution of the training data live close to the inductive bias  $p_\lambda(z)$ ? (2) What is the mutual information between latent variables and the training data?

To evaluate whether encoded examples are distributed according to the bias  $p_\lambda(z)$ , we compute the divergence  $\text{KL}(\tilde{p}_{\theta,\lambda}(z) \parallel p_\lambda(z))$  between the bias and the aggregate posterior, which is a mixture over training data

$$\hat{p}_{\theta,\lambda}(z) = \frac{1}{N} \sum_n p_{\theta,\lambda}(z|x_n), \quad x_n \sim p_{\text{data}}(x).$$

There are two reasons to consider this distribution, rather than the marginal  $p_{\theta,\lambda}(z)$  of the CEBM. The first is computational expedience; it is easier to approximate  $\hat{p}_{\theta,\lambda}(z)$  than it is to approximate  $p_{\theta,\lambda}(z)$ , since the latter requires samples  $x \sim p_{\theta,\lambda}(x)$  from the marginal of the CEBM. The second reason is that  $\hat{p}_{\theta,\lambda}(z)$  reflects the distribution over features that we might use in a downstream task.

We approximate  $\hat{p}_{\theta,\lambda}(z)$  with a Monte Carlo estimate over batches of size 1k (see [Esmaeili et al. \(2019\)](#)), which we use to estimate both the KL and the mutual information (see Table 3). Because the marginal KL in CEBMs is significantly lower compared to VAEs across datasets, we conclude that

CEBMs indeed attempted to place the aggregate posterior distribution close to the inductive bias.

Our evaluation of the mutual information proved more surprising: CEBMs learn a representation that has a very low mutual information between  $x$  and  $z$ . The reason for this is that the posterior parameters  $\tilde{\lambda}_\theta(x) = \lambda + t_\theta(x)$  are dominated by the parameters of the bias  $\lambda$ , which means that model essentially ignores the sufficient statistics  $t_\theta(x)$ , which tend to have a small magnitude relative to  $\lambda$ . This phenomenon could be interpreted as an instance of *posterior collapse* ([Alemi et al., 2017](#)), which has been observed in a variety of contexts when training variational autoencoders by maximizing the marginal likelihood, which in itself is not an objective that guarantees a high mutual information.

## 8. Discussion

In this paper, we introduced CEBMs, a class of latent-variable models that factorize into an energy-based distribution over data and a tractable posterior over latent variables. CEBMs can be trained using standard methods for EBMs and in this sense have a small “edit distance” relative to existing approaches, whilst also providing a mechanism for incorporating inductive biases for latent variables.

Our experimental results are encouraging but also raise questions. We observe a closer agreement between the unsupervised representation and class labels than in VAEs, which translates into improved performance in downstream classification tasks. At the same time, we observe that CEBMs do not learn a meaningful notion of uncertainty; the CEBM posterior is typically dominated by the inductive bias, which means that there is a very low mutual information between data and latent variables.

This work opens up a number of lines of future research. First and foremost, this work raises the question what objectives would be most suitable for learning energy-based latent-variable models in a manner maximizes agreement with respect to both the data distribution and the inductive bias terms, whilst also ensuring a sufficiently high mutual information between data and latent variables. More generally, we see opportunities to develop CEBMs with structured bias terms as an alternative to models based on VAEs in settings where we are hoping to reason about structured representations with little or no supervision.

## Acknowledgements

We would like to thank our reviewers for their thoughtful comments, as well as Heiko Zimmermann, Will Grathwohl and Jacob Kelly for helpful discussions. This work was supported by the Intel Corporation, the 3M Corporation, NSF award 1835309, startup funds from Northeastern University, the Air Force Research Laboratory (AFRL), and DARPA.

## References

- Alemi, A. A., Poole, B., Fischer, I., Dillon, J. V., Saurous, R. A., and Murphy, K. Fixing a broken elbo. *arXiv preprint arXiv:1711.00464*, 2017.
- Aneja, J., Schwing, A., Kautz, J., and Vahdat, A. Ncp-vae: Variational autoencoders with noise contrastive priors. *arXiv preprint arXiv:2010.02917*, 2020.
- Banerjee, A., Merugu, S., Dhillon, I. S., and Ghosh, J. Clustering with Bregman Divergences. *Journal of Machine Learning Research*, 6(58):1705–1749, 2005. ISSN 1533-7928.
- Bao, F., Li, C., Xu, T., Su, H., Zhu, J., and Zhang, B. Bi-level Score Matching for Learning Energy-based Latent Variable Models. In *Advances in Neural Information Processing Systems*, volume 33, 2020.
- Che, T., Zhang, R., Sohl-Dickstein, J., Larochelle, H., Paull, L., Cao, Y., and Bengio, Y. Your GAN is Secretly an Energy-based Model and You Should Use Discriminator Driven Latent Sampling. In *Advances in Neural Information Processing Systems*, volume 33, 2020.
- Chen, R. T., Li, X., Grosse, R. B., and Duvenaud, D. K. Isolating sources of disentanglement in variational autoencoders. In *Advances in neural information processing systems*, pp. 2610–2620, 2018.
- Crawford, E. and Pineau, J. Exploiting spatial invariance for scalable unsupervised object tracking. *arXiv preprint arXiv:1911.09033*, 2019a.
- Crawford, E. and Pineau, J. Spatially invariant unsupervised object detection with convolutional neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 3412–3420, 2019b.
- Donahue, J., Krähenbühl, P., and Darrell, T. Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.
- Du, Y. and Mordatch, I. Implicit generation and generalization in energy-based models. *arXiv preprint arXiv:1903.08689*, 2019.
- Engelcke, M., Kosioruk, A. R., Jones, O. P., and Posner, I. Genesis: Generative scene inference and sampling with object-centric latent representations. *arXiv preprint arXiv:1907.13052*, 2019.
- Eslami, S. M. A., Heess, N., Weber, T., Tassa, Y., Szepesvari, D., Kavukcuoglu, K., and Hinton, G. E. Attend, infer, repeat: Fast scene understanding with generative models. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS’16, pp. 3233–3241, Red Hook, NY, USA, December 2016. Curran Associates Inc. ISBN 978-1-5108-3881-9.
- Esmaeili, B., Wu, H., Jain, S., Bozkurt, A., Siddharth, N., Paige, B., Brooks, D. H., Dy, J., and Meent, J.-W. Structured disentangled representations. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 2525–2534. PMLR, 2019.
- Gao, R., Nijkamp, E., Kingma, D. P., Xu, Z., Dai, A. M., and Wu, Y. N. Flow contrastive estimation of energy-based models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 7518–7528, 2020.
- Grathwohl, W., Wang, K.-C., Jacobsen, J.-H., Duvenaud, D., Norouzi, M., and Swersky, K. Your classifier is secretly an energy based model and you should treat it like one. *arXiv preprint arXiv:1912.03263*, 2019.
- Grathwohl, W. S., Kelly, J. J., Hashemi, M., Norouzi, M., Swersky, K., and Duvenaud, D. No {mcmc} for me: Amortized sampling for fast and stable training of energy-based models. In *International Conference on Learning Representations*, 2021.
- Gutmann, M. and Hyvärinen, A. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304. JMLR Workshop and Conference Proceedings, 2010.
- Han, T., Nijkamp, E., Fang, X., Hill, M., Zhu, S.-C., and Wu, Y. N. Divergence triangle for joint training of generator model, energy-based model, and inferential model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8670–8679, 2019.
- Han, T., Nijkamp, E., Zhou, L., Pang, B., Zhu, S.-C., and Wu, Y. N. Joint training of variational auto-encoder and latent energy-based model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S., and Lerchner, A. betavae: Learning basic visual concepts with a constrained variational framework. 2016.
- Hinton, G. E. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.
- Hinton, G. E., Osindero, S., and Teh, Y.-W. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, May 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527.
- Hyvärinen, A. and Dayan, P. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(4), 2005.

- Kim, H. and Mnih, A. Disentangling by factorising. In *International Conference on Machine Learning*, pp. 2649–2658, 2018.
- Kingma, D. P. and Welling, M. Auto-encoding variational bayes. *International Conference on Learning Representations*, 2013.
- Kosiorek, A., Kim, H., Teh, Y. W., and Posner, I. Sequential attend, infer, repeat: Generative modelling of moving objects. In *Advances in Neural Information Processing Systems*, pp. 8606–8616, 2018.
- LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., and Huang, F. A tutorial on energy-based learning. *Predicting structured data*, 1(0), 2006.
- Lin, Z., Wu, Y.-F., Peri, S., Fu, B., Jiang, J., and Ahn, S. Improving generative imagination in object-centric world models. *arXiv preprint arXiv:2010.02054*, 2020a.
- Lin, Z., Wu, Y.-F., Peri, S. V., Sun, W., Singh, G., Deng, F., Jiang, J., and Ahn, S. SPACE: Unsupervised Object-Oriented Scene Representation via Spatial Attention and Decomposition. *arXiv:2001.02407 [cs, eess, stat]*, March 2020b.
- Liu, H. and Abbeel, P. Hybrid discriminative-generative training via contrastive learning. *arXiv preprint arXiv:2007.09070*, 2020.
- McInnes, L., Healy, J., and Melville, J. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.
- Nijkamp, E., Hill, M., Han, T., Zhu, S.-C., and Nian Wu, Y. On the anatomy of mcmc-based maximum likelihood learning of energy-based models. *arXiv*, pp. arXiv–1903, 2019a.
- Nijkamp, E., Hill, M., Zhu, S.-C., and Wu, Y. N. Learning non-convergent non-persistent short-run mcmc toward energy-based model. In *Advances in Neural Information Processing Systems*, pp. 5232–5242, 2019b.
- Pang, B., Han, T., Nijkamp, E., Zhu, S.-C., and Wu, Y. N. Learning latent space energy-based prior model. *Advances in Neural Information Processing Systems*, 33, 2020.
- Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In Xing, E. P. and Jebara, T. (eds.), *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pp. 1278–1286, Beijing, China, June 2014. PMLR.
- Rockafellar, R. T. *Convex analysis*, volume 36. Princeton university press, 1970.
- Salakhutdinov, R. and Hinton, G. Deep boltzmann machines. In *Artificial intelligence and statistics*, pp. 448–455, 2009.
- Smolensky, P. Information processing in dynamical systems: Foundations of harmony theory. Technical report, Colorado Univ at Boulder Dept of Computer Science, 1986.
- Song, Y. and Kingma, D. P. How to train your energy-based models. *arXiv preprint arXiv:2101.03288*, 2021.
- Song, Y., Garg, S., Shi, J., and Ermon, S. Sliced score matching: A scalable approach to density and score estimation. In *Uncertainty in Artificial Intelligence*, pp. 574–584. PMLR, 2020.
- Tieleman, T. Training restricted boltzmann machines using approximations to the likelihood gradient. In *Proceedings of the 25th international conference on Machine learning*, pp. 1064–1071, 2008.
- Tomczak, J. and Welling, M. Vae with a vampprior. In *International Conference on Artificial Intelligence and Statistics*, pp. 1214–1223, 2018.
- Vértes, E., Unit, U. G., and Sahani, M. Learning doubly intractable latent variable models via score matching, 2016.
- Vincent, P. A connection between score matching and denoising autoencoders. *Neural computation*, 23(7):1661–1674, 2011.
- Wainwright, M. J. and Jordan, M. I. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 1(1–2):1–305, 2008. doi: 10/bpnwrm.
- Welling, M. and Teh, Y. W. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 681–688, 2011.
- Welling, M., Rosen-zvi, M., and Hinton, G. E. Exponential family harmoniums with an application to information retrieval. In Saul, L., Weiss, Y., and Bottou, L. (eds.), *Advances in Neural Information Processing Systems*, volume 17, pp. 1481–1488. MIT Press, 2004.
- Wu, H., Zimmermann, H., Sennesh, E., Le, T. A., and van de Meent, J.-W. Amortized population gibbs samplers with neural sufficient statistics. In *Proceedings of the International Conference on Machine Learning*, pp. 10205–10215, 2020.

Xie, J., Lu, Y., Zhu, S.-C., and Wu, Y. A theory of generative convnet. In *International Conference on Machine Learning*, pp. 2635–2644. PMLR, 2016.

Xie, J., Lu, Y., Gao, R., and Wu, Y. N. Cooperative learning of energy-based model and latent variable model via mcmc teaching. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Xie, J., Lu, Y., Gao, R., Zhu, S.-C., and Wu, Y. N. Cooperative Training of Descriptor and Generator Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(1):27–45, January 2020. ISSN 1939-3539. doi: 10.1109/TPAMI.2018.2879081.

Xie, J., Zheng, Z., Fang, X., Zhu, S.-C., and Wu, Y. N. Cooperative training of fast thinking initializer and slow thinking solver for conditional learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021a.

Xie, J., Zheng, Z., and Li, P. Learning energy-based model with variational auto-encoder as amortized sampler. In *The Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI)*, volume 2, 2021b.

---

# Rate-Regularization and Generalization in VAEs

---

**Alican Bozkurt\***  
Northeastern University  
alican@ece.neu.edu

**Babak Esmaeili\***  
Northeastern University  
esmaeili.b@northeastern.edu

**Jean-Baptiste Tristan**  
Boston College  
tristanj@bc.edu

**Dana H. Brooks**  
Northeastern University  
brooks@ece.neu.edu

**Jennifer G. Dy**  
Northeastern University  
jdy@ece.neu.edu

**Jan-Willem van de Meent**  
Northeastern University  
j.vandemeent@northeastern.edu

## Abstract

Variational autoencoders optimize an objective that combines a reconstruction loss (the distortion) and a KL term (the rate). The rate is an upper bound on the mutual information, which is often interpreted as a regularizer that controls the degree of compression. We here examine whether inclusion of the rate also acts as an inductive bias that improves generalization. We perform rate-distortion analyses that control the strength of the rate term, the network capacity, and the difficulty of the generalization problem. Decreasing the strength of the rate paradoxically *improves* generalization in most settings, and reducing the mutual information typically leads to underfitting. Moreover, we show that generalization continues to improve even after the mutual information saturates, indicating that the gap on the bound (i.e. the KL divergence relative to the inference marginal) affects generalization. This suggests that the standard Gaussian prior is not an inductive bias that typically aids generalization, prompting work to understand what choices of priors improve generalization in VAEs.

## 1 Introduction

Variational autoencoders (VAEs) learn representations in an unsupervised manner by training an en-

coder, which maps high-dimensional data to a lower-dimensional latent code, along with a decoder, which parameterizes a manifold that is embedded in the data space (Kingma and Welling, 2013; Rezende et al., 2014). Much of the work on VAEs has been predicated on the observation that distances on the learned manifold can reflect semantically meaningful factors of variation in the data. This is commonly illustrated by visualizing interpolations in the latent space, or more generally, interpolations along geodesics (Chen et al., 2019).

The ability of VAEs to interpolate is often attributed to the variational objective (Ghosh et al., 2019). VAEs maximize a lower bound on the log-marginal likelihood, which comprises a reconstruction loss and a Kullback-Leibler (KL) divergence between the encoder and the prior (called the rate). Minimizing the reconstruction loss in isolation is equivalent to training a deterministic autoencoder. For this reason, the rate is often interpreted as a regularizer that induces a smoother representation (Chen et al., 2016; Berthelot et al., 2018).

In this paper, we ask the question of whether the inclusion of the rate term also improves generalization. That is, does this penalty reduce the reconstruction loss for inputs that were unseen during training? A known property of VAEs is that the optimal decoder will memorize training data in the limit of infinite capacity (Alemi et al., 2018; Shu et al., 2018), as will a deterministic autoencoder (Radhakrishnan et al., 2019). At the same time, there is empirical evidence that VAEs can underfit the training data, and that reducing the strength of the rate term can mitigate underfitting (Hoffman et al., 2017). Therefore, we might hypothesize that VAEs behave like any other model in machine learning; high-capacity VAEs will overfit the training data, but we can improve generalization by adjusting the strength of the KL term to balance overfitting and underfitting.

\*Equal contribution.

Proceedings of the 24<sup>th</sup> International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130. Copyright 2021 by the author(s).

To test this hypothesis, we performed experiments that systematically vary the strength of the rate term and the network capacity. In these experiments, we deliberately focus on comparatively simple network architectures in the form of linear and convolutional layers with standard spherical Gaussian priors. These architectures remain widely used in work on VAEs, particularly work that focuses on disentangled representations, and systematically investigating these cases provides us with results that can form a basis for understanding the wide variety of more sophisticated architectures that exist in the literature.

The primary aim of our experiments is to carefully control the difficulty of the generalization problem. Our goal in doing so is to disambiguate between apparent generalization that can be achieved by simply reconstructing the most similar memorized training examples and generalization that requires reconstruction of examples that differ substantially from those seen in the training set. To achieve this goal, we have created a dataset of J-shaped tetrominoes that vary in color, size, position, and orientation. This dataset gave us a sufficient variation of both the amount of training data and the density of data in the latent space, as well as sufficient sensitivity of reconstruction loss to variation in these factors, in order to evaluate out-of-domain generalization to unseen combinations of factors.

The surprising outcome of our experiments is that the rate term does not, in general, improve generalization in terms of the reconstruction loss. We find that VAEs memorize training data in practice, even for simple 3-layer fully-connected architectures. However, contrary to intuition, *reducing* the strength of the rate term *improves* generalization under most conditions, including in out-of-domain generalization tasks. The only case where an optimum level of rate-regularization emerges is when low-capacity VAEs are trained on data that are sparse in the latent space. We show that these results hold for both MLP and CNN-based architectures, as well as a variety of datasets.

These results suggest that we need to more carefully quantify the effect of each term in the VAE objective on the generalization properties of the learned representation. To this end, we decompose the KL divergence between the encoder and the prior into its constituent terms: the mutual information (MI) between data and the latent code and the KL divergence between the inference marginal and the prior. We find that the MI term saturates as we reduce the strength of the rate term, which indicates that it is in fact the KL between the inference marginal and prior that drives improvements in generalization in high-capacity models. This suggests that the standard spherical Gaussian prior in VAEs is not an inductive bias that aids generalization

in most cases, and that more flexible learned priors may be beneficial in this context.

## 2 Variational Autoencoders

VAEs jointly train a generative model  $p_\theta(\mathbf{x}, \mathbf{z})$  and an inference model  $q_\phi(\mathbf{x}, \mathbf{z})$ . The generative model comprises a prior  $p(\mathbf{z})$ , typically a spherical Gaussian, and a likelihood  $p_\theta(\mathbf{x} | \mathbf{z})$  that is parameterized by a neural network known as the decoder. The inference model is defined in terms of a variational distribution  $q_\phi(\mathbf{z} | \mathbf{x})$ , parameterized by an encoder network, and a data distribution  $q(\mathbf{x})$ , which is typically an empirical distribution  $q(\mathbf{x}) = \frac{1}{N} \sum_n \delta_{\mathbf{x}_n}(\mathbf{x})$  over training data  $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ . The two models are optimized by maximizing a variational objective (Higgins et al., 2017)

$$\begin{aligned} \mathcal{L}_\beta(\theta, \phi) = & \mathbb{E}_{q_\phi(\mathbf{z}, \mathbf{x})} [\log p_\theta(\mathbf{x} | \mathbf{z})] \\ & - \beta \mathbb{E}_{q(\mathbf{x})} [\text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p(\mathbf{z}))]. \end{aligned} \quad (1)$$

The multiplier  $\beta$ , which in a standard VAE is set to 1, controls the relative strength of the reconstruction loss and the KL loss. We will throughout this paper refer to these two terms  $\mathcal{L}_\beta = -D - \beta R$  as the distortion  $D$  and the rate  $R$ . The distortion defines a reconstruction loss, whereas the rate constrains the encoder distribution  $q_\phi(\mathbf{z} | \mathbf{x})$  to be similar to the prior  $p(\mathbf{z})$ . As  $\beta$  approaches 0, the VAE objective becomes similar to that of a deterministic autoencoder; in absence of the rate term, the distortion is minimized when the encoding is a delta-peaked at the maximum-likelihood value  $\text{argmax}_{\mathbf{z}} \log p(\mathbf{x} | \mathbf{z})$ . For this reason, a standard interpretation is that the rate serves to induce a smoother representation and ensures that samples from the generative model are representative of the data.

While there is evidence that the rate term indeed induces a smoother representation (Shamir et al., 2010), it is not clear whether this smoothness mitigates overfitting, or indeed to what extent VAEs are prone to overfitting in the first place. Several researchers (Bousquet et al., 2017; Rezende and Viola, 2018; Alemi et al., 2018; Shu et al., 2018) have pointed out that an infinite-capacity optimal decoder will memorize training data, which suggests that high-capacity VAEs will overfit. On the other hand, there is also evidence of underfitting; setting  $\beta < 1$  can improve the quality of reconstructions in VAEs for images (Hoffman et al., 2017; Engel et al., 2017), natural language (Wen et al., 2017), and recommender systems (Liang et al., 2018).

More broadly, precisely what constitutes generalization and overfitting in this model class is open to interpretation. If we view the VAE objective primarily as a means of training a generative model, then it makes sense to evaluate model performance in terms of the log marginal likelihood  $\log p_\theta(\mathbf{x})$ . This view is coherent

for the standard VAE objective ( $\beta = 1$ ), which defines a lower bound

$$\begin{aligned}\mathcal{L}(\theta, \phi) &= \mathbb{E}_{q(\mathbf{x})} [\log p_\theta(\mathbf{x}) - \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x}))] \\ &\leq \mathbb{E}_{q(\mathbf{x})} [\log p_\theta(\mathbf{x})].\end{aligned}$$

The KL term indirectly regularizes the generative model when the encoder capacity is constrained (Shu et al., 2018). Note however that  $\mathcal{L}_\beta$  is not a lower bound on  $\log p_\theta(\mathbf{x})$  when  $\beta < 1$ . This means that it does not make sense to evaluate generalization in terms of  $\log p_\theta(\mathbf{x})$  when  $\beta \rightarrow 0$ , or in deterministic autoencoders that do not define a generative model to begin with.

In this paper, we view the VAE primarily as a model for learning representations in an unsupervised manner. In this view, generation is more ancillary; The encoder and decoder serve to define a lossy compressor and decompressor, or equivalently to define a low-dimensional manifold that is embedded in the data space. Our hope is that the learned latent representation reflects semantically meaningful factors of variation in the data, whilst discarding nuisance variables.

The view of VAEs as lossy compressors can be formalized by interpreting the objective  $\mathcal{L}_\beta$  as a special case of information-bottleneck (IB) objectives (Tishby et al., 2000; Alemi et al., 2017, 2018). This interpretation relies on the observation that the decoder  $p_\theta(\mathbf{x} | \mathbf{z})$  defines a lower bound on the MI in the inference model  $q_\phi(\mathbf{z}, \mathbf{x})$  in terms of a distortion  $D$  and entropy  $H$

$$H - D \leq I_q[\mathbf{x}; \mathbf{z}], \quad (2)$$

$$D = -\mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} [\log p_\theta(\mathbf{x} | \mathbf{z})], \quad (3)$$

$$H = -\mathbb{E}_{q(\mathbf{x})} [\log q(\mathbf{x})]. \quad (4)$$

Similarly, the rate  $R$  is an upper bound on this same mutual information  $R \geq I_q[\mathbf{x}; \mathbf{z}]$ ,

$$\begin{aligned}R &= \mathbb{E}_{q_\phi(\mathbf{x}, \mathbf{z})} [\text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p(\mathbf{z}))] \\ &= I_q[\mathbf{x}; \mathbf{z}] + \text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z})).\end{aligned} \quad (5)$$

Here the term  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  is sometimes called “the marginal KL” in the literature (Rezende and Viola, 2018). The naming of the rate and distortion terms originates from rate-distortion theory (Cover and Thomas, 2012), which seeks to minimize  $I_q[\mathbf{x}; \mathbf{z}]$  subject to the constraint  $D \leq D^*$ . The connection to VAEs now arises from the observation that  $\mathcal{L}_\beta$  is a Lagrangian relaxation of the rate-distortion objective

$$\mathcal{L}_\beta = -D - \beta R. \quad (6)$$

The appeal of this view is that it suggests an interpretation of the distortion  $D$  as an empirical risk and of  $I_q[\mathbf{x}; \mathbf{z}]$  as a regularizer (Shamir et al., 2010). This leads to the hypothesis that VAEs may exhibit a classic

bias-variance trade-off: In the limit  $\beta \rightarrow 0$ , we may expect low distortion on the training set but poor generalization to the test set, whereas increasing  $\beta$  may mitigate this form of overfitting.

At the same time, the rate-distortion view of VAEs gives rise to some peculiarities. Standard IB methods use a regressor or classifier  $p_\theta(\mathbf{y} | \mathbf{x})$  to define a lower bound  $H - D \leq I_q[\mathbf{y}; \mathbf{z}]$  on the MI between the code  $\mathbf{z}$  and a target variable  $\mathbf{y}$  (Tishby et al., 2000). The objective is to maximize  $I_q[\mathbf{y}; \mathbf{z}]$ , which serves to learn a representation  $\mathbf{z}$  which is predictive of  $\mathbf{y}$ , whilst minimizing  $I_q[\mathbf{x}; \mathbf{z}]$ , which serves to compress  $\mathbf{x}$  by discarding information irrelevant to  $\mathbf{y}$ . However, this interpretation does not translate to the special case of VAEs, where  $\mathbf{x} = \mathbf{y}$ . Here any compression will necessarily increase the distortion since  $D \geq H - I_q[\mathbf{x}; \mathbf{z}]$ .

In our experiments, we will explicitly investigate to what extent  $\beta$  controls a trade-off between overfitting and underfitting. To do so, we will compute *RD* curves that track the rate and distortion under varying  $\beta$ . While *RD* curves have been used to evaluate model performance on the training set (Alemi et al., 2018; Rezende and Viola, 2018), we are not aware of work that explicitly probes generalization to a test set.

To see how overfitting and underfitting may manifest in this analysis, we can consider the hypothetical case of infinite-capacity encoders and decoders. For such networks, both bounds will be tight at the optimum and  $\mathcal{L}_\beta = (1 - \beta)I_q[\mathbf{x}; \mathbf{z}] - H$ . Maximizing  $\mathcal{L}_\beta$  with respect to  $\phi$  will lead to an *autodecoding* limit when  $\beta > 1$ , which minimizes  $I_q[\mathbf{x}; \mathbf{z}]$ , and an *autoencoding* limit when  $\beta < 1$ , which maximizes  $I_q[\mathbf{x}; \mathbf{z}]$  (Alemi et al., 2018). One hypothesis is that we will observe poor generalization to the test set in either limit, since maximizing  $I_q[\mathbf{x}; \mathbf{z}]$  could lead to overfitting whereas minimizing  $I_q[\mathbf{x}; \mathbf{z}]$  could lead to underfitting. Moreover, an infinite-capacity generator will fully memorize the training data, which could lead to poor generalization performance in terms of the log marginal likelihood.

In practice, it may well be that the decoder  $p_\theta(\mathbf{x} | \mathbf{z})$  can be approximated as an infinite-capacity model. We present empirical evidence of this phenomenon in Appendix ?? that is consistent with recent analyses (Bousquet et al., 2017; Rezende and Viola, 2018; Alemi et al., 2018; Shu et al., 2018). However, it is typically not the case that the prior  $p(\mathbf{z})$  has a high capacity. In fact, a standard spherical Gaussian prior effectively has 0 capacity, since its mean and variance define an affine transformation that can be trivially absorbed into the first linear layer of any encoder and decoder. This means that the upper bound will be loose and that the rate  $R$  may in practice represent a trade-off between  $I_q[\mathbf{x}; \mathbf{z}]$  and  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$ , at least when



Figure 1: We simulate 164k tetrominoes that vary in position, orientation, size, and color.

the encoder capacity is limited. We present evidence of this trade-off in Section 4.5.

### 3 Related Work

**Generalization in VAEs.** Recent work that evaluates generalization in VAEs has primarily considered this problem from the perspective of VAEs as generative models. Shu et al. (2018) consider whether constraining encoder capacity can serve to mitigate data memorization, whereas Zhao et al. (2018) ask whether VAEs can generate examples that deviate from training data. Kumar and Poole (2020) derive a deterministic approximation to the  $\beta$ -VAE objective and show that  $\beta$ -VAE regularizes the generative by imposing a constraint on the Jacobian of the encoder. Whereas Kumar and Poole (2020) evaluate generalization in terms of FID scores (Heusel et al., 2017), we here focus on *RD* curves. Huang et al. (2020) also discuss evaluating deep generative models based on *RD* curves. They show that this type of analysis can be used to uncover some of the known properties of VAEs such as the “holes problem” (Rezende and Viola, 2018) by tracking the change in the curve for different sizes of latent space. In our work, we focus on the change of the *RD* curve as the generalization problem becomes more difficult.

**Generalization and regularization in deterministic autoencoders.** Zhang et al. (2019) and Radhakrishnan et al. (2019) study generalization in deterministic autoencoders, showing that these models can memorize training data if they are over-parameterized. We overall observed a similar behaviour in our experiments. However, for our experiments, we did not consider architectures as deep as the ones in Zhang et al. (2019) and Radhakrishnan et al. (2019). Ghosh et al. (2019) show that combining deterministic autoencoders with regularizers other than the rate can lead to competitive generative performance.

**Generalization of disentangled representations.** Our work is indirectly related to research on disentangled representations, in the sense that some of this work is motivated by the desire to learn representations that can generalize to unseen combinations of factors (Narayanaswamy et al., 2017; Kim and Mnih, 2018; Esmaeili et al., 2019; Chen et al., 2018; Locatello et al., 2019). There has been some work to quantify the

effect of disentangling on generalization (Eastwood and Williams, 2018; Esmaeili et al., 2019; Locatello et al., 2019), but the extent of this effect remains poorly understood. In this paper, we explicitly design our experiments to test generalization to data with unseen combinations of factors, but we are not interested in disentanglement per se.

## 4 Experiments

To quantify the effect of rate-regularization on generalization, we designed a series of experiments that systematically control three factors in addition to the  $\beta$ -coefficient: the amount of training data, the density of training data relative to the true factors of variation, and the depth of the encoder and decoder networks. To establish baseline results, we begin with experiments that vary all three factors in fully-connected architectures on a simulated dataset of Tetrominoes. We additionally consider convolutional architectures, as well as other simulated and non-simulated datasets.

### 4.1 Tetrominoes Dataset

When evaluating generalization we have two primary requirements for a dataset. The first is that failures in generalization should be easy to detect. A good way to ensure this is to employ data for which we can achieve high-quality reconstructions for training examples, which makes it easier to identify degradations for test examples. The second requirement is that we need to be able to disambiguate effects that arise from a lack of data from those that arise from the difficulty of the generalization problem. When a dataset comprises a small number of examples, this may not suffice to train an encoder and decoder network. Conversely, even when employing a large training set, a network may not generalize when there are a large number of generative factors.

To satisfy both requirements, we begin with experiments on simulated data. This ensures that we can explicitly control the density of data in the space of generative factors, and that we can easily detect degradations in reconstruction quality. We initially considered the dSprites dataset (Matthey et al., 2017), which contains 3 shapes at 6 scales, 40 orientations, and  $32^2$  positions. Unfortunately, shapes in this dataset are close to convex. Varying either the shape or the rota-

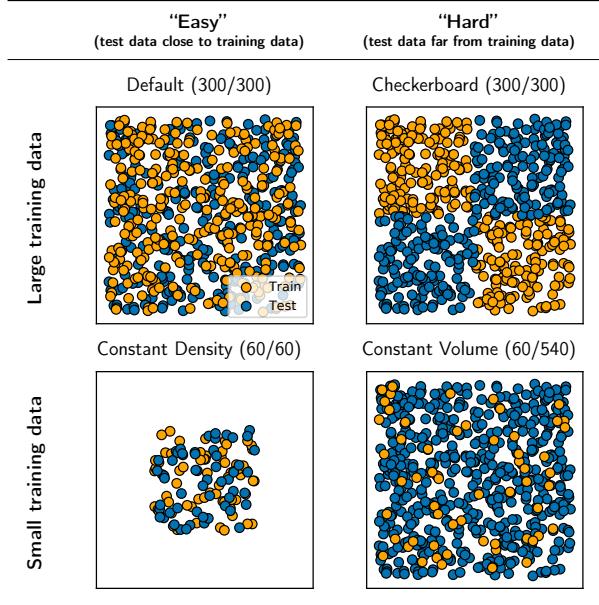


Figure 2: We define 4 train/test splits, which vary in the amount of training data and the typical distance between test data and their nearest neighbors in the training set. Here we show 600 samples with 2 generative factors for visualization.

tion results in small deviations in pixel space, which in practice makes it difficult to evaluate whether a model memorizes the training data.

To overcome this limitation, we created the Tetrominoes dataset. This dataset comprises 163,840 procedurally generated  $32 \times 32$  color images of a J-shaped tetromino, which is concave and lacks rotational symmetry. We generate images based on five i.i.d. continuous generative factors, which are sampled uniformly at random: rotation (sampled from the  $[0.0, 360.0]$  range), color (hue, sampled from  $[0.0, 0.875]$  range), scale (sampled from  $[2.0, 5.0]$  range), and horizontal and vertical position (sampled from an adaptive range to ensure no shape is placed out of bounds). To ensure uniformity of the data in the latent space, we generate a stratified sample; we divide each feature range into bins and sample uniformly within bins. Examples from the dataset are shown in Figure 1.

## 4.2 Train/Test Splits

In our experiments, we compare 4 different train/test splits that are designed to vary two components: (1) the amount of training data, (2) the typical distance between training and test examples.

1. *50/50 random split (Default).* The base case in our analysis (Figure 2, 1<sup>st</sup> from left) is a 82k/82k random train/test split of the full dataset. This case is designed

to define an “easy” generalization problem, where similar training examples will exist for most examples in the test set.

2. *Large data, (Checkerboard) split.* We create a 82k/82k split in which a 5-dimensional “checkerboard” mask partitions the training and test set (Figure 2, 2<sup>nd</sup> from left). This split has the same amount of training data as the base case, as well as the same (uniform) marginal distribution for each of the feature values. This design ensures that for any given test example, there are 5 training examples that differ in one feature (e.g. color) but are similar in all other features (e.g. position, size, and rotation). This defines an out-of-domain generalization task, whilst at the same time ensuring that the model does not need to extrapolate to unseen feature values.

3. *Small data, constant density (CD).* We create train/test splits for datasets of  $\{8k, 16k, 25k, 33k, 41k, 49k, 57k, 65k\}$  examples by constraining the range of feature values (Figure 2, 2<sup>nd</sup> from right), ensuring that the density in the feature space remains constant as we reduce the amount of data.

4. *Small data, constant volume (CV).* Finally, we create train/test splits by selecting  $\{8k, 16k, 25k, 33k, 41k, 49k, 57k, 65k\}$  training examples at random without replacement (Figure 2, 1<sup>st</sup> from right). This reduces the amount of training data whilst keeping the volume fixed, which increases the typical distance between training and test examples.

## 4.3 Network Architectures and Training

We use ReLU activations for both fully-connected and convolutional networks with a Bernoulli likelihood in the decoder<sup>1</sup>. We use a 10-dimensional latent space and assume a spherical Gaussian prior. All models are trained for 257k iterations with Adam using a batch size of 128, with 5 random restarts. For MLP architectures, we keep the number of hidden units fixed to 512 across layers. For the CNN architectures, we use 64 channels with kernel size 4 and stride 2 across layers. See Appendix ?? for further details.

## 4.4 Results

**Fully-Connected Architectures on Dense and Sparse Data.** We begin with a comparison between 1-layer and 3-layer fully-connected architectures on a dense CV (82k/82k) split and a sparser CV (16k/147k) split. Based on existing work (Radhakrishnan et al., 2019), our hypothesis in this experiment is that the 3-

<sup>1</sup>The Bernoulli likelihood is a very common choice in the VAE literature even for input domain of  $[0, 1]$ . For a more detailed discussion, see Appendix ??

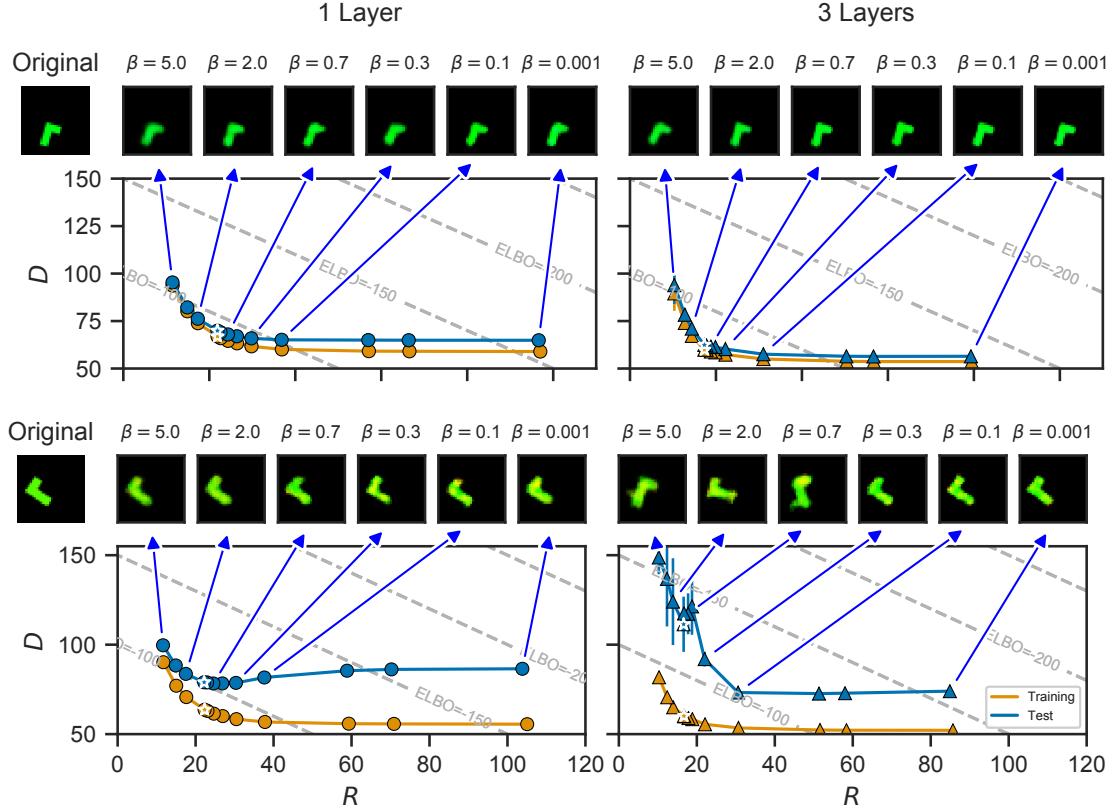


Figure 3: Training and test  $RD$  curves evaluated on the  $CV(82k/82k)$  split (*Top*) and  $CV(16k/147k)$  split (*bottom*), for a 1-layer and a 3-layer architecture. Each dot constitutes a  $\beta$  value (white stars indicate the  $\beta=1$ ), averaged over 5 restarts. Images show reconstructions of a test example.

layer architecture will be more prone to overfitting the training data (particularly in the sparser case), and our goal is to establish to what extent rate-regularization affects the degree of overfitting.

Figure 3 shows  $RD$  curves on the training and test set. We report the mean across 5 restarts, with bars indicating the standard deviation, for 12  $\beta$  values<sup>2</sup>. White stars mark the position of the standard VAE ( $\beta=1$ ) on the  $RD$  plane. Diagonal lines show iso-contours of the evidence lower bound  $\mathcal{L}_{\beta=1} = -D - R$ . Above each panel, we show reconstructions for a test-set example that is difficult to reconstruct, in the sense that it falls into the 90<sup>th</sup> percentile in terms of the  $\ell_2$ -distance between its nearest neighbor in the training set.

For the dense  $CV$  (82k/82k) split (*top*), we observe no evidence of memorization. Moreover, increasing model capacity uniformly improves generalization, in the sense that it decreases both the rate and the distortion, shifting the curve to the bottom left.

For the sparse  $CV$  (16k/147k) split (*bottom*), we see

<sup>2</sup> $\beta \in \{0.001, 0.005, 0.01, 0.1, 0.3, 0.5, 0.7, 0.9, 1, 2, 3, 5\}$

a different pattern. In the 1-layer model, we observe a trend that appears consistent with a classic bias-variance trade-off. The distortion on the training set decreases monotonically as we reduce  $\beta$ , whereas the distortion on the test set initially decreases, achieves a minimum, and somewhat increases afterwards. This suggests that  $\beta$  may control a trade-off between overfitting and underfitting, although there is no indication of data memorization. When we perform early stopping (see Appendix ??), the RD curve once again becomes monotonic, which is consistent with this interpretation in terms of overfitting.

In the 3-layer architecture, we observe a qualitatively different trend. Here we see evidence of data memorization; some reconstructions resemble memorized neighbors in the training set. However, counterintuitively, no memorization is apparent at smaller  $\beta$  values. When looking at the iso-contours, we observe that the test-set lower bound  $\mathcal{L}_{\beta=1} \leq \log p_\theta(\mathbf{x})$  achieves a maximum at  $\beta = 0.1$ . Additional analysis (see Appendix ??) shows that this maximum also corresponds to the maximum of the log marginal likelihood  $\log p_\theta(\mathbf{x})$ . In short, high-capacity networks are capable of memorizing the

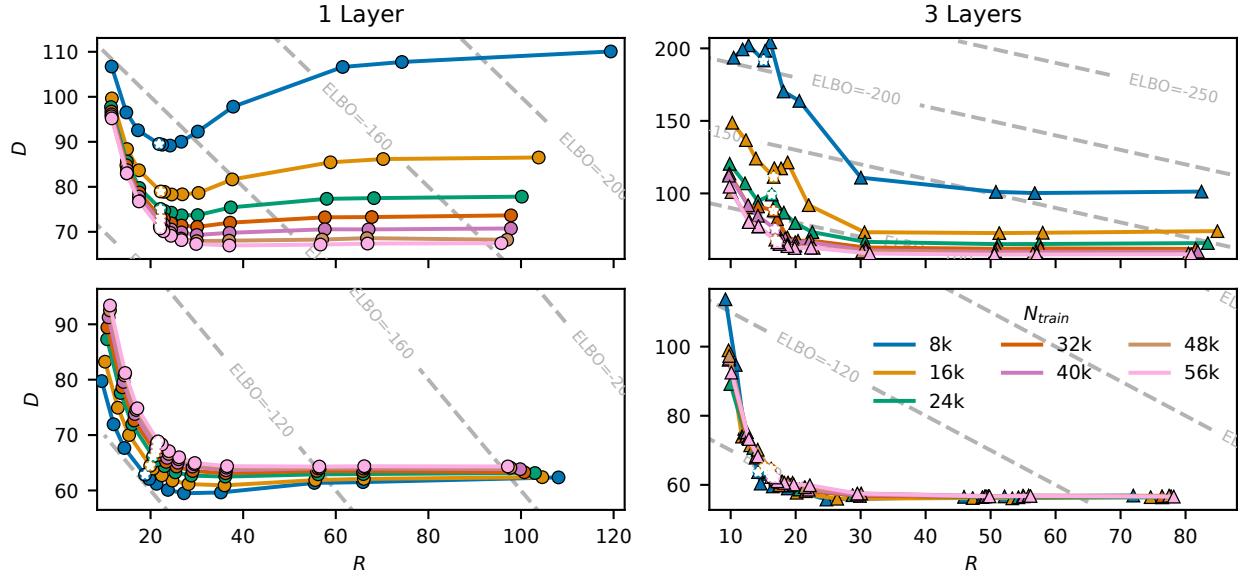


Figure 4: Test-set  $RD$  curves for constant volume (*top*) and constant density splits (*bottom*) with varying training set sizes. White stars indicate the  $RD$  value for a standard VAE ( $\beta=1$ ).

training data, as expected. However, paradoxically, this memorization occurs when  $\beta$  is large, where we would expect underfitting based on the 1-layer results, and the generalization gap, in terms of both  $D$  and  $\log p_\theta(\mathbf{x})$ , is smallest at  $\beta = 0.1$ .

**Role of the Training Set Size.** The qualitative discrepancy between training and test set  $RD$  curves in Figure 3 has to our knowledge not previously been reported. One possible reason for this is that this behavior would not have been apparent in other experiments; there is virtually no generalization gap in the dense CV (82k/82k) split. The differences between 1-layer and 3-layer architectures become visible in the sparse CV (16k/147k) split. Whereas the dense CV (82k/82k) split is representative of typically simulated datasets in terms of the number of examples and density in the latent space, the CV (16k/147k) split has a training set

that is tiny by deep learning standards. Therefore, we need to verify that the observed effects are not simply attributable to the size of the training set.

To disambiguate between effects that arise from the size of the data and effects that arise due to the density of the data, we compare CV and CD splits with training set sizes  $N_{\text{train}} = \{8\text{k}, 16\text{k}, 32\text{k}, 56\text{k}\}$ . Since CD splits have a fixed density rather than a fixed volume, the examples in the test set will be closer to their nearest neighbors in the training set, resulting in an easier generalization problem.

Figure 4 shows the test-set  $RD$  curves for this experiment. In the CV splits, the qualitative discrepancy between 1-layer and 3-layer networks becomes more pronounced as we decrease the size of the training set. However, in the CD splits, discrepancies are much less pronounced.  $RD$  curves for 3-layer networks are virtu-

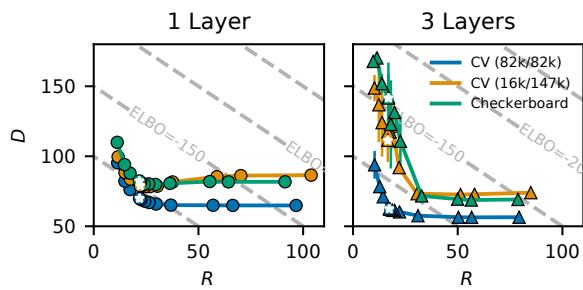


Figure 5:  $RD$  Curves for the CV(82k/82k), CV(16k/147k), and Checkerboard Splits.

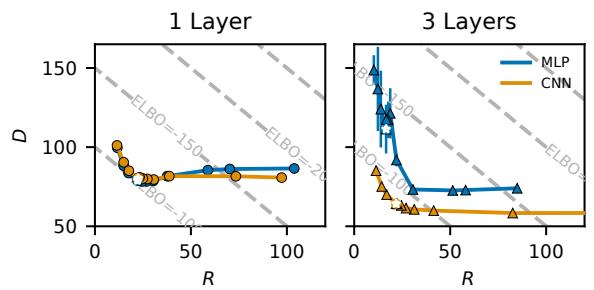


Figure 6:  $RD$  Curves for the CV(16k/147k) for MLP and CNN Architectures.

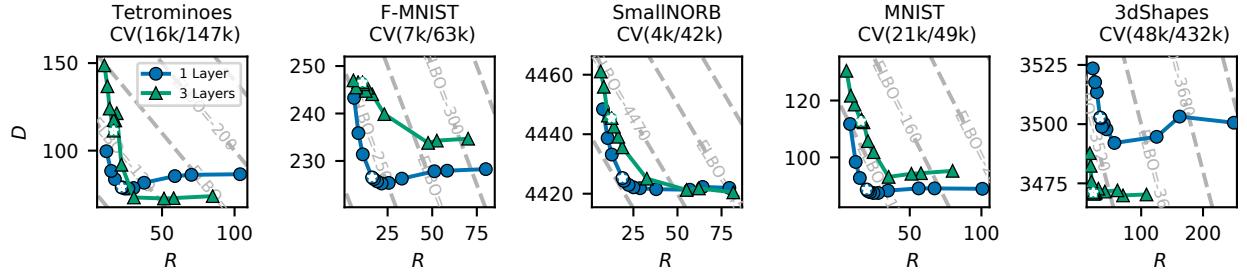


Figure 7: RD curves shown on various datasets trained with 1 and 3 layers.

ally indistinguishable.  $RD$  curves for 1-layer networks still exhibit a minimum, but there is a much weaker dependence on the training set size. Moreover, generalization performance marginally improves as we decrease the size of the training set. This may be attributable to the manner in which we construct the splits. Because we simulate data using a 5-dimensional hypercube of generative factors, limiting the volume has the effect of decreasing the surface to volume ratio, which would mildly reduce the typical distance between training and test set examples.

**In-Sample and Out-of-Sample Generalization.** A possible takeaway from the results in Figure 4 is that the amount of training data itself does not strongly affect generalization performance, but that the similarity between test and training set examples does. To further test this hypothesis, we compare the CV (82k/82k) and CV (16k/147k) splits to the Checkerboard (82k/82k) split, which allows us to evaluate out-of-sample generalization to unseen combinations of factors.  $RD$  curves in Figure 5 show similar generalization performance for the Checkerboard and CV (16k/147k) splits. This is consistent with the fact that these splits have a similar distribution over pixel-distances between test set and nearest training set examples (Figure ??).

**Convolutional architectures.** A deliberate limitation of our experiments is that we have considered fully-connected networks, which are an extremely simple architecture. There are of course many other encoder and decoder architectures for VAEs (Kingma et al., 2016; Gulrajani et al., 2017; Van den Oord et al., 2016). In Figure 6, we compare  $RD$  curves for MLPs with those for 1-layer and 3-layer CNNs (see Table ?? for details). We observe a monotonic curve for 3-layer CNNs and only a small degree of non-monotonicity in the 1-layer CNN. Since most architectures will have a higher capacity than a 3-layer MLP or CNN, we can interpret the results for 3-layer networks as the most representative of other architectures.

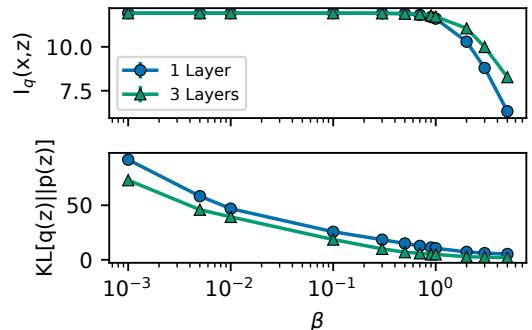
**Additional Datasets.** Our analysis thus far shows that the generalization gap grows when we increase

the difficulty of a generalization problem, which is expected. The unexpected result is that, depending on model capacity, we either observe U-shaped  $RD$  curves that are consistent with a bias-variance trade-off, or L-shaped curves in which generalization improves as we reduce  $\beta$ . To test whether both phenomena also occur in other datasets, we perform experiments on the Fashion-MNIST (Xiao et al., 2017), SmallNORB (LeCun et al., 2004), MNIST (LeCun et al., 1998), and 3dShapes (Burgess and Kim, 2018) datasets.

We show the full results of this analysis for a range of CV splits in Appendix ???. In Figure 7 we compare 1-layer and 3-layer networks for a single split with a small training set for each dataset. We see that the  $RD$  curves for the 1-layer network exhibits a local minimum in most datasets. Curves for the 3-layer network are generally closer to monotonic, although a more subtle local minimum is visible in certain cases. The one exception is the 3dShapes dataset, where the 3-layer network exhibits a more pronounced local minimum than the 1-layer network.

#### 4.5 Is the Rate a Regularizer?

Our experiments suggest that the rate is not an inductive bias that typically reduces the reconstruction loss in high-capacity models. One possible explanation for these findings is that we should consider both terms in


 Figure 8:  $I_q(\mathbf{x}, \mathbf{z})$  and  $KL(q_\phi(\mathbf{z}) \parallel p(\mathbf{z}))$  vs  $\beta$  for  $\beta$ -VAE Trained on CV(16k/147k).

the rate  $R = I_q(\mathbf{x}; \mathbf{z}) + \text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  when evaluating the effect of rate-regularization. The term  $I_q(\mathbf{x}; \mathbf{z})$  admits a clear interpretation as a regularizer (Shamir et al., 2010). However,  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  is not so much a regularizer as a constraint that the aggregate posterior  $q_\phi(\mathbf{z})$  should resemble the prior  $p(\mathbf{z})$ , which may require a less smooth encoder and decoder when learning a mapping from a multimodal data distribution to a unimodal prior. While we have primarily concerned ourselves with continuous factors for a single Tetramino shape, it is of course common to fit VAEs to multimodal data, particularly when the data contains distinct classes. A unimodal prior forces the VAE to learn a decoder that “partitions” the contiguous latent space into regions associated with each class, which will give rise to sharp gradients near class boundaries.

To understand how each of these two terms contributes to the rate, we compute estimates of  $I_q(\mathbf{x}; \mathbf{z})$  and  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  by approximating  $q_\phi(\mathbf{z})$  with a Monte Carlo estimate over batches of size 512 (see Esmaeili et al. (2019)). Figure 8 shows both estimates as a function of  $\beta$  for the CV (16k/147k) split. As expected,  $I_q(\mathbf{x}; \mathbf{z})$  decreases when  $\beta > 1$  but saturates to its maximum  $\log N_{\text{train}}$  when  $\beta < 1$ . Conversely, the term  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  is small when  $\beta > 1$  but increases when  $\beta < 1$ . Based on the fact that the generalization gap in terms of both the reconstruction loss and  $\log p_\theta(\mathbf{x})$  is minimum at  $\beta = 0.1$ , it appears that the  $\text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$  term can have a significant effect on generalization performance. Additional experiments where we train VAEs with either the marginal KL or the MI term removed from the loss function confirm this effect of the marginal KL term on the generalization performance of VAEs (see Appendix ??).

Our reading of these results is that it is reasonable to interpret the rate as an approximation of the MI when  $\beta$  is large. However, our experiments suggest that VAEs typically underfit in this regime, and therefore do not benefit from this form of regularization. When  $\beta$  is small, the MI saturates and we can approximate the rate as  $R = \log N_{\text{train}} + \text{KL}(q_\phi(\mathbf{z}) \| p(\mathbf{z}))$ . In this regime, we should not interpret the rate as a regularizer, but as a constraint on the learned representation, and there can be a trade-off between this constraint and the reconstruction accuracy.

## 5 Discussion

In this empirical study, we trained over 6000 VAE instances to evaluate how rate-regularization in the VAE objective affects generalization to unseen examples. Our results demonstrate that high-capacity VAEs can and do overfit the training data. However, paradoxically, memorization effects can be mitigated by

decreasing  $\beta$ . These effects are more pronounced when test-set examples differ substantially from their nearest neighbors in the training set. For real-world datasets, this is likely to be the norm rather than the exception; few datasets have a small number of generative factors.

Based on these results, we argue that we should give the role of priors as inductive biases in VAEs more serious consideration. The KL relative to a standard Gaussian prior does not improve generalization performance in the majority of cases. With the benefit of hindsight, this is unsurprising; When we use a VAE to model a fundamentally multimodal data distribution, then mapping this data onto a contiguous unimodal Gaussian prior may not yield a smooth encoder, semantically meaningful distances in the latent space, or indeed a representation that generalizes to unseen data. This motivates future work to determine to what extent other priors, including priors that attempt to induce structured or disentangled representations, can aid generalization performance.

While these experiments are comprehensive, we have explicitly constrained ourselves to comparatively simple architectures and datasets. These architectures are not representative of the state of the art (Vahdat and Kautz; Maaløe et al., 2019; Razavi et al., 2019; Gulrajani et al., 2017; Van den Oord et al., 2016), particularly when we are primarily interested in generation. It remains an open question to what extent rate-regularization affects generalization in much higher-capacity architectures that are trained on larger datasets of natural images. Moreover, there are other factors that could potentially impact our results which we do not study here, including but not limited to: dimensionality of the latent space, the choice of prior, and the choice of training method. We leave the investigation of these factors in *RD* analysis to future work.

## Acknowledgements

We would like to thank reviewers of a previous version of this manuscript for their detailed comments, as well as Sarthak Jain and Heiko Zimmermann for helpful discussions. This project was supported by the Intel Corporation, the 3M Corporation, the Air Force Research Laboratory (AFRL) and DARPA, NSF grant 1901117, NIH grant R01CA199673 from NCI, and startup funds from Northeastern University.

## References

- A. Aleji, B. Poole, I. Fischer, J. Dillon, R. A. Saurous, and K. Murphy. Fixing a broken ELBO. In *International Conference on Machine Learning*, pages 159–168, 2018.
- A. A. Aleji, I. Fischer, J. V. Dillon, and K. Murphy.

- Deep Variational Information Bottleneck. *International Conference on Learning Representations*, 2017.
- D. Berthelot, C. Raffel, A. Roy, and I. Goodfellow. Understanding and improving interpolation in autoencoders via an adversarial regularizer. *arXiv preprint arXiv:1807.07543*, 2018.
- O. Bousquet, S. Gelly, I. Tolstikhin, C.-J. Simon-Gabriel, and B. Schoelkopf. From optimal transport to generative modeling: the VEGAN cookbook. *arXiv preprint arXiv:1705.07642*, 2017.
- C. Burgess and H. Kim. 3D shapes dataset. <https://github.com/deepmind/3dshapes-dataset/>, 2018.
- N. Chen, F. Ferroni, A. Klushyn, A. Paraschos, J. Bayer, and P. van der Smagt. Fast approximate geodesics for deep generative models. In *International Conference on Artificial Neural Networks*, pages 554–566. Springer, 2019.
- T. Q. Chen, X. Li, R. B. Grosse, and D. K. Duvenaud. Isolating sources of disentanglement in variational autoencoders. In *Advances in Neural Information Processing Systems*, pages 2610–2620, 2018.
- X. Chen, D. P. Kingma, T. Salimans, Y. Duan, P. Dhariwal, J. Schulman, I. Sutskever, and P. Abbeel. Variational lossy autoencoder. *arXiv preprint arXiv:1611.02731*, 2016.
- T. M. Cover and J. A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
- C. Eastwood and C. K. I. Williams. A Framework for the Quantitative Evaluation of Disentangled Representations. In *International Conference on Learning Representations*, Feb. 2018.
- J. Engel, M. Hoffman, and A. Roberts. Latent Constraints: Learning to Generate Conditionally from Unconditional Generative Models. *arXiv:1711.05772 [cs, stat]*, Nov. 2017.
- B. Esmaeili, H. Wu, S. Jain, A. Bozkurt, N. Siddharth, B. Paige, D. H. Brooks, J. Dy, and J.-W. van de Meent. Structured disentangled representations. In K. Chaudhuri and M. Sugiyama, editors, *Proceedings of Machine Learning Research*, volume 89 of *Proceedings of Machine Learning Research*, pages 2525–2534. PMLR, 16–18 Apr 2019.
- P. Ghosh, M. S. Sajjadi, A. Vergari, M. Black, and B. Schölkopf. From variational to deterministic autoencoders. *arXiv preprint arXiv:1903.12436*, 2019.
- I. Gulrajani, K. Kumar, F. Ahmed, A. A. Taiga, F. Visin, D. Vazquez, and A. Courville. Pixelvae: A latent variable model for natural images. In *International Conference on Representations*, 2017.
- M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In *Advances in neural information processing systems*, pages 6626–6637, 2017.
- I. Higgins, L. Matthey, A. Pal, C. Burgess, X. Glorot, M. Botvinick, S. Mohamed, and A. Lerchner. beta-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017.
- M. D. Hoffman, C. Riquelme, and M. J. Johnson. The  $\beta$ -VAE’s Implicit Prior. In *Workshop on Bayesian Deep Learning, NIPS*, pages 1–5, 2017.
- S. Huang, A. Makhzani, Y. Cao, and R. Grosse. Evaluating lossy compression rates of deep generative models. *arXiv preprint arXiv:2008.06653*, 2020.
- H. Kim and A. Mnih. Disentangling by factorising. In *International Conference on Machine Learning*, pages 2654–2663, 2018.
- D. P. Kingma and M. Welling. Auto-encoding variational bayes. *International Conference on Learning Representations*, 2013.
- D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improved variational inference with inverse autoregressive flow. In *Advances in neural information processing systems*, pages 4743–4751, 2016.
- A. Kumar and B. Poole. On implicit regularization in  $\beta$ -vaes. *arXiv preprint arXiv:2002.00041*, 2020.
- Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- Y. LeCun, F. J. Huang, and L. Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 2, pages II–104. IEEE, 2004.
- D. Liang, R. G. Krishnan, M. D. Hoffman, and T. Jebara. Variational Autoencoders for Collaborative Filtering. In *Proceedings of the 2018 World Wide Web Conference, WWW ’18*, pages 689–698, Lyon, France, Apr. 2018. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-5639-8. doi: 10.1145/3178876.3186150.
- F. Locatello, S. Bauer, M. Lucic, G. Raetsch, S. Gelly, B. Schölkopf, and O. Bachem. Challenging common assumptions in the unsupervised learning of disentangled representations. In *International Conference on Machine Learning*, pages 4114–4124, 2019.
- L. Maaløe, M. Fraccaro, V. Lievin, and O. Winther. Biva: A very deep hierarchy of latent variables for generative modeling. In *33rd Conference on Neural*

- Information Processing Systems*, page 8882. Neural Information Processing Systems Foundation, 2019.
- L. Matthey, I. Higgins, D. Hassabis, and A. Lerchner. dsprites: Disentanglement testing sprites dataset. <https://github.com/deepmind/dsprites-dataset/>, 2017.
- S. Narayanaswamy, T. B. Paige, J.-W. Van de Meent, A. Desmaison, N. Goodman, P. Kohli, F. Wood, and P. Torr. Learning disentangled representations with semi-supervised deep generative models. In *Advances in Neural Information Processing Systems*, pages 5925–5935, 2017.
- A. Radhakrishnan, K. Yang, M. Belkin, and C. Uhler. Memorization in overparameterized autoencoders. *arXiv preprint arXiv:1810.10333v3*, 2019.
- A. Razavi, A. van den Oord, and O. Vinyals. Generating Diverse High-Fidelity Images with VQ-VAE-2. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alch  -Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL <https://proceedings.neurips.cc/paper/2019/file/5f8e2fa1718d1bbcadf1cd9c7a54fb8c-Paper.pdf>.
- D. J. Rezende and F. Viola. Taming VAEs. *arXiv preprint arXiv:1810.00597*, 2018.
- D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *Proceedings of The 31st International Conference on Machine Learning*, pages 1278–1286, 2014.
- O. Shamir, S. Sabato, and N. Tishby. Learning and generalization with the information bottleneck. *Theoretical Computer Science*, 411(29):2696–2711, June 2010. ISSN 0304-3975. doi: 10.1016/j.tcs.2010.04.006.
- R. Shu, H. H. Bui, S. Zhao, M. J. Kochenderfer, and S. Ermon. Amortized inference regularization. In *Advances in Neural Information Processing Systems*, pages 4393–4402, 2018.
- N. Tishby, F. C. Pereira, and W. Bialek. The information bottleneck method. *arXiv:physics/0004057*, Apr. 2000.
- A. Vahdat and J. Kautz. NVAE: A Deep Hierarchical Variational Autoencoder. page 13.
- A. Van den Oord, N. Kalchbrenner, L. Espeholt, O. Vinyals, A. Graves, et al. Conditional image generation with pixelcnn decoders. In *Advances in neural information processing systems*, pages 4790–4798, 2016.
- T.-H. Wen, Y. Miao, P. Blunsom, and S. Young. Latent Intention Dialogue Models. In *International Conference on Machine Learning*, pages 3732–3741, July 2017.
- H. Xiao, K. Rasul, and R. Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.
- C. Zhang, S. Bengio, M. Hardt, and Y. Singer. Identity crisis: Memorization and generalization under extreme overparameterization. *arXiv preprint arXiv:1902.04698*, 2019.
- S. Zhao, H. Ren, A. Yuan, J. Song, N. Goodman, and S. Ermon. Bias and generalization in deep generative models: An empirical study. In *Advances in Neural Information Processing Systems*, pages 10815–10824, 2018.

# A Formal Proof of PAC Learnability for Decision Stumps

*Dedicated to Olivier Danvy on the occasion of his sixtieth birthday.*

Joseph Tassarotti  
tassarot@bc.edu  
Boston College  
USA

Anindya Banerjee  
anindya.banerjee@imdea.org  
IMDEA Software Institute  
Spain

Koundinya Vajjha  
kov5@pitt.edu  
University of Pittsburgh  
USA

Jean-Baptiste Tristan<sup>\*</sup>  
tristanj@bc.edu  
Boston College  
USA

## Abstract

We present a formal proof in Lean of probably approximately correct (PAC) learnability of the concept class of decision stumps. This classic result in machine learning theory derives a bound on error probabilities for a simple type of classifier. Though such a proof appears simple on paper, analytic and measure-theoretic subtleties arise when carrying it out fully formally. Our proof is structured so as to separate reasoning about deterministic properties of a learning function from proofs of measurability and analysis of probabilities.

**CCS Concepts:** • General and reference → Verification;  
• Theory of computation → Sample complexity and generalization bounds.

**Keywords:** interactive theorem proving, probably approximately correct, decision stumps

## ACM Reference Format:

Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. 2021. A Formal Proof of PAC Learnability for Decision Stumps. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3437992.3439917>

<sup>\*</sup>Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. CPP '21, January 18–19, 2021, Virtual, Denmark

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8299-1/21/01...\$15.00  
<https://doi.org/10.1145/3437992.3439917>

## 1 Introduction

Machine learning (ML) has achieved remarkable success in a number of problem domains. However, the often opaque nature of ML has led to concerns about its use in important contexts such as medical diagnosis and fraud detection. To address these concerns, researchers have developed a number of algorithms with proved guarantees of robustness, privacy, fairness, and accuracy.

One essential property for an ML algorithm is to *generalize* well to unseen data. That is, after an algorithm has been trained on some data, it should be possible to present it with new data and expect it to classify or analyze it correctly. Valiant introduced the framework of Probably Approximately Correct (PAC) learnability [39], which gives a mathematical characterization of what it means for an algorithm to generalize well. This framework has become an essential part of the study of computational and statistical learning theory, and a large body of theoretical results has been developed for proving that an algorithm generalizes.

However, at present, the vast majority of these and other proofs in ML theory are pencil-and-paper arguments about idealized versions of algorithms. There is considerable room for error when real systems are built based on these algorithms. Such errors can go unnoticed for long periods of time, and are often difficult to diagnose with testing, given the randomized behavior of ML systems. Moreover, the original pencil-and-paper proofs of correctness may have errors. Because machine learning algorithms often involve randomized sampling of continuous data, their formal analysis usually requires measure-theoretic reasoning, which is technically subtle.

Formal verification offers a way to eliminate bugs from analyses of algorithms and close the gap between theory and implementation. However, the mathematical subtleties that complicate rigorous pencil-and-paper reasoning about ML algorithms also pose a serious obstacle to verification. In particular, while there has been great progress in recent

years in formal proofs about randomized programs, this work has often been restricted to *discrete* probability theory. In contrast, machine learning algorithms make heavy use of both discrete and continuous data, a mixture that requires measure-theoretic probability.

Thus, to even begin formally verifying ML algorithms in a theorem prover, results from measure theory must be formalized first. In recent years, some libraries of measure-theoretic results have been developed in various theorem provers [1, 20, 27, 36, 38]. However, it can be challenging to tell which results needed for ML algorithms are missing from these libraries. The difficulty is that, on the one hand, standard textbooks on the theoretical foundations of machine learning [28, 29, 35] omit practically all measure-theoretic details. Meanwhile, research monographs that give completely rigorous accounts [16] present results in maximal generality, well beyond what appears to be needed for many ML applications. This generality comes at the cost of mathematical prerequisites that go beyond even a first or second course in measure theory.

This paper describes the formal verification in Lean [15] of a standard result of theoretical machine learning which illustrates the complexities of measure-theoretic probability. We consider a simple type of classifier called a *decision stump*. A decision stump classifies real-numbered values into two groups with a simple rule: all values  $\leq$  some threshold  $t$  are labeled 1, and those above  $t$  are labeled 0. We show that decision stumps are PAC learnable by proving a *generalization* bound, which bounds the number of training examples needed to obtain a chosen level of classification accuracy with high probability.

We describe more precisely below how decision stumps are trained and the bound that we have proved (Section 2). Although decision stumps appear simple, they are worth considering because they are the 1-dimensional version of a classifying algorithm for axis-aligned rectangles that is used as a motivating example in all of the standard textbooks in this field [28, 29, 35].

In spite of the seeming simplicity of this example, all three of the cited textbooks either give an incorrect proof of this result or omit what we found to be the most technically challenging part of the proof (Section 3). In noting this, we do not wish to exaggerate the importance of these errors. The proofs can be fixed, and in each book, the results are correctly re-proven later as consequences of general theorems. Rather our point is to emphasize that even basic results in this area can touch on subtle issues, and errors can evade notice despite much review and scrutiny by a wide audience. We believe this further motivates the need for machine-checked proofs of such results if they are to be deployed in high-importance settings.

A key component of our work is that we structure our formal proof in a manner that lets us separate the high-level reasoning found in textbook descriptions from the low-level

details about measurability. We outline the structure used to achieve this separation of concerns in Section 4. We then describe some preliminaries about measure-theoretic probability in Section 5. The Giry monad [18] allows us to give a precise description of the sources of randomness involved in training and evaluating the performance of classifiers (Section 6). We exploit this to split deterministic reasoning about basic properties of the stump learning algorithm (Section 7) from proofs of measurability (Section 8) and the analysis of bounds on probabilities (Section 9).

The proof is publicly available at <https://github.com/jtristan/stump-learnable>.

## 2 Decision Stumps and PAC Learnability

To motivate decision stumps and the result that we have formalized, consider the following scenario. Suppose a scientist has developed a test to measure levels of some protein in blood in order to diagnose a disease. Assume there is some (unknown) threshold  $t \in \mathbb{R}$  such that if the protein level is  $\leq t$ , then the patient has the disease, and otherwise does not. Given a random sample of patients whose disease status is known, the scientist wants to estimate the threshold  $t$  so that the test can be used to screen and diagnose future patients whose disease statuses are unknown.

In other words, the scientist wants to find a decision stump to classify whether patients have the disease or not. We can model the blood test as returning a nonnegative real number. There is some distribution  $\mu$  on the interval  $[0, \infty)$  representing levels of the protein in the population. The scientist has samples  $x_1, \dots, x_n \in [0, \infty)$  independently drawn from  $\mu$  giving the results of the test on a collection of  $n$  patients, along with labels  $y_1, \dots, y_n \in \{0, 1\}$  giving each patient's disease status. A label of 1 means a patient has the disease, while 0 means they do not, so that  $y_i = 1$  if and only if  $x_i \leq t$ . The  $(x_1, y_1), \dots, (x_n, y_n)$  are called *training examples*. The scientist is trying to pick some value  $\hat{t}$  as an estimate of  $t$  to use to classify future patients. In particular, she will use her estimate to define a decision stump classifier. To state this formally, we first define a function *LABEL* that assigns a label to a point  $x$  according to a threshold  $d$ :

$$\text{LABEL}(d, x) = \begin{cases} 1 & \text{if } x \leq d \\ 0 & \text{if } x > d \end{cases} \quad (1)$$

The scientist will pick some threshold  $\hat{t}$  and then use the classifier  $\lambda x. \text{LABEL}(\hat{t}, x)$  to label future patients. We call such a classifier a *hypothesis*.

How should the scientist select  $\hat{t}$ ? One idea is to take  $\hat{t}$  to be the maximum of the  $x_i$  that have label 1. (If no  $x_i$  has label 1, she can take  $\hat{t}$  to be 0.) This estimate, at least, would correctly label all of the training examples. This corresponds to the following *learning algorithm*  $\mathcal{A}$ , which returns a classifier

using this estimate:

$$\begin{aligned} \mathcal{A} &([ (x_1, y_1), \dots, (x_n, y_n) ]) \\ &= \text{let } \hat{t} = \max\{x_i \mid y_i = 1\} \text{ in} \\ &\quad \lambda x. \text{LABEL}(\hat{t}, x) \end{aligned} \tag{2}$$

where  $\max$  of the empty set is defined to be 0.<sup>1</sup> Of course, the estimate  $\hat{t}$  used in the classifier returned by this algorithm is not going to be exactly the value of  $t$ , especially if the number of training examples,  $n$ , is small. But if  $n$  is large enough, we might hope that a good estimate can be produced. The key question then becomes, how many training examples should the scientist use?

To answer this more precisely, we need to decide how to evaluate the quality of the classifier returned by  $\mathcal{A}$ . At first, one might think that the goal should be to minimize  $|\hat{t} - t|$ , that is, to get an estimate that is as close as possible to the true threshold  $t$ . While minimizing the distance between  $\hat{t}$  and  $t$  is useful, our primary concern should be how well we classify future examples. The **ERROR** of a classifier  $h$  is the probability that  $h$  mislabels a test example  $x$  randomly sampled from  $\mu$ :

$$\text{ERROR}(h) = \Pr_{x \sim \mu} (h(x) \neq \text{LABEL}(t, x)) \tag{3}$$

We write  $x \sim \mu$  in the above to indicate that the random variable  $x$  has distribution  $\mu$ . This  $x$  is independent of the training examples used by the scientist. While the definition of **ERROR** refers to this randomized scenario of drawing a test sample, for a fixed hypothesis  $h$  the quantity **ERROR**( $h$ ) is a real number.

Because the training examples the scientist uses are randomly selected, the **ERROR** of the hypothesis she selects using  $\mathcal{A}$  is a random variable. In practice, the scientist does not know either the distribution  $\mu$  or the target  $t$ , so she cannot compute the exact **ERROR** of the classifier she obtains. Nevertheless, she might want to try to ensure that the **ERROR** of the classifier she defines using  $\mathcal{A}$  will be below some small  $\epsilon$  with high probability.

To talk about the **ERROR** of the selected hypothesis precisely, let us first introduce a helper function which takes an unlabeled list of examples and returns a list where each example has been paired with its true label:

$$\begin{aligned} \text{LLIST} &([x_1, \dots, x_n]) \\ &= \text{MAP } (\lambda x. (x, \text{LABEL}(t, x))) [x_1, \dots, x_n] \end{aligned} \tag{4}$$

Then through her choice of  $n$ , the scientist can bound the following probability:

$$\Pr_{(x_1, \dots, x_n) \sim \mu^n} (\text{ERROR}(\mathcal{A}(\text{LLIST}([x_1, \dots, x_n]))) \leq \epsilon)$$

<sup>1</sup>We call this function an *algorithm* here, following Shalev-Shwartz and Ben-David [35], although the operations on real numbers involved are non-computable.

where  $(x_1, \dots, x_n) \sim \mu^n$  indicates that the variables  $x_i$  are drawn independently from the same distribution  $\mu$ . The following theorem, which is the central result that we have formalized, tells the scientist how to select  $n$  to achieve a desired bound on this probability:

**Theorem 2.1.** For all  $\epsilon$  and  $\delta$  in the open interval  $(0, 1)$ , if  $n \geq \frac{\ln(\delta)}{\ln(1-\epsilon)} - 1$  then

$$\Pr_{(x_1, \dots, x_n) \sim \mu^n} (\text{ERROR}(\mathcal{A}(\text{LLIST}([x_1, \dots, x_n]))) \leq \epsilon) \geq 1 - \delta \tag{5}$$

Before giving an informal sketch of how this theorem is proved, we briefly describe how this result fits into the framework of PAC learnability [39].

**PAC Learnability.** PAC learning theory gives an abstract way to explain scenarios like the one with the scientist described above. In this general set up, there is some set  $\mathcal{X}$  of possible examples and a set  $C$  of hypotheses  $h : \mathcal{X} \rightarrow \{0, 1\}$ , which are possible classifiers we might select. The set  $C$  is also called a *concept class*. In the case of decision stumps,  $\mathcal{X} = \mathbb{R}^{\geq 0}$ , and  $C = \{\lambda x. \text{LABEL}(d, x) \mid d \in \mathbb{R}^{\geq 0}\}$ .

As in the stump example, there is assumed to be some unknown distribution  $\mu$  over  $\mathcal{X}$ . Additionally, there is some function  $f : \mathcal{X} \rightarrow \{0, 1\}$  that maps examples to their true labels. The **ERROR** of a hypothesis is the probability that it incorrectly labels an example drawn according to  $\mu$ . The function  $f$  is said to be *realizable* if there is some hypothesis  $h \in C$  that has error 0. The goal is to select a hypothesis with minimal **ERROR** when given a collection of training examples that have been labeled according to  $f$ . A concept class is said to be PAC learnable if there is some algorithm to select a hypothesis for which we can compute the number of training examples needed to achieve error bounds with high probability as in the stump example:

**Definition 2.2.** A concept class  $C$  is PAC learnable if there exists an algorithm  $\mathcal{A} : \text{List}(\mathcal{X} \times \{0, 1\}) \rightarrow C$  and a function  $g : (0, 1)^2 \rightarrow \mathbb{N}$  such that for all distributions  $\mu$  on  $\mathcal{X}$  and realizable label functions  $f$ , when  $\mathcal{A}$  is run on a list of at least  $g(\epsilon, \delta)$  independently sampled examples from  $\mu$  with labels computed by  $f$ , the returned hypothesis  $h$  has error  $\leq \epsilon$  with probability  $\geq 1 - \delta$ .

By “there exists” in the above definition, one typically conveys a constructive sense: one can exhibit the algorithm and compute  $g$ .<sup>2</sup> The function  $g$  is a bound on the *sample complexity* of the algorithm, telling us how many training examples are needed to achieve a bound on the **ERROR** with a given probability. There is a large body of theoretical results for showing that a concept class is PAC learnable and for bounding the sample complexity by analyzing the VC-dimension [41] of the concept class.

<sup>2</sup>Some authors require further that the algorithm  $\mathcal{A}$  has polynomial running time, but we will not do so.

**Theorem 2.1** states that the concept class of decision stumps is PAC learnable. We next turn to how this theorem is proved.

### 3 Informal Proof of PAC Learnability

The PAC learnability of decision stumps follows from the general VC-dimension theory alluded to above. However, mechanizing that underlying theory in all its generality is beyond the scope of this paper.

Instead this paper formalizes a more elementary proof, based on a description from three textbooks [28, 29, 35]. The proofs in the textbooks are for a slightly different result—learning a 2-dimensional rectangle instead of a decision stump—but the idea is essentially the same. We first sketch how the proof is presented in two of the textbooks [28, 35]. As we shall see, this argument has a flaw.

*Proof sketch of Theorem 2.1.* Given  $\mu$  and  $\epsilon$ , consider labeled samples  $(x_1, y_1), \dots, (x_n, y_n)$ . Recall that the true threshold is some unknown value  $t$ , and the learning algorithm  $\mathcal{A}$  here returns a classifier that uses the maximum of the positively labeled examples as the decision threshold  $\hat{t}$ .

We start by noting some deterministic properties about this classifier. Observe that  $\mathcal{A}$  ensures  $\hat{t} \leq t$ , because all positively labeled training examples must be  $\leq t$ . Furthermore, a test example  $x$  is misclassified only if  $\hat{t} < x \leq t$ . Thus the only errors that the classifier selected by  $\mathcal{A}$  can make is by incorrectly assigning the label 0 to an example that should have label 1.

With this in mind, the proof goes by cases on the probability that a randomly sampled example  $x \sim \mu$  will have true label 1. First assume  $\Pr_{x \sim \mu}(x \leq t) \leq \epsilon$ . That is, this case assumes that examples with true label 1 are rare. Then the classifier returned by  $\mathcal{A}$  must have **ERROR** that is  $\leq \epsilon$ . In other words, if the returned classifier can only misclassify examples whose true label is 1, and those are sufficiently rare, i.e., have probability  $\leq \epsilon$  by the above assumption, then the classifier has the desired error bound.

Next assume  $\Pr_{x \sim \mu}(x \leq t) > \epsilon$ . The idea for this case is to find an interval  $\mathcal{I}$  such that, so long as at least one of the training examples  $x_i$  falls into  $\mathcal{I}$ , the classifier returned by  $\mathcal{A}$  will have error  $\leq \epsilon$ . Then we find a bound on the probability that *none* of the  $x_i$  fall into  $\mathcal{I}$ .

In particular, set  $\mathcal{I} = [\theta, t]$ , choosing  $\theta$  so that

$$\Pr_{x \sim \mu}(x \in \mathcal{I}) = \epsilon$$

That is, we want  $\mathcal{I}$  to enclose exactly probability  $\epsilon$  under  $\mu$ . Let  $E$  be the event that at least one of the training examples falls in  $\mathcal{I}$ . If  $E$  occurs, then the threshold  $\hat{t}$  selected by  $\mathcal{A}$  is in  $\mathcal{I}$ . To see this, observe that if for some  $x_i$  we have  $\theta \leq x_i \leq t$ , then we know  $y_i = 1$ , and hence  $\theta \leq x_i \leq \hat{t} \leq t$ .

In that case, for a test example  $x$  to be misclassified, we must have  $\hat{t} < x \leq t$ , meaning  $x$  must also lie in  $\mathcal{I}$ . Thus, the event of misclassifying  $x$  is a subset of the event that  $x$  lies in  $\mathcal{I}$ . Hence, if  $E$  occurs, the probability of misclassifying  $x$  is at

most the probability that  $x$  lies in  $\mathcal{I}$ . But the probability that  $x$  lies in  $\mathcal{I}$  is  $\epsilon$  by the way we defined  $\theta$ . Therefore if  $E$  occurs, the probability that a randomly selected example  $x$  will be misclassified is  $\leq \epsilon$ , meaning the **ERROR** of the classifier will be  $\leq \epsilon$ .

So for the error to be above  $\epsilon$  means that *none* of our training examples  $x_i$  came from  $\mathcal{I}$ . For each  $i$ , we have

$$\Pr_{(x_1, \dots, x_n)}(x_i \notin \mathcal{I}) = 1 - \epsilon$$

Because each  $x_i$  is sampled independently from  $\mu$ , the probability that none of the  $x_i$  lie in  $\mathcal{I}$  is  $(1 - \epsilon)^n$ . Thus the probability of  $E$ , the event that *at least* one  $x_i$  is in  $\mathcal{I}$ , is  $1 - (1 - \epsilon)^n$ . Since we have shown that if  $E$  occurs, then the **ERROR** is  $\leq \epsilon$ , this means that the probability that the **ERROR** is  $\leq \epsilon$  is *at least* the probability of  $E$ . The rest of the proof follows by choosing  $n$  to ensure  $1 - (1 - \epsilon)^n \geq 1 - \delta$ .  $\square$

The careful reader may notice that there is one subtle step in the above: how do we choose  $\theta$  to ensure that “ $\mathcal{I}$  encloses exactly probability  $\epsilon$  under  $\mu$ ”? The phrasing “encloses exactly” comes from Kearns and Vazirani [28] (page 4), which does not say how to prove that  $\theta$  exists, beyond giving some geometric intuition in which we visualize shifting the left edge of  $\mathcal{I}$  until the enclosed amount has the specified probability. Shalev-Shwartz and Ben-David [35] similarly instructs us to select  $\theta$  so that the probability “is exactly”  $\epsilon$ .<sup>3</sup>

Unfortunately, the argument is not correct, because such a  $\theta$  may not exist.<sup>4</sup> The following counterexample demonstrates this.

**Counterexample 3.1.** Take  $\mu$  to be the Bernoulli distribution which returns 1 with probability .5 and 0 otherwise. Let  $t = .5$ , and  $\epsilon = .25$ . Then for all  $a$  we have:

$$\Pr_{x \sim \mu}(x \in [a, t]) = \begin{cases} .5 & \text{if } a \leq 0 \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

so this means that no matter how we select  $\theta$ , we cannot have  $\Pr_{x \sim \mu}(x \in [\theta, t]) = \epsilon$ , so the desired  $\theta$  does not exist.

The issue in the proof is that the distribution function  $\mu$  has been assumed to be *continuous*, whereas in the above counterexample  $\mu$  is a *discrete* distribution. In particular, if there is some point  $y$  such that  $\Pr_{x \sim \mu}(x = y) > 0$  then this introduces a *jump discontinuity* in the distribution function.

However, the statement of PAC learnability says that the error bound should be achievable for *any* distribution  $\mu$ . In order to fix the proof to work for any  $\mu$ , we need to consider

<sup>3</sup>The cited references address the more general problem of axis-aligned rectangles instead of stumps, so more specifically they describe shifting the edge of a rectangle until the enclosed probability is  $\epsilon/4$ .

<sup>4</sup>We are not the first to observe this error. The errata for the first printing of Mohri et al. [29] points out the issue in the proof of Kearns and Vazirani [28].

the following revised definition of  $\theta$ , which will ensure it exists:

$$\theta = \sup \left\{ d \in \mathcal{X} \mid \Pr_{x \sim \mu}(x \in [d, t]) \geq \epsilon \right\} \quad (7)$$

In this definition, the set (say  $S$ ) over which we are taking the supremum might be infinite. However, recall that we only need to construct the point  $\theta$  in the sub-case of the proof where we assume that  $\Pr_{x \sim \mu}(x \leq t) > \epsilon$ . This assumption implies that the supremum exists, because it means that  $S$  is nonempty, and furthermore we know that  $S$  is bounded above by  $t$ . The existence of the supremum then follows from the fact that the real numbers are Dedekind complete.

The idea behind this definition of  $\theta$  is that, if the distribution function is continuous, then the definition picks a  $\theta$  that has the property required in the erroneous proof. Instead if there is a discontinuity that causes a jump in the distribution function past the value  $\epsilon$ , then the definition selects the point at that discontinuity. In particular, we can show that with this definition

$$\Pr_{x \sim \mu}(x \in [\theta, t]) \geq \epsilon \quad (8)$$

and also that

$$\Pr_{x \sim \mu}(x \in (\theta, t]) \leq \epsilon \quad (9)$$

Note in [Equation 8](#) we have the closed interval  $[\theta, t]$ , while [Equation 9](#) is about the half-open interval  $(\theta, t]$ . This means that if  $\Pr_{x \sim \mu}(x = \theta) = 0$ , as we would have in a continuous probability distribution, then  $\Pr_{x \sim \mu}(x \in [\theta, t]) = \epsilon$ . Whereas if  $\Pr_{x \sim \mu}(x = \theta) \neq 0$ , as can occur in a discrete distribution, the probabilities of lying in  $[\theta, t]$  and  $(\theta, t]$  will differ. For example, for the discrete distribution in [Counterexample 3.1](#), the definition of  $\theta$  in [Equation 7](#) would yield  $\theta = 0$ . Observe that  $\Pr_{x \sim \mu}(x = 0) = .5 \neq 0$  while  $\Pr_{x \sim \mu}(x = v) = 0$  for any  $v \in (0, 1)$ .

The original proof of PAC learnability of the class of rectangles [\[13\]](#) did give a correct definition of  $\theta$ , as does the textbook by Mohri et al. [\[29\]](#), although neither gives a proof for why the point defined this way has the desired properties. Indeed, [Mohri et al.](#) say that it is “not hard to see” that these properties hold.

In fact, this turned out to be the most difficult part of the whole proof to formalize. While it only requires some basic results in measure theory and topology, it is nevertheless the most technical step of the argument. There were two other parts of the proof that seemed obvious on paper but turned out to be much more technically challenging than expected, having to do with showing that various functions are measurable. Often, details about measurability are elided in pencil-and-paper proofs. This is understandable because these measurability concerns can be tedious and trivial, and checking that everything is measurable can clutter an otherwise insightful proof. However, many important results in

statistical learning theory do not hold without certain measurability assumptions, as discussed by Blumer et al. [\[13\]](#) and Dudley [\[16, chapter 5\]](#).

Now that we have seen some intuition for this result and some of the pitfalls in proving it, we describe the structure of our formal proof and how it addresses these challenges.

## 4 Structure of Formal Proof

When we examine the informal proof sketched above, we can see that there are several distinct aspects of reasoning. Instead of intermingling these reasoning steps as in the proof sketch, we structure our formal proof to separate these components. As we will see, this decomposition is enabled by features of Lean. We believe that this proof structure applies more generally to other proofs of PAC learnability and related results in ML theory.

Specifically, we identify the following four components. We describe each briefly in the paragraphs below. The remaining sections of the paper then elaborate on each of these parts of the formal proof, after giving some background on measure theory in [Section 5](#).

**Specifying sources of randomness:** As we saw in [Section 2](#), there are two randomized scenarios under consideration in the statement of [Theorem 2.1](#). First, there is the randomized choice of the training examples that are given as input to  $\mathcal{A}$ . These are sampled independently and from the same distribution  $\mu$ . This first source of randomness is explicit since it appears in the statement of the probability appearing in [Equation 5](#). The second kind of randomization is in the definition of **ERROR**. Recall that we defined **ERROR** in [Equation 3](#) as the probability that an example randomly sampled from  $\mu$  would be misclassified. This random sampling is entirely separate from the sampling of the training examples, although both samplings utilize the same  $\mu$ .

Finally, the theorem statement is quantifying over the distribution  $\mu$ . As we saw in the counterexample to the informal proof, inadvertently considering only certain classes of distributions (such as continuous ones) leads to erroneous arguments.

All of these details must be represented formally in the theorem prover. To handle these issues, we make use of the Giry monad [\[18\]](#) which allows us to represent sampling from distributions as monadic computations. [Section 6](#) explains how this provides a convenient way to model the training and testing of a learning algorithm in order to formally state [Theorem 2.1](#).

**Deterministic properties of the algorithm:** In the beginning of the proof sketch of [Theorem 2.1](#) we started by noting certain *deterministic* properties of the learning algorithm  $\mathcal{A}$ , such as the fact that the threshold value  $\hat{t}$  in the classifier returned by  $\mathcal{A}$  must be  $\leq$  the true unknown threshold  $t$ . These deterministic properties were the only details

about  $\mathcal{A}$  upon which we relied when later establishing the ERROR bound. This means that an analogue of [Theorem 2.1](#) will hold for any other stump learning algorithm with those properties.

As we will see, the Giry monad enables us to encode  $\mathcal{A}$  as a purely functional Lean term that selects the maximum of the positively labeled training examples. This means we can prove these preliminary deterministic properties in the usual way one reasons about pure functions in Lean. The Lean statements of these deterministic properties are described in [Section 7](#).

**Measurability of maps and events:** One detail missing from the informal proof was any consideration of *measurability* of functions and events. In measure-theoretic probability, probability spaces are equipped with a collection of *measurable sets*. We can only speak of the probability of an event if we show that the set corresponding to the event is *measurable*, meaning that it belongs to this collection. Similarly, random variables, such as the learning algorithm  $\mathcal{A}$  itself, must be *measurable functions*.

While these facts are necessary for a rigorous proof, they risk cluttering a formal proof and obscuring all of the intuition that the informal proof gave. However, with Lean’s typeclass mechanism and other proof automation, we can mostly separate the parts of the proof concerning measurability from the rest of the argument, as we describe in [Section 8](#).

**Quantitative reasoning about probabilities:** The last step of the proof involves constructing the point  $\theta$  described above and showing bounds on the probability that a sampled example lies in the interval  $[\theta, t]$ . Other than correcting the issue involved in the definition of  $\theta$ , this stage of reasoning is similar to the proof style found in informal accounts of this result. Our goal is that this portion of the proof should resemble the kind of probabilistic reasoning that is familiar to experts in ML theory. This portion of the argument, and the final proof of [Theorem 2.1](#) are described in [Section 9](#).

## 5 Preliminaries

In this section we describe some basic background on measure-theoretic probability and how measure theory has been formalized in Lean as part of the `mathlib` library [38].

**Measure theory.** The starting point for probability theory is a set  $\Omega$  called a *sample space*. Elements of  $\Omega$  are called outcomes, and represent possible results of some randomized situation. For example, if the randomized situation is the roll of a six-sided die, we would have  $\Omega = \{1, 2, 3, 4, 5, 6\}$ . In naive probability theory, subsets of  $\Omega$  are called events, and a probability function  $P$  on  $\Omega$  is a function mapping events to real numbers in the interval  $[0, 1]$ , satisfying some axioms. While this naive approach works so long as  $\Omega$  is a finite or countable set, attempting to assign probabilities to *all*

subsets of  $\Omega$  runs into technical difficulties when  $\Omega$  is an uncountable set such as  $\mathbb{R}$ .

Measure-theoretic probability theory resolves this issue by only assigning probabilities to a collection  $\mathcal{F}$  of subsets of  $\Omega$ . The elements of  $\mathcal{F}$  are called *measurable sets*. This collection  $\mathcal{F}$  must be a *sigma-algebra*, which means that it must be closed under certain operations (e.g. taking countable unions). We call the pair  $(\Omega, \mathcal{F})$  a *measurable space*. A probability measure  $\mu$  is then a function of type  $\mathcal{F} \rightarrow [0, 1]$  satisfying the following axioms:

- $\mu(\emptyset) = 0$
- $\mu(\Omega) = 1$
- If  $A_1, A_2, \dots$  is a countable collection of measurable sets such that  $A_i \cap A_j = \emptyset$  for  $i \neq j$ , then

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i)$$

For the reader familiar with topology, the notion of a measurable space is analogous to the situation in topology, where a topological space is a pair  $(X, \mathcal{V})$  where  $X$  is a set and  $\mathcal{V}$  is a collection of subsets of  $X$  called the open sets, and  $\mathcal{V}$  must be closed under various set operations. Indeed, for every topological space there is a minimal sigma-algebra containing all open sets, which is called the *Borel sigma-algebra*. We use the Borel sigma-algebra on the real numbers throughout the following.

A function  $f : (\Omega_1, \mathcal{F}_1) \rightarrow (\Omega_2, \mathcal{F}_2)$  between two measurable spaces is said to be a *measurable function* if, for all  $A \in \mathcal{F}_2$ , we have  $f^{-1}(A) \in \mathcal{F}_1$ . Again, there is an analogy to topology, where a function between topological spaces is continuous if inverse images of open sets are open. In fact, if  $f$  is a continuous function between two topological spaces, then  $f$  is measurable when those spaces are equipped with their Borel sigma-algebras. Continuity implies that many standard arithmetic operations on the reals are measurable. Other examples which are measurable but not continuous include functions for testing whether a real number is  $=, \leq$  or  $\geq$  to some value.

The general measure theory in `mathlib` allows the measures of events to be greater than 1. To obtain just probability measures, we restrict these general definitions to require that if  $\mu$  is a probability measure on  $X$ , then  $\mu(X) = 1$ . In [Section 2](#) we subscripted the `Pr` notation to indicate the distributions that we were considering. In the context of a theorem prover, which obliges us to be precise in this manner, we forgo the `Pr` notation altogether. Instead, for the probability of an event  $E$  with respect to some measure  $\mu$ , we simply write  $\mu(E)$ .

Above we have used the traditional mathematical notation of writing  $f(x)$  for the application of a function  $f$  to an argument  $x$ . However, to more closely match notation in Lean, the sequel uses  $f x$  when referring to definitions from our Lean development.

**Typeclasses.** In mathematical writing, we often associate a particular mathematical structure, such as a topology or sigma-algebra with a given set, with the convention that the structure should be used throughout. For example, when talking about continuous functions from  $\mathbb{R} \rightarrow \mathbb{R}$ , we do not constantly clarify that we mean continuous functions with respect to the topology generated by the Euclidean metric on  $\mathbb{R}$ .

This style of mathematical writing can be mimicked with Lean's typeclasses. After defining a typeclass, the user can declare instances of that typeclass, which associate a default structure with a given type. This mechanism is used throughout `mathlib` to supply default topologies, ring structures, and so on with particular types.

For example, the commands below first introduce the notation `H` to refer to the type `nnreal` of nonnegative real numbers from `mathlib`. We will use this notation when referring to the type of training examples and thresholds used for classification. After declaring this notation, an instance of `measurable_space` is defined on this type:

```
notation `H` := nnreal
instance meas_H: measurable_space H := ...
```

where we have omitted the definition after the `:=` sign. After this instance is declared, any time we refer to `H` in a context where we need a sigma-algebra, this instance will be used.

The `mathlib` library comes with lemmas to automatically derive instances of `measurable_space` from other instances. For example, if a type has been associated with a topology, we can automatically derive the Borel sigma-algebra as an instance of `measurable_space` for that type. We use this Borel sigma-algebra on `H` above. Similarly, we can derive a product sigma-algebra on the product `A × B` of two types from existing instances for `A` and `B`, as in the following example:

```
instance meas_lbl: measurable_space (H × bool)
```

**Measurable spaces for stump training.** At this point, considerations about the sigma-algebras with which a type is equipped introduce the first discrepancy between the informal set-up in [Section 2](#) and our formalization. As we described there, it is common to treat the learning algorithm as if it returned a *function* of type  $\mathbb{H} \rightarrow \{0, 1\}$ , mapping examples to labels. Since one wants to speak about probabilities involving these classifiers, this means the type of classifiers must be equipped with a sigma-algebra. What sigma-algebra should be chosen? While there are canonical choices for types that have a topology (the Borel sigma-algebra) and for various operations on spaces such as products, there is no such standard choice for function types. In particular, the category of measurable spaces is not Cartesian closed [4]. Hence, there is no generic sigma-algebra on function types that would also make evaluation measurable. The textbook by [Shalev-Shwartz and Ben-David](#) points out that the PAC

learnability framework requires the existence of a sigma-algebra on the class of hypotheses that makes classification measurable [35, Remark 3.1], but the *construction* of this sigma-algebra is not typically explained in examples.

Fortunately, the subset of decision stump classifiers has a simpler structure than the type of *all* functions from  $\mathbb{H} \rightarrow \{0, 1\}$ . In particular, the behavior of a decision stump classifier is entirely determined by the threshold used as a cut-off when assigning labels. These thresholds have type `H`, which is equipped with the Borel sigma-algebra. In particular, given a threshold `t`, the function  $\lambda x. \text{LABEL}(t, x)$  is measurable, and this is the evaluation function for a decision stump classifier. Thus, as we will see in the next section, we formalize the learning algorithm  $\mathcal{A}$  such that it directly returns a threshold instead of a classifier. Similarly we adjust the definitions of `ERROR` (and associated functions) to take a threshold as input instead of a classifier.

A similar concern arises with how we represent the collection of training examples passed to the learning algorithm  $\mathcal{A}$ . In the earlier informal presentation,  $\mathcal{A}$  takes a list of labeled examples as input. However, the construction of a sigma-algebra on variable-length lists is not commonly discussed in measure theory texts. We therefore work with dependently typed vectors of a specified length. Given a type `A` and natural `n`, the type `vec A n` represents vectors of size `n+1` of values of type `A`. When `A` is a topological space, `vec A n` can be given the  $(n + 1)$ -ary product topology, and we can then make it into a measurable space by equipping it with the Borel sigma-algebra.

## 6 Specifying Randomized Processes with the Giry Monad

Now that we have described some of the preliminaries of measure-theoretic probability, we turn to the question of how to formally represent the learning algorithm in the theorem prover.

In traditional probability theory, it is common to fix some sample space  $\Omega$  and then work with a collection of *random variables* on this sample space. If  $V$  is a measurable space, a  $V$ -valued random variable is a measurable function of type  $\Omega \rightarrow V$ . One can think of the elements of the sample space  $\Omega$  as some underlying source of randomness, and then the random variables encode how that randomness is transformed into an observable value. For example,  $\Omega$  could be a sequence of random coin flips, and a random variable  $f$  might be a randomized algorithm that uses those coin flips.

In fact, at a certain point most treatments of probability theory start to leave the sample space  $\Omega$  completely abstract. One simply postulates the existence of some  $\Omega$  on which a collection of random variables with various distributions are said to exist. To ensure that the resulting theory is not vacuous, a theorem is proven to show that there exists an  $\Omega$

and a measure  $\mu$  on  $\Omega$  for which a suitably rich collection of random variables can be constructed.

While this pencil-and-paper approach could be used in formalization, it is inconvenient in several ways. First, while random variables are formally functions on the sample space, in practice we often treat them as if they were elements of their codomain. For example if  $X$  and  $Y$  are two real-valued random variables, then one writes  $X + Y$  to mean the random variable  $\lambda\omega. X(\omega) + Y(\omega)$ . Similarly, if  $f : \mathbb{R} \rightarrow \mathbb{R}$ , we write  $f(X)$  for the random variable  $(\lambda\omega. f(X(\omega)))$ . While this kind of convention is well understood on paper, trying to overload notations in a theorem prover to support it seems difficult.

The Giry monad [18] solves this problem by providing a syntactic sugar to describe stochastic procedures concisely.

## 6.1 Definition of the Giry Monad

The Giry monad is a triple  $(\text{Meas}(\cdot), \text{bind}, \text{ret})$ . For any measurable space  $X$ ,  $\text{Meas}(X)$  is the space of probability measures over  $X$ , that is, functions from measurable subsets of  $X$  to  $[0, 1]$  that satisfy the additional axioms of probability measures. The function  $\text{bind}$  is of type  $\text{Meas}(X) \rightarrow (X \rightarrow \text{Meas}(Y)) \rightarrow \text{Meas}(Y)$ . That is, it takes a probability measure on  $X$ , a function that transforms values from  $X$  into probability measures over  $Y$ , and returns a probability measure on  $Y$ . The return function  $\text{ret}$  is of type  $X \rightarrow \text{Meas}(X)$ . It takes a value from  $X$  and returns a probability measure on  $X$ . The `mathlib` library defines this monad for general measures, which we then restrict to probability measures.

Functions  $\text{bind}$  and  $\text{ret}$  construct probability measures, so their definitions say what probability they assign to an event. Letting  $A$  be an event we define:

$$\text{bind } \mu f A = \int_{x \in X} f(x)(A) d\mu \quad (10)$$

$$\text{ret } x A = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

While we give the definitions here using standard mathematical notation, the formalization in `mathlib` uses the Lean definition of the integral. Here,  $\text{ret}(x)$  is the distribution that always returns  $x$  with probability 1. The definition of  $\text{bind } \mu f$  corresponds to first sampling from  $\mu$  to obtain some value  $x$ , and then continuing with the probability measure  $f(x)$ . Then,  $\text{bind}$  and  $\text{ret}$  satisfy the usual monad laws

$$\text{bind } (\text{ret } x) f = \text{ret } (f x) \quad (12)$$

$$\text{bind } \mu (\lambda x. \text{ret } x) = \mu \quad (13)$$

$$\text{bind } (\text{bind } \mu f) g = \text{bind } \mu (\lambda x. \text{bind } (f x) g) \quad (14)$$

However, these laws only hold when  $f$  and  $g$  are measurable functions. We will use the usual `do`-notation, writing `do x ← μ ; g(x)` for  $\text{bind } \mu g$ .

As with any monad, we can define a function  $\text{map}$  which lifts a function  $f : A \rightarrow B$ , to a function of type  $\text{Meas}(A) \rightarrow \text{Meas}(B)$ . Given  $\mu : \text{Meas}(A)$ , we interpret  $\text{map } f \mu$  as the

probability distribution that first samples from  $\mu$  to obtain a value of type  $A$ , and then applies  $f$  to it. Concretely, this is defined in terms of `bind` and `ret` as:

$$\text{map } f \mu = \text{do } x \leftarrow \mu ; \text{ret } (f x) \quad (15)$$

As an example, we show how to construct a distribution that samples  $n$  independent times from a distribution  $\mu$  and returns the result as a tuple. That is, we formally define the  $\mu^n$  distribution that we used in Section 2. Let  $(X, \mathcal{F})$  be a measurable space and  $(X^n, \mathcal{F}^n)$  be the measurable space where  $X^n$  is the cartesian product of  $X$  ( $n$  times) and  $\mathcal{F}^n$  is the product of  $\mathcal{F}$  ( $n$  times). Let  $\mu$  be a probability measure on  $(X, \mathcal{F})$ . Then, we define the measure  $\mu^n$  recursively on  $n$  as

$$\mu^1 = \mu \quad (16)$$

$$\mu^n = \text{do } v \leftarrow \mu^{n-1} ; \text{do } s \leftarrow \mu ; \text{ret } (s, v) \quad (17)$$

Instead of nested tuples, as in the above, or lists of training examples, as we did in Section 2, we will use dependently typed vectors in Lean. The following term (definition omitted) gives the probability measure corresponding to taking  $n+1$  independent samples from a distribution and assembling them in a vector:

```
def vec.prob_measure
  (n : ℕ) (μ : probability_measure A)
  : probability_measure (vec A n) := ...
```

## 6.2 Modeling Stump Training and Testing

Let us now see how the decision stump training and testing procedures can be described using the Giry monad. Lean has a sectioning mechanism and a way to declare local variables. In our formalization, the lines below declare probability measures over the class of examples, and an arbitrary target threshold value for labeling samples:

```
variables (μ: probability_measure ℙ) (target: ℙ)
```

When we write definitions that use these variables, Lean will interpret the definition to treat these variables as if they were additional parameters on which the function depends.<sup>5</sup>

The following pure function labels a sample according to the target.

```
def label (target: ℙ): ℙ → ℙ × bool :=
  λ x: ℙ, (x, rle x target)
```

where `rle x target` returns true if  $x \leq \text{target}$  and false otherwise.

Finally, we can define the event that an example is misclassified, and the `ERROR` function:

```
def error_set (h: ℙ) :=
  {x: ℙ | label h x ≠ label target x}

def error := λ h, μ (error_set h target)
```

<sup>5</sup>This is different from the behavior in Coq, where a definition that depends on section variables is only generalized to take additional arguments *outside* the section where the variables are declared.

The learning function  $\mathcal{A}$  will take a vector of labeled examples as input and output a hypothesis. The function `vec_map` takes a function as an argument and applies it pointwise to the elements of the vector. We use this to label the inputs to the learning function:

```
def label_sample := vec_map (label target)
```

Our learning algorithm starts by transforming any negative example to 0 and then stripping off the labels, with the following function:

```
def filter :=
  vec_map (λ p, if p.snd then p.fst else 0)
```

This is safe since, if there were no positive examples, the learning algorithm should return 0 as we described in [Section 2](#). Finally, we can define the learning function  $\mathcal{A}$  that we will use. We call this function `choose` in the formalization. It selects the largest example after the above filtering process:

```
def choose (n: ℕ):vec (ℍ × bool) n → ℍ :=
```

```
λ data: (vec (ℍ × bool) n), max n (filter n data)
```

We then use the Giry monad to describe the measure on classifiers that results from running the algorithm:

```
def denot: probability_measure ℍ :=
  let η := vec.prob_measure n μ in
  let ν := map (label_sample target n) η in
  let γ := map (choose n) ν in
  γ
```

Note that `map` here is the monadic map operation defined in [Equation 15](#), not `vec_map`. Thus,  $\eta$  is a distribution on training examples which is then transformed to  $\nu$ , a distribution on labeled training examples. Then  $\nu$  is transformed into a distribution  $\gamma$  on thresholds by lifting `choose`.

Finally, we have the following formal version of [Theorem 2.1](#):

`theorem choose_PAC:`

```
∀ ε: nnreal, ∀ δ: nnreal, ∀ n: ℕ,
ε > 0 → ε < 1 → δ > 0 → δ < 1 →
n > (complexity ε δ) →
(denot μ target n) {h: ℍ | error μ target h ≤ ε}
  ≥ 1 - δ
```

where `complexity` is the following function:

```
def complexity (ε: ℝ) (δ: ℝ) : ℝ :=
(log(δ) / log(1 - ε)) - (1: nat)
```

The following sections outline how we prove this theorem in Lean.

## 7 Deterministic Reasoning

The overall proof builds on two simple assumptions that must be satisfied by the learning algorithm. First, the algorithm must return an estimate that is  $\leq \text{target}$ . Formally,

```
lemma choose_property_1:
  ∀ n: ℕ, ∀ S: vec ℍ n,
  choose n (label_sample target n S) ≤ target
```

Our implementation satisfies this property since after the `filter` step in `choose`, all of the examples will have been mapped to a value  $\leq \text{target}$ , and we then select the maximum value from the vector.

Second, the algorithm must return an estimate that is greater or equal to any positive example. This is because the proof uses the assumption that no examples lie in the region between the estimate and the target. Formally,

`lemma choose_property_2:`

```
  ∀ n: ℕ, ∀ S: vec ℍ n,
  ∀ i,
  ∀ p = kth_projn (label_sample target n S) i,
  p.snd = tt →
  p.fst ≤ choose n (label_sample target n S)
```

where `kth_projn l i` is an expression giving component  $i$  of the vector  $l$ . Our implementation satisfies this property because `choose` calls `filter` which leaves positive examples unchanged, and then we select the maximum from the filtered vector.

These proofs are trivial and account for only a very small fraction of the overall proof. Yet, these are the two specific properties of the algorithm that we need.

## 8 Measurability Considerations

About a quarter of the formalization consists in proving that various sets and functions are measurable. The predicate `is_measurble S` states that the set  $S$  is a measurable set, while `measurable f` states that the function  $f$  is measurable. These proofs can be long but are generally routine, with a few notable exceptions.

We will need to divide up the sample space into various intervals. If  $a$  and  $b$  are two reals, then we write `Ioo a b`, `Ioc a b`, `Ico a b`, `Icc a b` in Lean to refer to the intervals  $(a, b)$ ,  $[a, b]$ ,  $[a, b)$ , and  $[a, b]$ , respectively. To start, we observe that the error of a hypothesis is the measure of the interval between it and the target.

`lemma error_interval_1:`

```
  ∀ h, h ≤ target →
  error μ target h = μ (Ioc h target)
```

`lemma error_interval_2:`

```
  ∀ h, h > target →
  error μ target h = μ (Ioc target h)
```

We next use these lemmas to prove that the function that computes the `ERROR` of a hypothesis is measurable:

`lemma error_measurable:`

```
  measurable (error μ target)
```

*Proof.* First, note that if  $A$  and  $B$  are measurable subsets such that  $A \subseteq B$ , then  $\mu(B \setminus A) = \mu B - \mu A$ . If  $h \leq \text{target}$ , then

$$\begin{aligned} \text{error } μ \text{ target } h &= μ(h, \text{target}) \\ &= μ[0, \text{target}] - μ[0, h] \end{aligned}$$

Likewise, if  $h > \text{target}$  then  $\text{error } \mu \text{ target } h = \mu [0, h] - \mu [0, \text{target}]$ . Subtraction is measurable and testing whether a value is  $\leq \text{target}$  is measurable. Therefore, since measurability is closed under composition, it suffices to show that the function  $\lambda x. \mu [0, x]$  is Borel measurable. Because this function is monotone, its measurability is a standard result, though this fact was missing from `mathlib`.  $\square$

Next, one must show that the learning algorithm `choose` is measurable, after fixing the number of input examples:

```
lemma choose_measurable: measurable (choose n)
```

*Proof.* To prove that `choose` is a measurable function, we must prove that `max` is a measurable function. Because `max` is continuous, it is Borel measurable.  $\square$

Although the previous proof is straightforward, it hinges on the fact that the sigma-algebra structure we associate with `vec H n` is the Borel sigma-algebra. But, because we define a vector as an iterated product, another possible sigma-algebra structure for `vec H n` is the  $n+1$ -ary product sigma-algebra.

Recall from the previous section that our development uses Lean's typeclass mechanism to automatically associate product sigma-algebras with product spaces, and Borel sigma-algebras with topological spaces. As the preceding paragraph explains, for `vec H n` there are two possible choices. Which choice should be used? In programming languages with typeclasses, the problem of having to select between two potentially different instances of a typeclass is called a *coherence* problem [31]. Because of this ambiguity, `mathlib` is careful to only enable certain instances by default. Of course, this same potential ambiguity arises in normal mathematical writing, when we omit mentioning the associated sigma-algebra.

Fortunately, in the case of `vec H n`, these two sigma-algebras happen to be the same. In general, if  $X$  and  $Y$  are topological spaces with a *countable basis*, then the Borel sigma-algebra on  $X \times Y$  is equal to the product of the Borel sigma-algebras on  $X$  and  $Y$ . The standard topology on the nonnegative reals has a countable basis, so the equivalence holds. Thus, although the proof of measurability for `max` can be simple, it uses a subtle fact that resolves the ambiguity involved in referring to sets without constantly mentioning their sigma-algebras.

## 9 Probabilistic Reasoning

The remainder of the proof involves the construction of the point  $\theta$  and explicitly bounding the probability of various events.

Recall from the informal sketch in Section 3 that we first case split on whether  $\Pr_{x \sim \mu}(x \leq \text{target}) \leq \epsilon$  or not. In the language of measures, this is equivalent to a case split on whether  $\mu [0, \text{target}] \leq \epsilon$ . In the formalization, it simplifies slightly the application of certain lemmas if we instead split on whether  $\mu (0, \text{target}] \leq \epsilon$ . The following lemma is the key property in the case where the weight between 0 and

target is  $\leq \epsilon$ . In that case, the learning algorithm always chooses a hypothesis with error at most  $\epsilon$ .

lemma always\_succeed:

$$\begin{aligned} \forall \epsilon: \text{nnreal}, \epsilon > 0 \rightarrow \forall n: \mathbb{N}, \\ \mu (\text{Ioc } 0 \text{ target}) \leq \epsilon \rightarrow \\ \forall S: \text{vec } \mathbb{H} n, \\ \text{error } \mu \text{ target} \\ (\text{choose } n (\text{label\_sample target } n S)) \\ \leq \epsilon \end{aligned}$$

*Proof.* By `error_interval_1`, we know that the error is going to be equal to the measure of the interval

$$(\text{Ioc } (\text{choose } n (\text{label\_sample target } n S)) \text{ target})$$

Because we know `choose` must return a threshold between 0 and `target`, this interval is a subset of  $(\text{Ioc } 0 \text{ target})$ . Since measures are monotone, this means the measure of that interval must be  $\leq \mu (\text{Ioc } 0 \text{ target})$ , which is  $\leq \epsilon$  by assumption.  $\square$

For the case where  $\mu (0, \text{target}] > \epsilon$ , the informal sketch selected a point  $\theta$  such that  $\mu [\theta, \text{target}] = \epsilon$ . However, as we saw in Counterexample 3.1, such a  $\theta$  may not exist when  $\mu$  is not continuous. Instead, we construct  $\theta$  so that  $\mu [\theta, \text{target}] \geq \epsilon$ , and  $\mu (\theta, \text{target}] \leq \epsilon$ . The following theorem states the existence of such a point:

theorem extend\_to\_epsilon\_1:

$$\begin{aligned} \forall \epsilon: \text{nnreal}, \epsilon > 0 \rightarrow \\ \mu (\text{Ioc } 0 \text{ target}) > \epsilon \rightarrow \\ \exists \theta: \text{nnreal}, \mu (\text{Icc } \theta \text{ target}) \geq \epsilon \wedge \\ \mu (\text{Ioc } \theta \text{ target}) \leq \epsilon \end{aligned}$$

*Proof.* We take  $\theta$  to be  $\sup\{x \in X \mid \mu [x, \text{target}] \geq \epsilon\}$ . The supremum exists because the set in question is bounded above by `target`, and the set is nonempty because it must contain 0 by our assumption that  $\mu (0, \text{target}] > \epsilon$ . To see that  $\mu [\theta, \text{target}] \geq \epsilon$ , we can construct an increasing sequence of points  $x_n \leq \theta$  such that  $\lim_{n \rightarrow \infty} x_n = \theta$ , where for each  $n$ ,  $\mu [x_n, \text{target}] \geq \epsilon$ . We have then that:

$$\bigcap_i [x_i, \text{target}] = [\theta, \text{target}]$$

We use this in conjunction with the fact that measures are continuous from above, meaning that if  $A_1, A_2, \dots$  is a sequence of measurable sets such that  $A_{i+1} \subseteq A_i$  for all  $i$ , then

$$\mu \left( \bigcap_{i=1}^{\infty} A_i \right) = \lim_{i \rightarrow \infty} \mu A_i$$

Hence we have

$$\begin{aligned} \mu [\theta, \text{target}] &= \mu \left( \bigcap_{i=1}^{\infty} [x_i, \text{target}] \right) \\ &= \lim_{n \rightarrow \infty} \mu [x_n, \text{target}] \\ &\geq \epsilon \end{aligned}$$

The proof that  $\mu(\theta, \text{target}] \leq \epsilon$  is the dual argument, using continuity from below.  $\square$

The conclusion of this theorem states two inequalities involving  $\theta$ . On the one hand, we need  $\theta$  to be small enough that we can ensure at least one *training example* will lie between  $\theta$  and target. On the other hand, we want  $\theta$  to be large enough that if we *only* misclassify *test examples* that lie between  $\theta$  and target, the error will nevertheless be at most  $\epsilon$ .

The next two lemmas formalize these properties. Recall that choose maps all negative training examples to 0, leaves positive examples unchanged, and then takes the maximum of the resulting vector. The next lemma says that given a point  $\theta$  such that  $\mu[\theta, \text{target}] \geq \epsilon$ , the measure of the event that an example gets mapped to something less than  $\theta$  is at most  $1 - \epsilon$ .

```
lemma miss_prob:
  ∀ ε, ∀ θ: nnreal, θ > 0 →
  μ(Icc θ target) ≥ ε →
  μ{x : ℙ | ∀ a b,
    (a,b) = label target x →
    (if b then a else 0) < θ} ≤ 1 - ε
```

The next lemma shows why the property  $\mu(\theta, \text{target}] \leq \epsilon$  is useful. In particular, it says that for such a  $\theta$ , in order to have an error  $> \epsilon$  on the hypothesis selected by choose, all training examples must get mapped to something less than  $\theta$ . Formally, we say that the set of training samples which would lead to an error greater than  $\epsilon$ , is a subset of those in which all the examples get mapped to a value less than  $\theta$ .

```
lemma all_missed:
  ∀ ε: nnreal,
  ∀ θ: nnreal,
  μ(Ioc θ target) ≤ ε →
  {S | error μ target
    (choose n (label_sample target n S))
    > ε} ⊆
  {S | ∀ i,
    ∀ p = label target (kth_projn S i),
    (if p.snd then p.fst else 0) < θ}
```

Finally, we prove a bound related to the complexity function, which computes the number of training examples needed:

```
lemma complexity_enough:
  ∀ ε: nnreal, ∀ δ: nnreal, ∀ n: ℙ,
  ε > (0: nnreal) → ε < (1: nnreal) →
  δ > (0: nnreal) → δ < (1: nnreal) →
  (n: ℙ) > (complexity ε δ) → ((1 - ε)^(n+1)) ≤ δ
```

Combining these lemmas together, we can finish the proof:

*Proof of choose\_PAC.* We have seen that always\_succeed handles the case  $\mu(0, \text{target}] \leq \epsilon$ . For the other case, where  $\mu(0, \text{target}] > \epsilon$ , we can apply extend\_to\_epsilon\_1 to get a  $\theta$  with the specified properties. By all\_missed we

know that the event that the hypothesis selected has error  $> \epsilon$  is a subset of the event where all the training examples get mapped to  $< \theta$ . Then, by miss\_prob we know the probability that a given example gets mapped to  $< \theta$  is  $\leq 1 - \epsilon$ . Because the training examples are selected independently, the probability that all  $n + 1$  examples get mapped to a value  $< \theta$  is at most  $(1 - \epsilon)^{n+1}$ . Applying complexity\_enough, we have that  $(1 - \epsilon)^{n+1} \leq \delta$ , so we are done.  $\square$

## 10 Related Work

Classic results about the average case behavior of quicksort and binary search trees have been formalized by a number of authors using different proof assistants [17, 37, 40]. In each case, the authors write down the algorithm to be analyzed using a variant of the monadic style we discuss in Section 6. Gopinathan and Sergey [19] verify the error rate of Bloom Filters and variants. Affeldt et al. [2] formalize results from information theory about lossy encoding. For the most part, these formalizations only use discrete probability theory, with the exception of Eberl et al.'s analysis of treaps [17], which requires general measure-theoretic probability. They report that dealing with measurability issues adds some overhead compared to pencil-and-paper reasoning, though they are able to automate many of these proofs.

Several projects have formalized results from cryptography, which also involves probabilistic reasoning [8, 9, 12, 30]. A challenge in formalizing such proofs lies in the need to establish a relation between the behavior of two different randomized algorithms, as part of the game-playing approach to cryptographic security proofs. Because cryptographic proofs generally only use discrete probability theory, these libraries do not formalize measure-theoretic results. There are many connections between cryptography and learning theory [32], which would be interesting to formalize.

There have been formalizations of measure-theoretic probability theory in a few proof assistants. Hurd [23] formalized basic measure theory in the HOL proof assistant, including a proof of Caratheodory's extension theorem. Hözl and Heller [21] developed a more substantial library in the Isabelle theorem prover, which has since been extended further. Avigad et al. [5] used this library to formalize a proof of the Central Limit Theorem. Several measure theory libraries have also been developed in Coq [1, 27, 36]. The ALEA library [3] instead uses a synthetic approach to discrete probability in Coq, a technique that has subsequently been extended to continuous probabilities by Bidlingmaier et al. [11].

More recent work has formalized theoretical machine learning results. Selsam et al. [34] use Lean to prove the correctness of an optimization procedure for stochastic computation graphs. They prove that the random gradients used in their stochastic backpropagation implementation are unbiased. In their proof, they add axioms to the system for various basic mathematical facts. They argue that even if

there are errors in these axioms that could potentially lead to inconsistency, the process of constructing formal proofs for the rest of the algorithm still helps eliminate mistakes.

Bagnall and Stewart [6] use Coq to give machine-checked proofs of bounds on generalization errors. They use Hoeffding's inequality to obtain bounds on error when the hypothesis space is finite or there is a separate test-set on which to evaluate a classifier after training. They apply this result to bound the generalization error of ReLU neural networks with quantized weights. Their proof is restricted to discrete distributions and adds some results from probability theory as axioms (Pinsker's inequality and Gibbs' inequality).

Bentkamp et al. [10] use Isabelle/HOL to formalize a result by Cohen et al. [14], which shows that deep convolutional arithmetic circuits are more expressive than shallow ones, in the sense that shallow networks must be exponentially larger in order to express the same function. Although convolutional arithmetic circuits are not widely used in practice compared to other artificial neural networks, this result is part of an effort to understand theoretically the success of deep learning. Bentkamp et al. report that they proved a stronger version of the original result, and doing so allowed them to structure the formal proof in a more modular way. The formalization was completed only 14 months after the original arXiv posting by Cohen et al., suggesting that once the right libraries are available for a theorem prover, it is feasible to mechanize state of the art results in some areas of theoretical machine learning in a relatively brief period of time.

After the development described by our paper was publicly released, Zinkevich [42] published a Lean library for probability theory and theoretical machine learning. Among other results, this library contains theorems about PAC learnability when the class of hypotheses is finite. Because the decision stump hypothesis class is the set of all nonnegative real numbers, our result is not covered by these theorems.

A related but distinct line of work applies machine learning techniques to automatically construct formal proofs of theorems. Traditional approaches to automated theorem proving rely on a mixture of heuristics and specialized algorithms for decidable sub-problems. By using a pre-existing corpus of formal proofs, supervised learning algorithms can be trained to select hypotheses and construct proofs in a formal system [7, 22, 24–26, 33].

## 11 Conclusion

We have presented a machine-checked, formal proof of PAC learnability of the concept class of decision stumps. The proof is formalized using the Lean theorem prover. We used the Giry monad to keep the formalization simple and close to a pencil-and-paper proof. To formalize this proof, we specialized the measure theory formalization of the `mathlib` library to the necessary basic probability theory. As expected,

the formalization is at times subtle when we must consider topological or measurability results, mostly to prove that the learning algorithm and `ERROR` are measurable functions. The most technical part of the proof has to do with proving the existence of an interval with the appropriate measure, a detail that standard textbook proofs either omit or get wrong.

Our work shows that the Lean prover and the `mathlib` library are mature enough to tackle a simple but classic result in statistical learning theory. A next step would be to formally prove more general results from VC-dimension theory. In addition, there exist a number of generalizations of PAC learnability, such as *agnostic* PAC learnability, which removes the assumption that some hypothesis in the class perfectly classifies the examples. Other generalizations allow for more than two classification labels and different kinds of `ERROR` functions. It would be interesting to formalize these various extensions and some related applications.

## Acknowledgments

We thank Gordon Stewart for comments on a previous draft of the paper. We thank the anonymous reviewers from the CPP'21 PC for their feedback. Some of the work described in this paper was performed while Koundinya Vajjha was an intern at Oracle Labs. Vajjha was additionally supported by the Alfred P. Sloan Foundation under grant number G-2018-10067. Banerjee's research was based on work supported by the US National Science Foundation (NSF), while working at the Foundation. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the NSF.

## References

- [1] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Kazuhiko Sakaguchi, and Pierre-Yves Strub. 2020. `math-comp` Analysis Library. <https://github.com/math-comp/analysis>.
- [2] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Autom. Reason.* 53, 1 (2014), 63–103.
- [3] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* 74, 8 (2009), 568–589.
- [4] Robert J. Aumann. 1961. Borel structures for function spaces. *Illinois J. Math.* 5, 4 (12 1961), 614–630.
- [5] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. 2017. A Formally Verified Proof of the Central Limit Theorem. *J. Autom. Reason.* 59, 4 (2017), 389–423.
- [6] Alexander Bagnall and Gordon Stewart. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *AAAI'19: The Thirty-Third AAAI Conference on Artificial Intelligence*. 2662–2669.
- [7] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Thirty-sixth International Conference on Machine Learning (ICML)*. 454–463.
- [8] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 146–166.

- [9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. 90–101.
- [10] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. 2019. A formal proof of the expressiveness of deep learning. *Journal of Automated Reasoning* 63, 2 (2019), 347–368.
- [11] Martin E. Bidlingmaier, Florian Faissole, and Bas Spitters. 2019. Synthetic topology in Homotopy Type Theory for probabilistic programming. *CoRR* abs/1912.07339 (2019). arXiv:1912.07339 <http://arxiv.org/abs/1912.07339>
- [12] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy*. 140–154.
- [13] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)* 36, 4 (1989), 929–965.
- [14] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the Expressive Power of Deep Learning: A Tensor Analysis. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016*. 698–728.
- [15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE-25 - 25th International Conference on Automated Deduction*. 378–388.
- [16] R. M. Dudley. 2014. *Uniform Central Limit Theorems* (2nd ed.). Cambridge University Press.
- [17] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2018. Verified Analysis of Random Trees. In *ITP*. 196–214.
- [18] Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis (Lecture Notes in Mathematics, Vol. 915)*, B. Banaschewski (Ed.), 68–85.
- [19] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In *CAV*, Shuvendu K. Lahiri and Chao Wang (Eds.). 279–303.
- [20] Johannes Hözl. 2013. *Construction and stochastic applications of measure spaces in higher-order logic*. Ph.D. Dissertation. Technical University Munich.
- [21] Johannes Hözl and Armin Heller. 2011. Three Chapters of Measure Theory in Isabelle/HOL. In *ITP*. 135–151.
- [22] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *7th International Conference on Learning Representations, ICLR 2019*.
- [23] Joe Hurd. 2003. *Formal Verification of Probabilistic Algorithms*. Ph.D. Dissertation. Cambridge University.
- [24] Jan Jakubuv and Josef Urban. 2019. Hammering Mizar by Learning Clause Guidance (Short Paper). In *ITP*. 34:1–34:8.
- [25] Cezary Kaliszyk, François Chollet, and Christian Szegedy. 2017. Hol-Step: A Machine Learning Dataset for Higher-order Logic Theorem Proving. In *5th International Conference on Learning Representations, ICLR 2017*.
- [26] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšík. 2018. Reinforcement Learning of Theorem Proving. In *NeurIPS*. 8836–8847.
- [27] Robert Kam. 2008. coq-markov Library. <https://github.com/coq-contribs/markov>.
- [28] Michael J Kearns and Umesh Virkumar Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT press.
- [29] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of Machine Learning*. MIT press.
- [30] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *POST*. 53–72.
- [31] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*.
- [32] Ronald L. Rivest. 1991. Cryptography and Machine Learning. In *Advances in Cryptology - ASIACRYPT '91*. 427–439.
- [33] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019*. 336–353.
- [34] Daniel Selsam, Percy Liang, and David Dill. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In *International Conference on Machine Learning (ICML)*. 3047–3056.
- [35] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- [36] Joseph Tassarotti. 2020. coq-proba Probability Library. <https://github.com/jtassarotti/coq-proba>.
- [37] Joseph Tassarotti and Robert Harper. 2018. Verified Tail Bounds for Randomized Programs. In *ITP*. 560–578.
- [38] The mathlib Community. 2020. The Lean Mathematical Library. In *CPP*. 367–381.
- [39] Leslie G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.
- [40] Eelis van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *TYPES*. 256–271.
- [41] Vladimir Naumovich Vapnik. 2000. *The Nature of Statistical Learning Theory, Second Edition*. Springer.
- [42] Martin Zinkevich. 2020. <https://github.com/google/formal-ml>

# mad-GP: Automatic Differentiation of Gaussian Processes for Molecules and Materials

Daniel Huang,<sup>\*,†</sup> Chong Teng,<sup>‡</sup> Junwei Lucas Bao,<sup>\*,‡</sup> and Jean-Baptiste Tristan<sup>\*,¶</sup>

<sup>†</sup>*Department of Computer Science, San Francisco State University, San Francisco, California 94132, United States*

<sup>‡</sup>*Department of Chemistry, Boston College, Chestnut Hill, Massachusetts 02467, United States*

<sup>¶</sup>*Department of Computer Science, Boston College, Chestnut Hill, Massachusetts 02467, United States*

E-mail: danehuang@sfsu.edu; lucas.bao@bc.edu; tristanj@bc.edu

## Abstract

In this paper, we introduce a Python library called mad-GP that enables users to more easily explore the design space of *Gaussian process* (GP) surrogate models for modeling potential energy surfaces (PESs). A user of mad-GP only needs to write down the functional form of the prior mean function (*i.e.*, a prior guess for the PES) and kernel function (*i.e.*, a constraint on the class of PESs), and the library handles all required derivative implementations via *automatic differentiation* (AD). We validate the design of mad-GP by applying it to perform geometry optimization of small molecules. In particular, we test the effectiveness of fitting GP surrogates to energies and/or forces, and perform a preliminary study on the use of non-constant priors and hierarchical kernels in GP PES surrogates. We find that GPs that fit forces perform comparably with GPs that fit both energies and forces, although force-only GPs are more robust for optimization because they do not require an additional step to be applied during optimization. We also confirm that constant mean functions and Matérn kernels work well as reported in the literature, although our tests

also identify several other promising candidates (*e.g.*, Coulomb matrices with three-times differentiable Matérn kernels). Our tests validate that AD is a viable method for performing geometry optimization with GP surrogate models on small molecules.

## 1 Introduction

Potential energy surfaces (PESs) are fundamental to theoretical chemistry. The PES of a system of  $N_A$  atoms takes the form of a  $(3N_A - 6)$ -dimensional hypersurface and is the solution to the multi-electron Schrödinger equation under a fixed-nuclei approximation. Crucially, one can study the chemical properties of a system (*e.g.*, thermodynamics, activation energies, reaction mechanisms, and reaction rates) by identifying the local extrema on its PES, a process known as *geometry optimization*. In particular, local minima correspond to reactants, products, and reactive intermediates, while first-order saddle points correspond to transition-state structures.

There is a rich line of work on developing electronic-structure methods (*e.g.*, density-

functional theory) that approximately evaluate the electronic-structure energy of a system for a given geometry (*i.e.*, position of nuclei). These methods can be applied to perform, *e.g.*, single-point energy (SPE) calculations to evaluate the electronic-structure energy of a given point on a PES. However, calculating a system’s energy at even a single geometry with relatively high accuracy (*i.e.*, to an error under 1 kcal/mol) can be computationally expensive. In particular, the computational cost of high-level electronic-structure methods scales unfavorably with respect to system size (*i.e.*, number of basis functions). Consequently, it is often best practice to keep to a minimum the number of SPE calculations required for convergence in geometry optimization. (Alternatively, one can attempt to accelerate the SPE computation itself.)

One promising approach for managing the number of SPE calculations is to use a surrogate for a PES with the following properties: (1) it is computationally inexpensive to evaluate the surrogate at any point, and (2) the surrogate can incorporate information from an electronic-structure calculation (*e.g.*, energy and force) to refine its approximation of the PES. The first property allows for rapid but potentially inaccurate evaluations, while the second enables us to refine the surrogate’s accuracy in a controlled manner by determining when and where to pay the cost of an electronic-structure calculation.

Researchers have demonstrated that *Gaussian processes* (GPs) form an effective surrogate for representing the PESs of molecular systems,<sup>1–5</sup> and can reduce the number of SPE calculations required for geometry optimization (*e.g.*, finding local extrema and minimum-energy reaction paths<sup>6–9</sup>). One particularly favorable property of GPs for modeling PESs is that they can encode both the PES and its gradient in a consistent manner. As a result, GP surrogates naturally express the physical relationship between energy and forces, and can exploit electronic-structure methods providing information on both.

The encouraging results on GPs as PES surrogates (*e.g.*, see<sup>1,2,10,11</sup>) warrant further study, especially since GPs form a large and flexible class of models parameterized by *prior mean*

and *kernel* functions. The prior mean function for a surrogate GP encodes prior beliefs about the functional form of a PES (*e.g.*, the asymptotic behavior of stretching a bond from equilibrium length to the dissociation limit) while the kernel function constrains its shape (*e.g.*, it should be “smooth”).

A primary barrier to exploring the GP design space, particularly when incorporating force information, is that the user must manually implement the first and second-order derivatives for a GP’s mean and kernel functions. Prior work<sup>6,7,12,13</sup> has largely examined GPs with mean and kernel functions that have simple derivatives (*e.g.*, constant mean functions with constant 0 derivative or Matérn kernels with well-known derivatives). As researchers begin to explore more complex mean and kernel functions that model more of the chemistry and physics, the burden of deriving and implementing derivatives will only increase.

In this paper, we introduce a Python library called mad-GP—**m**olecular/**a**material and **a**utodiff GPs—that enables researchers to more systematically explore GPs as PES surrogates (Section 2). Users of mad-GP need only specify the functional form of the prior mean and kernel functions parameterizing a GP as Python code, and the library implements the first and second-order derivatives. As proof of concept, we have written non-constant mean functions (*e.g.*, Leonard-Jones potential) and kernel functions based on chemical descriptors (*e.g.*, Coulomb matrices,<sup>14</sup> Soap<sup>15</sup>) in mad-GP.

mad-GP handles differentiation of mean and kernel functions by applying *automatic differentiation* (AD, or autodiff), a technique from computer science which has played a significant role in the rise of deep learning. The hope is that researchers, unencumbered from the task of manually implementing derivatives, will be able to experiment with functions that better capture the chemical structure (*e.g.*, by design or by learning from data).

We apply mad-GP to the task of geometry optimization for small molecules to demonstrate its use (Section 3). Notably, mad-GP enables users to fit GP surrogates to energies and/or forces. We use this functionality to compare

the efficacy of two popular approaches to GP surrogates for geometry optimization: (1) those that fit energies and forces (*e.g.*,<sup>6,7,12,13</sup>) and (2) those that fit forces exclusively (*e.g.*,<sup>3,10,16</sup>). We also evaluate a third approach, namely, the energy-only approach. To the best of our knowledge, there has been relatively little work comparing these approaches. We show that both the first two approaches are comparable at reducing the number of SPE calculations, but that force-only GPs are more robust for optimization because they do not require an additional step to be applied during optimization. We also perform a preliminary study on the use of non-constant priors and hierarchical kernels in GP PES surrogates. We confirm that constant mean functions and Matérn kernels work well as reported in the literature, although our studies also identify several other promising combinations (*e.g.*, Coulomb matrices with three-times differentiable Matérn kernels). Our tests validate that AD is a viable method for performing geometry optimization with GP surrogate models on small molecules.

## 2 Methods

In this section, we review PESs and PES surrogate models (2.1). In the pursuit of self-containment, we also give a brief introduction of Gaussian processes (Section 2.2); likewise, we later discuss AD, which is central to the implementation of mad-GP (Section 2.4).

### 2.1 Models of Potential Energy Surfaces

PESs are complex hypersurfaces and there is a rich line of work on characterizing their properties and their connections to chemistry (*e.g.*, see<sup>17</sup> for an earlier account). Notably, Fukui et al.<sup>18</sup> introduces the formal concept of an intrinsic reaction coordinate for a PES which enables us to trace the “path” of a chemical reaction on a PES between products and/or reactants through transition states (*e.g.*, see<sup>19,20</sup>), and consequently, study chemical reactions (*e.g.*, see<sup>21–26</sup> for PES models used to

study chemical reactions). Mezey in a series of works<sup>27–36</sup> studies PESs from a topological perspective. One important result shows that a PES can be formally partitioned into cachment regions (*i.e.*, basins of attractions around local minima), thus justifying the view that PESs do indeed encode the information necessary for studying chemical reactions (*i.e.*, reactants, products, transition states). Crucially, the topological perspective enables a description of a PES in a coordinate-free manner, which provides insight into the design of computational representations of PESs that are invariant to physical symmetries (*e.g.*, by working with a unique representation of an equivalence class given by a quotient space).

The mathematical analysis of PESs inspires computational representations of PESs for modeling and simulation purposes (*e.g.*, see the software POTLIB<sup>37</sup>), including many-mode representations<sup>38–44</sup> as well as sum-of-product representations.<sup>45–48</sup> Recently, many ML models have been explored as candidates for modeling PESs. ML models take a statistical point-of-view and attempt to model the PES directly from data in line with earlier work that uses simpler models (*e.g.*, see<sup>49–51</sup>). These models thus have to pay more attention to capturing important physical properties of PESs with the hope of having better computational scaling potential. These ML models include neural networks,<sup>52–63</sup> kernel methods,<sup>1–4</sup> and Gaussian processes.<sup>6,7,12,13</sup> Some ML models are designed specifically for modeling PESs (*e.g.*, Gaussian Approximation Potentials,<sup>1,2</sup> Behler-Parrinello Neural Networks,<sup>56–58</sup> and q-Spectral Neighbor Analysis Potentials<sup>64</sup>).

The choice of PES representation is informed by the task at hand. Neural network surrogates have demonstrated success in regression settings where we would like to directly predict a quantity (*e.g.*, energy, force, dipole moments) from a geometry. However, it is expensive to train a neural network and so their use case in geometry optimization is limited. GP surrogates, on the other hand, have demonstrated success in geometry optimization because they are trainable in an online manner and also provide a closed form solution that is searchable by

gradient descent (Section 2.2). Moreover, GPs can also express conservation of energy. However, GPs are impractical to train when the number of examples is large (cubic scaling in the number of examples).

## 2.2 Gaussian Processes

A *Gaussian process* (GP) defines a probability distribution on a class of real-valued functions as specified by a *mean* function  $\mu : \mathbb{R}^D \rightarrow \mathbb{R}^K$  and a (symmetric and positive definite) *kernel* function  $k : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}^{(K \times K)}$ . The notation

$$f \sim \text{GP}(\mu, k) \quad (1)$$

indicates that  $f : \mathbb{R}^D \rightarrow \mathbb{R}^K$  is a function drawn from a GP with mean  $\mu$  and kernel  $k$ . The connection with multivariate Gaussian distributions is that

$$f(\mathbf{X}) \sim \mathcal{N}(\mu(\mathbf{X}), K(\mathbf{X}, \mathbf{X})) \quad (2)$$

where the notation  $f(\mathbf{X})$  is shorthand for

$$f(\mathbf{X}) = (f(\mathbf{x}_1) \dots f(\mathbf{x}_M))^T \quad (3)$$

for  $\mathbf{x}_1, \dots, \mathbf{x}_M \in \mathbb{R}^D$  (similarly, for  $\mu(\mathbf{X})$ ) and the covariance matrix  $K(\mathbf{X}, \mathbf{Y})$  is defined as

$$K(\mathbf{X}, \mathbf{Y}) = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{y}_1) & \dots & k(\mathbf{x}_1, \mathbf{y}_L) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_M, \mathbf{y}_1) & \dots & k(\mathbf{x}_M, \mathbf{y}_L) \end{pmatrix} \quad (4)$$

with  $\mathbf{X}^{(D \times M)} = (\mathbf{x}_1 \dots \mathbf{x}_M)$  (similarly,  $\mathbf{Y}^{(D \times L)} = (\mathbf{y}_1 \dots \mathbf{y}_L)$ ).

We will review two properties of GPs that make them useful as PES surrogates: (1) GP surrogates support gradient-based optimization which can be used in application such as geometry optimization (Section 2.2.1) and (2) GP surrogates support fitting force information (Section 2.2.3). Before we do this, we start by highlighting aspects of GPs that are usually left abstract from a mathematical point-of-view that are important for its application to PESs. For more background on GPs, we refer the reader to standard references (*e.g.*, see Williams et al.<sup>65</sup>).

### 2.2.1 Kernels for atomistic systems

As a reminder, GPs are parameterized by a kernel function  $k : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}^{(K \times K)}$ . Intuitively, the kernel function can be thought of as a measure of similarity between two vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ . For the use case of modeling PESs, the vectors  $\mathbf{x}$  and  $\mathbf{y}$  are then vector encodings of two molecular structures  $m_{\mathbf{x}}$  and  $m_{\mathbf{y}}$ . A *molecular structure*  $m_{\mathbf{x}}$  describes a molecule with  $N_A$  atoms by giving (1) the atomic nuclear charges ( $Z_1 \dots Z_{N_A}$ ), (2) the masses ( $W_1 \dots W_{N_A}$ ), and nuclear positions ( $x^1 \dots x^{3N_A}$ )<sup>T</sup> in Cartesian (xyz) coordinates. Consequently, we will need to convert molecular structures into vectors so that kernel functions can operate on them.

For example, if molecular structures are described in Cartesian coordinates, then  $D$  would be  $3N_A$  where  $N_A$  is the number of atoms in the system. The atomic charges and masses would be ignored. The encoding would then be an identity function on the nuclear positions, and  $\mathbf{x} = (x^1 \dots x^{3N_A})^T = (\mathbf{x}^1 \dots \mathbf{x}^{N_A})^T$  would then correspond to a particular geometry in Cartesian coordinates represented by a column vector consisting of coordinates of all atoms. We could then use a standard GP kernel such as a (twice-differentiable, or  $p = 2$ ) Matérn kernel defined as

$$k_m(\mathbf{x}, \mathbf{y}) = \sigma_M^2 \left( 1 + \frac{\sqrt{5}\rho}{l} + \frac{5\rho^2}{3l^2} \right) \exp\left(\frac{-\sqrt{5}\rho}{l}\right) \quad (5)$$

where  $\rho = \|\mathbf{x} - \mathbf{y}\|$ , and  $\sigma_M$  and  $l$  are hyperparameters, to measure the geometric similarity between two structures. In full detail, the kernel function would be

$$k_m(\text{xyz}(m_{\mathbf{x}}), \text{xyz}(m_{\mathbf{y}})) \quad (6)$$

where xyz converts a molecular structure into its xyz coordinates (an identity function on the nuclear positions in this case). This kernel is the one used in many state-of-the-art works on GP surrogates (*e.g.*, see<sup>12</sup>). This approach, while effective, leaves at least two directions of potential improvement.

First, the description of a molecule in terms

of xyz coordinates does not respect basic physical invariances such as permutation invariance. To solve these problems, researchers have developed chemical descriptors (*e.g.*, global descriptors such as Coulomb matrices<sup>14</sup> and local descriptors such as Soap<sup>15</sup> that describe a molecule as a set of local environments around its atoms) that describe molecules in a way such that these physical invariances are respected. We can take advantage of these more sophisticated (and physically accurate) descriptors by defining a GP kernel

$$k_m(d(m_{\mathbf{x}}), d(m_{\mathbf{y}})) \quad (7)$$

where  $d : \mathcal{M} \rightarrow \mathcal{D}$  is some descriptor,  $\mathcal{M}$  is the space of molecular structures, and  $\mathcal{D}$  is the target space of a descriptor (*e.g.*,  $\mathcal{D} = \mathbb{R}^D$ ). Different descriptors may be better for different kinds of molecules and chemical systems.

Second, once we generalize from xyz coordinates to descriptors, the Matérn kernel  $k_m$  may no longer be a sensible choice as a measure of similarity. For example, local descriptors such as Soap descriptors produce a set of variable-sized local environments that describe a neighborhood of a system from the perspective of each atom in the system. Consequently, we may also be interested in kernels  $k_d : \mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}^{(K \times K)}$  that compare descriptors directly, resulting in a GP kernel of the form

$$k_d(d(m_{\mathbf{x}}), d(m_{\mathbf{y}})). \quad (8)$$

Note that descriptors of different molecules may return encodings of different sizes (*e.g.*, the xyz encoding of molecules with different number of atoms has different sizes), and so these kernel functions  $k_d$  are not necessarily the standard ones defined on a fixed-size vector space. For instance, in the case of local descriptors, we may need to perform structure matching (*e.g.*, see<sup>66</sup>).

In short, while we can leave aspects of a GP such as the kernel function  $k$  abstract, the use case of PESs highlights a hierarchical and compositional structure. The choice of kernel depends on factors such as (1) what invariants we hope to capture (*e.g.*, rotational and permuta-

tional invariance), (2) whether we are working with molecules or materials, and (3) do we need to compare molecules with a different number of atoms and atomic types. Constructing the appropriate kernels that capture the relevant chemistry and physics can lead to improved GP PES surrogates.

### 2.2.2 Geometry optimization with GPs

Previously, we saw that we could define hierarchical kernels with descriptors for PES modeling. Although this may be interesting as a theoretical exercise, it is not useful from a practical perspective if we cannot effectively compute with a GP surrogate defined in this manner. For example, in an application such as using a GP surrogate for geometry optimization, we would like to be able to extract a physical geometry out. In this section, we will see that defining hierarchical kernels interacts nicely with gradient-based surrogate optimization. Before we explain this, we will first review how to update a GP surrogate with data obtained from an electronic-structure calculation. This produces a *posterior* surrogate which will more closely approximate a PES.

**Fitting a GP** We can fit GPs to observed data, called *Gaussian Process Regression* (GPR), to refine the class of functions defined by a GP in a principled manner (*e.g.*, *maximum a posteriori* or MAP inference) by solving a linear system of equations. In the context of modeling a PES of a molecule, the observed data would be a set of observations  $\{(\mathbf{x}_1, E_1), \dots, (\mathbf{x}_N, E_N)\}$  (*i.e.*, *training set*) where each  $\mathbf{x}_i$  is a vector encoding of a molecule's geometry and  $E_i$  the corresponding energy as obtained from an electronic-structure method. (When forces are available, we can additionally fit  $\mathbf{F} = (\mathbf{F}_1 \dots \mathbf{F}_N)^T$  (see Section 2.2.3).)

In practice, we perform GPR where we assume our observations are corrupted with some independent, normally distributed noise  $\epsilon \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$  where  $\mathbf{I}$  is the identity matrix, which

corresponds to the model

$$f \sim GP(\mu, k) \quad (9)$$

$$E_{\mathbf{x}} = f(\mathbf{x}) + \epsilon. \quad (10)$$

We can equivalently view this from the multivariate Gaussian perspective as using the covariance matrix

$$K_\sigma(\mathbf{X}, \mathbf{X}) = \sigma^2 \mathbf{I} + K(\mathbf{X}, \mathbf{X}) \quad (11)$$

because the noise is also normally distributed. The noise  $\epsilon$  can be used to model uncertainty in a SPE calculation (generated by standard quantum chemistry computations), *e.g.*, the energy convergence threshold used or the intrinsic error of the electronic-structure method applied.

The system of equations to solve when fitting energy information is

$$K_\sigma(\mathbf{X}, \mathbf{X})\boldsymbol{\alpha} = \mathbf{E} - \mu(\mathbf{X}) \quad (12)$$

where  $\mathbf{X} = (\mathbf{x}_1 \dots \mathbf{x}_N)$  and  $\mathbf{E} = (E_1 \dots E_N)^T$  are observations, and  $\boldsymbol{\alpha} \in \mathbb{R}^N$  indicates the weights to solve for. These equations can be solved with familiar algorithms such as Cholesky decomposition (cubic complexity, Scipy<sup>67</sup> implementation). When GP surrogates are used in applications such as geometry optimization, we can use incremental Cholesky decomposition (quadratic complexity, custom implementation following Lawson et al.<sup>68</sup>). Note that GPR with noise has the added benefit for increasing numerical stability of linear system solvers.

**Gradient-based optimization** To evaluate a GP surrogate that is fit to a training set  $(\mathbf{X}, \mathbf{E})$ , we compute the function

$$f^*(\mathbf{x}) = \sum_{i=1}^N k(\mathbf{x}, \mathbf{x}_i)\alpha_i + \mu(\mathbf{x}) \quad (13)$$

where  $f^*$  is the *posterior mean* (to distinguish it from the prior  $f$ ) and  $\boldsymbol{\alpha}$  are weights that are determined during the training process. The posterior mean gives the GPs prediction of the energy.

Gradient-based optimization of the posterior

mean corresponds to using the gradient

$$\nabla f^*(\mathbf{x}) = \sum_{i=1}^N (\nabla_1 k)(\mathbf{x}, \mathbf{x}_i)\alpha_i + (\nabla \mu)(\mathbf{x}) \quad (14)$$

to perform optimization ( $\nabla_1 k = \frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x}}$ ). Consequently, gradient-based optimization of a GP surrogate requires the first-order derivatives of the mean ( $\nabla \mu$ ) and kernel functions ( $\nabla_1 k$ ). When mean and kernel functions are sufficiently complicated, a GP user will need to implement these derivatives in order to perform gradient-based optimization of the posterior mean. This highlights the benefit of having a method that enables us to automatically obtain derivatives of mean and kernel functions.

We return now to the problem of ensuring that using hierarchical kernels does not prevent us from using it in applications such as geometry optimization. As a reminder, the issue is that if we use a descriptor  $d$  to define a GP kernel, then we will need to somehow recover physical coordinates such as xyz coordinates during geometry optimization. One approach pursued in the literature (*e.g.*, see Meyer et al.<sup>69</sup>) is to use an inverse mapping  $d^{-1}$  to map back into physical coordinates. While this may exist for "simple" descriptors, it is unlikely to exist or be easily computable for more complex descriptors.

Let us define a kernel  $k_d$  with an arbitrary descriptor  $d$  as

$$k_d(d \circ \text{xyz}(m_{\mathbf{x}}), d \circ \text{xyz}(m_{\mathbf{y}})) \quad (15)$$

to highlight that we start in xyz coordinates. Then the gradient of this kernel with respect to xyz coordinates can be computed by the chain-rule. Because the posterior mean can be written as a linear combination of gradients of the kernel (*e.g.*, see (14)), an optimization procedure will thus be able to optimize directly in xyz coordinates while also factoring the comparison of molecules through a descriptor, giving us the best of both worlds. Note that this works for arbitrarily complex kernels and descriptors so long as they are differentiable. (We will see with AD that strict differentiability is not required.)

So far, there is no need for second-order

derivatives. As we will see in the next section, second-order derivatives will appear when we use GPs that fit force information.

### 2.2.3 Models with conservative forces

GPs have the following remarkable property: if

$$f \sim \text{GP}(\mu, k) \quad (16)$$

then

$$\nabla f \sim \text{GP}(\nabla \mu, \nabla_1 k \nabla_2^T) \quad (17)$$

when  $\nabla_1 k \nabla_2^T = \frac{\partial^2}{\partial \mathbf{x} \partial \mathbf{y}} k(\mathbf{x}, \mathbf{y})$  exists. ( $\nabla_1 k = \frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x}}$  and  $k \nabla_2^T = \frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{y}^T}$ .) The consequence for PES modeling is that we can define a specific GP kernel that constrains the class of functions a GP describes to be consistent with its gradients so that the physical relationship between energy and force (*i.e.*, the negative gradient of a PES) is encoded. In particular, the GP

$$\begin{pmatrix} f \\ \nabla f \end{pmatrix} \sim \text{GP}(\tilde{\mu}, \tilde{k}) \quad (18)$$

with

$$\tilde{\mu} = \begin{pmatrix} \mu \\ \nabla \mu \end{pmatrix} \quad (19)$$

and

$$\tilde{k}(\mathbf{x}, \mathbf{y}) = \begin{pmatrix} k(\mathbf{x}, \mathbf{y}) & k \nabla_2^T(\mathbf{x}, \mathbf{y}) \\ \nabla_1 k(\mathbf{x}, \mathbf{y}) & \nabla_1 k \nabla_2^T(\mathbf{x}, \mathbf{y}) \end{pmatrix} \quad (20)$$

accomplishes this.

Second-order derivatives appear when we leverage force information during both GPR and posterior optimization. To see the former, observe that the system of equations to solve when fitting both energies and forces is

$$\sum_{j=1}^N \left( \tilde{k}(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij} \begin{pmatrix} \sigma_e^2 & \mathbf{0} \\ \mathbf{0} & \sigma_f^2 \mathbf{I} \end{pmatrix} \right) \begin{pmatrix} \alpha_j \\ \beta_j \end{pmatrix} = \begin{pmatrix} E_i \\ -\mathbf{F}_i \end{pmatrix} - \begin{pmatrix} \mu(\mathbf{x}_i) \\ \nabla \mu(\mathbf{x}_i) \end{pmatrix} \quad (21)$$

for the weights  $\alpha_i$  and  $\beta_i$  where  $\sigma_e$  is a noise parameter for the energy and  $\sigma_f$  is a noise parameter for the force. The second-order derivatives appear through the kernel  $\tilde{k}$  used to set up the system of linear equations. To see the

```

1 def mean(atoms_x, x):
2     """
3     :param atoms_x: Atoms
4     :param x: jax.DeviceArray[D]
5     :return: jax.Floating64
6     """
7     return 0.0
8
9 def kernel(atoms_x1, atoms_x2, x1, x2):
10    """
11    :param atoms_x1: Atoms
12    :param atoms_x2: Atoms
13    :param x1: jax.DeviceArray[D]
14    :param x2: jax.DeviceArray[D]
15    :return: jax.Floating64
16    """
17    w = 1.0; l = 0.4
18    scaled_diff = (x - y) / l
19    r2 = jnp.dot(scaled_diff, scaled_diff)
20    r = jnp.sqrt(r2 + 1e-8)
21    root_five = 2.23606798
22    c = w*w*(1.0 + root_five*r + 5.0/3.0*r*r)
23    e = jnp.exp(-root_five*r)
24    return c * e
25
26 gppes = EnergyForceGPES(mean, kernel)

```

Figure 1: Example of using mad-GP. This is the entire code required for the user to write in order to experiment with a constant mean function and Matérn kernel ( $k_m$ , see (5)) which are used in state-of-the-art results.

latter, observe that the posterior predictions of the energy and gradient of the energy are given by

$$f^*(\mathbf{x}) = \sum_{i=1}^N (k(\mathbf{x}, \mathbf{x}_i) \alpha_i + k \nabla_2^T(\mathbf{x}, \mathbf{x}_i) \beta_i) + \mu(\mathbf{x}) \quad (22)$$

and

$$(\nabla f)^*(\mathbf{x}) = \sum_{i=1}^N (\nabla_1 k(\mathbf{x}, \mathbf{x}_i) \alpha_i + \nabla_1 k \nabla_2^T(\mathbf{x}, \mathbf{x}_i) \beta_i) + \nabla \mu(\mathbf{x}). \quad (23)$$

Note that the gradient of the posterior mean for the energy is given by the prediction of the

posterior mean for the gradient.

$$\begin{aligned}
& \nabla f^*(\mathbf{x}) && \text{(gradient of posterior energy)} \\
&= \nabla_1 \left( \sum_{i=1}^N (k(\mathbf{x}, \mathbf{x}_i) \alpha_i + k \nabla_2^\top(\mathbf{x}, \mathbf{x}_i) \boldsymbol{\beta}_i) \right) + \nabla \mu(\mathbf{x}) \\
&&& \text{(definition)} \\
&= \sum_{i=1}^N (\nabla_1 k(\mathbf{x}, \mathbf{x}_i) \alpha_i + \nabla_1 k \nabla_2^\top(\mathbf{x}, \mathbf{x}_i) \boldsymbol{\beta}_i) + \nabla \mu(\mathbf{x}) \\
&&& \text{(linearity)} \\
&= (\nabla f)^*(\mathbf{x}). \\
&&& \text{(prediction of posterior gradient)}
\end{aligned}$$

Thus the second-order derivative comes into play either by differentiating the posterior mean  $f^*$  or by performing a posterior gradient prediction using  $(\nabla f)^*$ .

For completeness, we also give the equations for fitting forces exclusively.

$$\sum_{j=1}^N (\nabla_1 k \nabla_2^\top(\mathbf{x}_i, \mathbf{x}_j) + \delta_{ij} \sigma_f^2 \mathbf{I}) \boldsymbol{\beta}_j = -\mathbf{F}_i - \nabla \mu(\mathbf{x}_i)$$
(24)

We use  $\text{GP}_E$ ,  $\text{GP}_F$ , and  $\text{GP}_{EF}$  to refer to the GPS that fit energies exclusively (solving (12)), forces only (solving (24)), and energies and forces (solving (21)) respectively.

### 2.3 A Gaussian Process Library for Representing Potential Energy Surfaces

We introduce mad-GP first via an example (Figure 1). The code in Figure 1 corresponds to defining the model

$$\begin{pmatrix} f \\ \nabla f \end{pmatrix} \sim \text{GP}_{EF}(\mathbf{0}, \tilde{k}_m) \tag{25}$$

$$\begin{pmatrix} E_i \\ -\mathbf{F}_i \end{pmatrix} = \begin{pmatrix} f(\mathbf{x}_i) \\ \nabla f(\mathbf{x}_i) \end{pmatrix} + \begin{pmatrix} \epsilon_e \\ \epsilon_f \end{pmatrix} \tag{26}$$

where  $\epsilon_e \sim \mathcal{N}(0, \sigma_e^2)$  and  $\epsilon_f \sim \mathcal{N}(0, \sigma_f^2 \mathbf{I})$ .

The first portion of the code (lines 1 – 7) defines a mean function `mean` implementing a constant 0 mean. The function takes two arguments (1) `atoms_x` and (2) `x`. The first ar-

gument `atoms_x` contains a molecular structure’s atomic charges and masses (packaged in the type `Atoms`). The second argument `x` corresponds to a vector encoding of a molecular structure which may be a function of a molecular structure’s atomic charges, masses, and nuclear positions (packaged in the type `jax.DeviceArray[D]` which encodes a  $\mathbb{R}^D$  vector). The reason that `x` does not just give the nuclear positions of a molecular structure is because we may want to use a chemical descriptor that is a function of nuclear positions (among other information). Thus `atoms_x` and `x` together encode all the information available from a molecular structure. The constant mean function, by definition, ignores all arguments and returns a constant 0.

The second portion of the code (lines 9 – 24) defines a kernel function `kernel` that implements the Matérn kernel ( $k_m$ , see (5)). The signature of the function is similar to the case for the prior. The arguments `atoms_x1` and `atoms_x2` are not used because the Matérn kernel does not require either atomic charge or atomic mass in a structure.

The last portion of the code (line 26) creates a GP surrogate that fits both energies and forces by supplying (1) the user-defined mean function `mean` and (2) the user-defined kernel function `kernel`. The default vector encoding is xyz coordinates, *i.e.*, an identity function on nuclear positions of a molecular structure. This concludes the amount of code that the user is required to write and illustrates the main point: the user does not need to derive or implement derivatives of the mean or kernel functions. Consequently, arbitrarily complex mean and kernel functions can be defined so long as they can be programmed in Python and use mathematical primitives from an AD library.

#### 2.3.1 Support for kernels for atomistic systems

When we introduced kernels for atomistic systems (Section 2.2.1), we saw that there were many design choices involved in constructing a GP kernel including (1) choice of descriptor and

$d_g \in \{\text{xyz} : \mathbb{R}^{3N_A} \rightarrow \mathbb{R}^D, \text{coulomb} : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathbb{R}^{(N_A \times N_A)}, \dots\}$	(global descriptors)
$d_\ell \in \{\text{soap} : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A}, \dots\}$	(local descriptors)
$\oplus \in \{\text{flatten} : \mathbb{R}^{(M \times L)} \rightarrow \mathbb{R}^{ML}, \text{average} : \mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A} \rightarrow \mathbb{R}^D, \dots\}$	(descriptor-to-vector)
$k \in \{\text{dot} : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}, \text{matern} : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}, \dots\}$	(vector kernels)
$k_d \in \{\text{pairwise} : (\mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}) \times (\mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A}) \times (\mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A}) \rightarrow \mathbb{R}, \dots\}$	(structure kernels)

Figure 2: A small combinator library for expressing GP kernels for PES surrogates.

(2) how to compare descriptors. For example, as a quick test of a simple kernel that is (1) invariant to translation and rotation and (2) takes into account atomic charges, we may choose the kernel

$$k_m(\text{flatten} \circ \text{coulomb}(m_x), \text{flatten} \circ \text{coulomb}(m_y)) \quad (27)$$

where  $k_m$  is a Matérn kernel,  $\text{flatten} : \mathbb{R}^{(M \times L)} \rightarrow \mathbb{R}^{ML}$  flattens a matrix into a vector,  $\text{coulomb}$  is a Coulomb descriptor,<sup>14</sup> and  $m_x$  and  $m_y$  are molecular structures. We could decide later that a Matérn kernel on flattened Coulomb matrices is not ideal and opt to compare Coulomb matrices directly as in

$$k'(\text{coulomb}(m_x), \text{coulomb}(m_y)) \quad (28)$$

for some kernel  $k' : \mathbb{R}^{(N_A \times N_A)} \times \mathbb{R}^{(N_A \times N_A)} \rightarrow \mathbb{R}^K$  defined on matrices directly (*e.g.*, using a matrix norm). To assist the user in managing the multitude of kernels one could construct, mad-GP provides a small *combinator* library (Figure 2). A combinator library identifies (1) primitive building blocks and (2) ways to put those building blocks together.

**Global and local descriptors** The base building blocks that mad-GP’s combinator library provides are local and global descriptors, similar in spirit to libraries such as DSCRIBE.<sup>70</sup> (The difference with libraries like DSCRIBE is that our combinator library supports AD, and consequently, can derive derivatives of descriptors automatically from their implementation.) A global descriptor  $d_g : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathbb{R}^D$

convert atomic charges, masses, and nuclear positions into vector encodings of the molecule. Cartesian coordinates  $\text{xyz}$  are a typical example. Another example includes Coulomb matrices ( $\text{coulomb} : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathbb{R}^{(N_A \times N_A)}$ ). A local descriptor  $d_\ell : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A}$  produces atom-centered encodings of the molecules  $\mathcal{D}^{N_A}$ . Examples of local descriptors include Soap descriptors ( $\text{soap} : \text{Atoms} \times \mathbb{R}^{3N_A} \rightarrow \mathcal{D}_1 \times \dots \times \mathcal{D}_{N_A}$ ) which are a differentiable description of a molecular structure that preserves rotational, translational, and permutation invariance.

We have prototype implementations of Coulomb matrix descriptors<sup>14</sup> and Soap descriptors<sup>15</sup> in mad-GP to demonstrate its flexibility and expressiveness. Note that in some cases some of the building blocks of a kernel are not differentiable. For example, when we sort the entries of a Coulomb matrix to obtain permutation invariance (see Section 3.2) or when we use a topological algorithm such as Munkres algorithm<sup>71</sup> to match the structure of two molecules, these descriptors are not differentiable. They can still be handled with mad-GP because AD will assign a *sub-gradient* to the points that the descriptor are not differentiable at.

**Descriptor-to-vector combinator** Once we have described a molecular structure in terms of descriptors, mad-GP provides combinator  $\oplus$  that convert descriptors into vectors. Examples of combinations include  $\text{flatten} : \mathbb{R}^{(M \times L)} \rightarrow \mathbb{R}^{ML}$  which flattens a matrix into a vector (as in the Coulomb descriptor example).

The combinator `average` :  $\mathcal{D}^{N_A} \rightarrow \mathcal{D}$  takes a local descriptor and averages the representation over all atom centers as in

$$\frac{1}{N_A} \sum_{i=1}^{N_A} (d_\ell(\text{atoms}_x, \mathbf{x}))_i. \quad (29)$$

For example, we can average each local environment produced by a Soap descriptor to produce a global descriptor (*e.g.*, see outer average in DScribe’s Soap implementation<sup>70</sup>).

**Vector kernels** The kernel functions  $k$  are the ordinary symmetric and positive-definite functions used to parameterize GPs and include dot products `dot` and Matérn kernels `matern`.

**Structure kernels** Finally, the structure kernels are kernels that are used to compare local descriptors. One simple structure kernel `pairwise` :  $(\mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}) \times (\mathcal{D}_1 \times \cdots \times \mathcal{D}_{N_A}) \times (\mathcal{D}_1 \times \cdots \times \mathcal{D}_{N_A}) \rightarrow \mathbb{R}$  performs a pairwise comparison between every pair of atoms in a local environments using a kernel  $k : \mathcal{D} \times \mathcal{D} \rightarrow \mathbb{R}$  that compares the descriptor at two atom centers. There are also more advanced structure matching kernels such as those based on topological algorithms<sup>71</sup> or optimal transport<sup>66</sup> that improve upon pairwise matching by only comparing “suitably-aligned” portions of each molecular structure’s local descriptor. We leave the implementation of more advanced structure kernels for future work.

Using these combinator, the Matérn kernel can be written as

$$\begin{aligned} \text{kernel}(\text{atoms}_x, \text{atoms}_y, \mathbf{x}, \mathbf{y}) &= \\ k_{\text{Matérn}}(\text{xyz}(\text{atoms}_x, \mathbf{x}), \text{xyz}(\text{atoms}_y, \mathbf{y})) \end{aligned}$$

to explicitly highlight that we are using xyz descriptors of a molecule.

### 2.3.2 Support for gradient-based surrogate optimization

mad-GP supports gradient-based optimization of a trained GP surrogate by exposing (1) a prediction of the posterior energy and (2) the gradient of the posterior energy. As a reminder,

this functionality is useful for geometry optimization. How the gradient is obtained depends on whether we are using  $\text{GP}_E$ ,  $\text{GP}_F$ , or  $\text{GP}_{EF}$ .

**Energy** For  $\text{GP}_E$ , the posterior mean gives the prediction of the energies as

$$f^*(\mathbf{x}) = \sum_{i=1}^N k(\mathbf{x}, \mathbf{x}_i) \alpha_i + \mu(\mathbf{x}) \quad (30)$$

and the gradient of the posterior mean

$$\nabla f^*(\mathbf{x}) = \sum_{i=1}^N \nabla_1 k(\mathbf{x}, \mathbf{x}_i) \alpha_i + \nabla \mu(\mathbf{x}) \quad (31)$$

is computed with (*reverse-mode*) AD. (See Section 2.4 for the distinction between reverse-mode and forward-mode.)

**Force** For  $\text{GP}_F$ , observe that the posterior prediction of the gradient is

$$g^*(\mathbf{x}) = \sum_{i=1}^N \nabla_1 k \nabla_2^\top(\mathbf{x}, \mathbf{x}_i) \beta_i + \nabla \mu(\mathbf{x}). \quad (32)$$

The second-order derivatives of the kernel are obtained with (reverse-mode then forward-mode) AD. A prediction of the energy is recovered up to a constant via the line integral along the curve  $\gamma_{\mathbf{x}_0}^{\mathbf{x}}$  starting at  $\mathbf{x}_0$  and ending at  $\mathbf{x}$  as

$$E(\mathbf{x}) = \int_{\gamma_{\mathbf{x}_0}^{\mathbf{x}}} g^*(\mathbf{y}) d\mathbf{y} \quad (33)$$

$$= \int_{\gamma_{\mathbf{x}_0}^{\mathbf{x}}} \sum_{i=1}^N \nabla_1 k \nabla_2^\top(\mathbf{x}, \mathbf{x}_i) \beta_i + \nabla \mu(\mathbf{x}) \quad (34)$$

$$= \sum_{i=1}^N k \nabla_2^\top(\mathbf{x}, \mathbf{x}_i) \beta_i + \mu(\mathbf{x}) + C \quad (35)$$

where  $C$  is some constant. (The base point  $\mathbf{x}_0$  is arbitrary and unspecified.) That is, we “compute” the integral using the converse of the gradient theorem for conservative forces: any conservative force field can be written as the gradient of a scalar field. Thus we compute (35) directly by computing  $k \nabla_2^\top$  using (reverse-mode) AD.

**Energy and force** For  $\text{GP}_{\text{EF}}$ , we simply evaluate the posterior mean at a geometry to obtain both the prediction of the energy and (negative) forces. Recall that the gradient of the posterior mean for the energy is given by the prediction of the posterior mean for the gradient.

### 2.3.3 Support for GPs with forces

We implement GPs that fit forces (*i.e.*,  $\text{GP}_F$  and  $\text{GP}_{\text{EF}}$ ) by implementing their respective kernels using AD where appropriate. For instance, for  $\text{GP}_{\text{EF}}(\mu, k)$ , we implement  $\nabla \mu$ ,  $\nabla_1 k$ , and  $k \nabla_2^T$  with (reverse-mode) AD, and  $\nabla_1 k \nabla_2^T$  with (reverse-mode then forward-mode) AD. We will see in Section 2.4 that the run-time of using AD to calculate a second-order term such as  $\nabla_1 k \nabla_2^T$  is proportional to  $DT$  where  $D$  is the number of dimensions of the molecular representation and  $T$  is the run-time of  $k$ . To fit these GPs to training data, we apply the mean and kernel functions to the training data to construct mean vectors and kernel matrices, and use a generic linear system solver to solve the system of equations. Note that AD does not need to interact with the linear solver.

## 2.4 Automatic Differentiation

*Automatic differentiation* (AD) is a technique that algorithmically transforms a computer program evaluating a function into one evaluating the derivative of that function that has the same time-complexity as the program evaluating the original function. This statement is a mouthful—let us unpack this by comparison with (1) finite differences (FD) and (2) symbolic differentiation (SD) in the context of an example. Table 1 summarizes the differences between the methods.

### 2.4.1 AD by example

Consider the function  $f : \mathbb{R} \rightarrow \mathbb{R}$  defined as

$$f(x) = g(x)h(x)i(x) \quad (36)$$

for some  $g, h, i : \mathbb{R} \rightarrow \mathbb{R}$ . The (simplest) finite differences method approximates the derivative

```

1 def ad_f(x):
2     adj_x = 0
3     # The original function
4     t1 = g(x); adj_t1 = 0
5     t2 = h(x); adj_t2 = 0
6     t3 = i(x); adj_t3 = 0
7     t5 = t1 * t2; adj_t5 = 0
8     t6 = t5 * t3; adj_t6 = 0
9     # The derivative
10    adj_t5 += adj_t6 * t3; adj_t3 += adj_t6 * t5
11    adj_t1 += adj_t5 * t2; adj_t2 += adj_t5 * t1
12    adj_x += adj_t3 * ad_i(x) # ad_i = AD(i)
13    adj_x += adj_t2 * ad_h(x) # ad_h = AD(h)
14    adj_x += adj_t1 * ad_g(x) # ad_g = AD(g)
15    return adj_x

```

Figure 3: A pedagogical example of applying  $\text{AD}(f) = \text{ad\_f}$ .

of the function as

```

1 def FD(f):
2     def deriv_f(x):
3         return (f(x + 1e-8) - f(x))/1e-8
4     return deriv_f

```

This approach can easily be implemented in code (as above), operates on programs, and approximately computes the gradient via a linear approximation. The time-complexity of the function is proportional to the time-complexity of the original function  $f$ —we call it twice.

One form of the symbolic derivative for the function  $f$  is

$$\begin{aligned} f'(x) &= g'(x)h(x)i(x) + g(x)h'(x)i(x) \\ &\quad + g(x)h(x)i'(x) \end{aligned} \quad (37)$$

where  $g', h', i'$  are the respective (symbolic) derivatives of  $g, h, i$ . Thus the symbolic derivative is not unique—depending on how we factor this expression, the symbolic derivative may take worst-case exponential time compared to the evaluation of the original function. Consequently, we should be careful when translating symbolic derivatives into code to pick the appropriate factorization.

Listing 3 provides a pedagogical example of applying source-to-source and reverse-mode AD to  $f$ . (mad-GP uses combinations of reverse-

Table 1: A comparison of finite differences (FD), symbolic differentiation (SD), and automatic differentiation (AD) for differentiating a function  $f : \mathbb{R}^D \rightarrow R$ . All three methods support higher-order derivatives.  $=^*$  indicates  $=$  up to floating-point precision.

Method	Time-Complexity	Precision	Unique?	Input/Output
FD	guaranteed $\propto \text{Time}(f)$	$\text{FD}(f) \approx \nabla f$	yes	$\text{FD} : \text{program} \rightarrow \text{program}$
SD	worst-case $\exp(\text{Time}(f))$	$\text{SD}(f) =^* \nabla f$	no	$\text{SD} : \text{math} \rightarrow \text{math}$
AD	guaranteed $\propto \text{Time}(f)$	$\text{AD}(f) =^* \nabla f$	yes	$\text{AD} : \text{program} \rightarrow \text{program}$

mode and forward-mode AD when appropriate, as well as non source-to-source methods of implementation.) We have presented the code in a way such that one can see how this code could be mechanically generated by a computer. Note that each line of code of the original code is translated into

$$\text{tX} = \text{op}(\text{tI}, \text{tJ})$$

and produces the corresponding derivative code

```
adj_tI += tX * ad_op_1st(tI, tJ)
adj_tJ += tX * ad_op_2nd(tI, tJ)
```

where `ad_op_1st` and `ad_op_2nd` correspond to the derivatives of the operation `op` with respect to the first and second arguments. This code can be obtained with a recursive application of AD.

In this form, it is easy to see that the time-complexity of the derivative is proportional to the original code because each line of code in the original function produces at most 2 additional lines of code of the same time-complexity. Upon closer inspection, we see that AD is a method that, as a default, mechanically selects the factorization of the SD that has the appropriate time-complexity. Thus it also computes the derivative exactly, like SD and unlike FD.

AD naturally generalizes from functions of a single variable to functions of multiple variables. Moreover, AD is composable—in the example, we can compute AD( $f$ ) by composing the constituents of its sub-functions AD( $g$ ), AD( $h$ ), AD( $i$ ). Another implication of the compositability of AD is that we can use it to compute higher-order derivatives.

## 2.4.2 AD for higher-order derivatives

AD(AD( $f$ )) computes the second-order derivative of  $f$  because the result of the inner call produces a program evaluating the derivative of  $f$ , which can be fed back into another call to AD. Note that the time-complexity guarantees of AD hold for higher-order derivatives, whereas the time-complexity guarantees for SD are amplified for higher-order derivatives.

For computing higher-order derivatives of functions with multivariate output spaces, the distinction between *reverse-mode* (what was presented) and *forward-mode* AD is important for time-complexity reasons. In particular, for a function  $f : \mathbb{R}^D \rightarrow \mathbb{R}^K$ , computing the derivative with forward-mode AD takes  $D$  passes whereas it takes  $K$  passes for reverse-mode AD. Thus forward-mode is advantageous when  $K > D$  and reverse-mode is advantageous when  $D > K$ . When  $D = K$ , forward-mode is advantageous because it takes less memory. Consequently, for taking second-order derivatives of scalar-valued functions, it is best to perform reverse-mode AD followed by forward-mode AD. We refer the interested reader to references (*e.g.*, see the survey<sup>72</sup> and the references within) for more background.

## 2.4.3 AD in mad-GP

For the use case of PES modeling where we require second-order derivatives when taking forces into account, we should be careful to select an AD library that supports both reverse-mode and forward-mode. Popular AD libraries such as Torch<sup>73</sup> that are specialized for neural networks prioritize reverse-mode AD because training neural networks only requires first-order derivatives of a loss function, *i.e.*,

single-value output. Jax<sup>74</sup> is an AD library that supports both forward-mode and reverse-mode AD. Consequently, Jax is the default AD library in mad-GP. Jax additionally supports Just-In-Time (JIT) compilation. This means that Jax will generate optimized derivative code based on the sizes of vectors and matrices it sees during run-time, further reducing the cost of using a high-level interpreted language like Python for performing efficient numerical computation.

### 3 Results

In the previous section, we introduced and built machinery that enables systematic exploration of GPs for modeling PESs. In this section, we apply mad-GP to geometry optimization of small molecules to validate its use (Section 3.1). We use mad-GP to test the effectiveness of fitting forces ( $\text{GP}_E$  vs.  $\text{GP}_F$  and  $\text{GP}_{EF}$ , Section 3.1.1), the effectiveness of fitting energies for GPs that fit forces ( $\text{GP}_F$  vs.  $\text{GP}_{EF}$ , Section 3.1.2), and a preliminary study on the use of non-constant priors and hierarchical kernels (Section 3.1.3). For completeness, we also qualitatively describe our experiences using AD for constructing GP surrogates for representing molecular PESs (Section 3.2).

#### 3.1 Geometry Optimization

We use mad-GP to perform geometry optimization on the Baker-Chan dataset<sup>75</sup> of molecules (Figure 4), a benchmark dataset consisting of initial-guess structures of small molecules, which is tested in prior work on GP surrogates.<sup>12</sup> Note that in Baker et al.’s work, their goal was to optimize to transition-state structures (*i.e.*, first-order saddle points); however, in this work (as well as in Denzel et al.’s prior work), we optimize to stable local minima. There are many geometry optimization algorithms that use GP surrogates.<sup>12,13,76</sup> We use the geometry optimization algorithm designed for GPs based on the one given in ASE<sup>13,76</sup> (Atomistic Simulation Environment). We choose this algorithm as a baseline so we can focus on varying (1) fitting energies/forces

and (2) mean and kernel functions to explore mad-GP’s capabilities. We summarize the algorithm below.

- Step 1. Update the current training set of  $N$  points— $(\mathbf{X}, \mathbf{E})$ ,  $(\mathbf{X}, \mathbf{F})$ , or  $(\mathbf{X}, (\mathbf{E}, \mathbf{F}))$ —with a call to an electronic-structure method to obtain energies  $E_{N+1}$  and/or forces  $\mathbf{F}_{N+1}$  at a geometry  $\mathbf{x}_{N+1}$ . In this work, we use the PM7 semi-empirical method<sup>77</sup> implemented in MOPAC2016,<sup>78</sup> which we refer to as MOPAC from now on. To manage the computational complexity of fitting a GP, we only keep the last 100 data points.
- Step 2. Fit the GP with the new data point  $(\mathbf{x}_{N+1}, E_{N+1})$ ,  $(\mathbf{x}_{N+1}, \mathbf{F}_{N+1})$ , or  $(\mathbf{x}_{N+1}, (\mathbf{E}_{N+1}, \mathbf{F}_{N+1}))$ . If we are fitting energies, perform a “set maximum” step: set the mean function  $\mu$  to be the maximum energy seen during geometry optimization

$$\mu(\mathbf{x}) = \max_{1 \leq i \leq N+1} E_i. \quad (38)$$

(Note that this does not apply to  $\text{GP}_F$  which we will comment on in Section 3.1.2.)

- Step 3. Use L-BFGS to search the posterior gradient for a new local minimum. Our current implementation uses Scipy’s L-BFGS algorithm<sup>67</sup> using gradients supplied by mad-GP.
- Step 4. If the energy found at the new local minimum is higher than the last energy as determined by a SPE calculation, restart the search for a new local minimum by going to step 2. Otherwise we return the last point  $\mathbf{x}_{N+1}$  found.
- Step 5. Iterate until convergence. Because we are comparing against MOPAC, the convergence criterion we use is the same as MOPAC’s GNORM keyword:

$$\|\mathbf{F}_{N+1}\| \leq 0.0054 \cdot 3N_A \quad (39)$$

where the units are in eV/Å and  $N_A$  is the number of atoms in the system so that  $3N_A$  is the dimensionality of the xyz encoding of a molecule. (A common choice is to also constrain the maximum force,

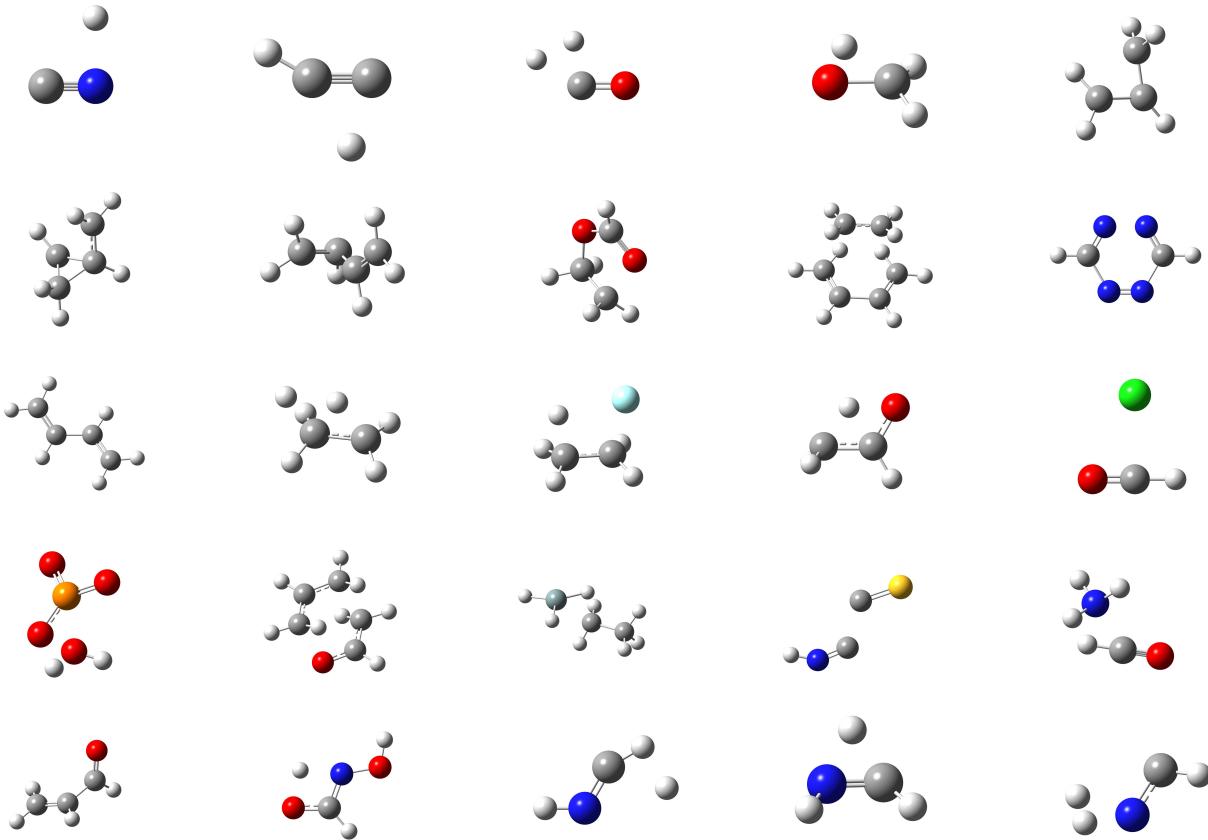


Figure 4: A visualization of initial geometries from systems (system 01 to system 25 arranged left-to-right and top-to-bottom) in the Baker-Chan dataset.<sup>75</sup> White, grey, blue, red, cyan, green, navy green, orange and yellow balls represent H, C, N, O, F, Cl, Si, P, and S atoms, respectively.

*i.e.*, both the  $l_\infty$  norm and the  $l_2$  norm. We do not do that here to be consistent with MOPAC.)

For the purposes of reducing the number of hyper-parameters we need to tune, we do not use a maximum step-size or adaptive length-scale tuning.<sup>12,79</sup> Note that smaller length-scales might be needed closer to convergence. We emphasize that the choices of MOPAC and the optimization algorithm above are for demonstrating the use of mad-GP for geometry optimization. Different electronic-structure calculators and geometry optimization algorithms can easily be used/implemented in mad-GP.

### 3.1.1 Fitting forces

mad-GP supports GP surrogates that fit energies exclusively ( $\text{GP}_E$ ), fit forces exclusively ( $\text{GP}_F$ ), and fit both energies and forces ( $\text{GP}_{EF}$ ).

We use this functionality to compare  $\text{GP}_E$ ,  $\text{GP}_F$ , and  $\text{GP}_{EF}$  with a baseline implementation of L-BFGS in MOPAC to test the efficacy of fitting forces. We perform SPE calculations for both GP and L-BFGS optimizations by using the semiempirical PM7 Hamiltonian within the unrestricted Hartree-Fock formalism (not to be confused with the Hartree-Fock method, which uses the full non-empirical Fock operator with explicitly calculated one- and two-electron integrals) via the MOPAC keyword "PM7 UHF 1SCF GRAD". The self-consistent field (SCF) energy convergence criterion was set to  $10^{-8}$  kcal mol<sup>-1</sup>.

All GPs use a constant 0 mean and a Matérn kernel with  $\sigma_M = 1.0$  and  $l = 0.4$ . We set the energy noise  $\sigma_e = 0.002 \text{ eV}^{1/2}$  and the force noise  $\sigma_f = 0.005 (\text{eV}/\text{\AA})^{1/2}$ . We have tested  $2 \times 10^{-10} \leq \sigma_e \leq 2 \times 10^{-5}$  and  $5 \times 10^{-10} \leq \sigma_e \leq 5 \times 10^{-5}$ , and found that they do not signifi-

Table 2: Benchmarking a simple geometry optimizer that uses  $\text{GP}_E$  (not shown because it fails to converge),  $\text{GP}_F$ , and  $\text{GP}_{EF}$  surrogates against L-BFGS geometry optimizer in MOPAC to test effect of fitting forces. The column id gives the Baker-Chan system id,  $N_{\text{atom}}$  gives the number of atoms, # SPE gives the number of SPE calculations,  $\Delta E$  gives the final energy difference between the optimized structure with respect to MOPAC’s in eV, and RMSD gives the root-mean-squared distances in Å of the final geometry with respect to MOPAC’s.

id	$N_{\text{atom}}$	# SPE			$\Delta E$ (eV)		RMSD (Å)	
		L-BFGS	$\text{GP}_F$	$\text{GP}_{EF}$	$\text{GP}_F$	$\text{GP}_{EF}$	$\text{GP}_F$	$\text{GP}_{EF}$
1	3	18	16	15	0.00	0.00	0.00	0.00
2	4	77	25	22	-0.38	-0.38	0.07	0.07
3	4	22	36	33	-0.01	-0.01	0.18	0.17
4	5	11	10	10	-0.00	-0.00	0.00	0.00
5	8	22	33	25	0.00	0.00	0.01	0.01
6	10	22	29	26	0.01	-0.00	0.06	0.02
7	10	36	50	46	0.00	0.00	0.04	0.04
8	10	24	33	28	0.01	0.01	0.06	0.07
9	16	26	40	36	0.02	0.00	0.14	0.10
10	8	13	10	10	0.00	0.00	0.00	0.00
11	10	11	8	10	0.00	-0.00	0.00	0.00
12	8	15	15	17	0.00	0.00	0.01	0.01
13	8	20	11	11	-0.00	0.00	0.00	0.01
14	7	18	18	18	-0.00	-0.00	0.04	0.04
15	4	31	12	11	-0.78	-0.77	0.70	0.70
16	7	25	34	26	0.00	0.00	0.05	0.05
17	14	50	36	35	0.11	0.10	0.39	0.37
18	11	17	22	23	-0.00	-0.00	0.03	0.03
19	5	34	28	24	0.01	0.01	0.02	0.02
20	7	23	37	36	-0.17	-0.17	0.12	0.11
21	8	18	12	11	0.00	0.00	0.02	0.02
22	7	19	24	22	0.00	-0.00	0.02	0.01
23	5	26	20	20	0.00	0.00	0.00	0.00
24	5	22	21	21	0.00	-0.00	0.00	0.00
25	5	17	52	49	0.01	0.01	0.14	0.14

cantly affect performance. We refer the reader to the Supporting Information for more details (Table S1).

Table 2 summarizes the results of benchmarking GP surrogates against each other and L-BFGS in MOPAC. We include  $\Delta E$  (eV), which measure the energy difference between the final geometries obtained by each method with respect to the one obtained by MOPAC, as well as the root-mean-squared distance (RMSD in Å). The RMSD calculation between the mad-GP optimized and L-BFGS baseline structures were calculated after the alignment (translate the center-of-mass and perform the proper ro-

tation) of the two molecules using an implementation of the Quaternion algorithm<sup>80</sup> from the rmsd package.<sup>81</sup> We refer the reader to the Supporting Information for more details about the RMSD calculation (Figure S1).

Recall again that we are not using an adaptive length-scale. In particular, a Matérn kernel with a length-scale of 0.4 will be quite large once we near the convergence threshold of  $\|\mathbf{F}_{N+1}\| \leq 0.0054 \cdot 3N_A$ , especially given that we do not dampen the size of the optimizer steps once we near convergence.

Our results show that GPs that fit forces perform better compared to those that do not on

the Baker-Chan dataset. In particular, none of the geometry optimizations using  $\text{GP}_E$  converge. The geometry optimizations that use  $\text{GP}_F$  and  $\text{GP}_{EF}$  converge for all of the molecules. Both  $\text{GP}_F$  and  $\text{GP}_{EF}$  are competitive with the baseline L-BFGS optimizer in MOPAC with respect to the number of SPE calculations, consistent with previous work demonstrating the effectiveness of GP surrogates. Occasionally,  $\text{GP}_F$  and  $\text{GP}_{EF}$  find geometries that are lower-energy than the L-BFGS optimizer in MOPAC. All final geometries are stable local minima as verified by frequency calculations conducted with MOPAC except for system id 3, which has a relatively flat PES and has a small imaginary frequency ( $56.7i \text{ cm}^{-1}$  in  $\text{GP}_F$ ,  $5.5i \text{ cm}^{-1}$  in  $\text{GP}_{EF}$ , and  $56.5i \text{ cm}^{-1}$  in L-BFGS baseline) under current convergence criteria. Note that some optimized structures in the Baker-Chan set are pre/post-reaction complex (*e.g.*, 03, 09, 15, 17) with multiple local minima, leading to higher RMSD values and slightly larger energy differences between GP and L-BFGS. This also indicates that the advantage of GP is finding lower-energy structures compared to the L-BFGS algorithm. Next, we will take a closer look at  $\text{GP}_F$  and  $\text{GP}_{EF}$ .

### 3.1.2 Fitting forces versus energies and forces

It is popular to fit both energies and forces when using GPs as a surrogate function for a PES for geometry optimization. This is a natural design decision to make because knowing both energies and forces gives more information than just forces. Nevertheless, some work<sup>3,16</sup> considers fitting forces exclusively. To the best of our knowledge, there has been relatively little work comparing the two approaches, and we aim to fill that gap now.

At first glance, the results in Table 2 support the intuition that it is better to fit both energies and forces.  $\text{GP}_{EF}$  and  $\text{GP}_F$  require roughly the same number of SPE calculations across the Baker-Chan dataset. Nevertheless, there is an important optimization detail that is different between  $\text{GP}_{EF}$  and  $\text{GP}_F$  which we term the “set maximum” step.

State-of-the-art results on applying  $\text{GP}_{EF}$  surrogates use constant mean functions and a variation of the following step: update the mean function to be

$$\mu(\mathbf{x}) = \max_{1 \leq i \leq N+1} E_i + c \quad (40)$$

which sets the mean function to be the constant function that returns the highest energy seen during geometry optimization and  $c$  is some constant. The justification for this design decision is that it makes the optimizer more stable by forcing a local minima (*e.g.*, Denzel et al.<sup>12</sup>). We find this reasoning plausible: setting the prior to a high value in unseen regions should encourage the optimizer to stay around regions that it has seen before because regions far away from observed points will take on the mean value (which is high). The reason that the “set maximum” step is important is because it is not applicable to  $\text{GP}_F$ . As a reminder, the bias term only affects the energy component when the mean function is constant (the derivative of a constant is 0), and a force-only GP does not have an energy component.

Table 3 summarizes the results of comparing  $\text{GP}_F$  and  $\text{GP}_{EF}$  by varying the prior choices, including the ability to use the “set maximum” step. The entries with a  $\Delta E$  but no corresponding SPE exceed the max cycle limitation before reaching convergence (> 300 steps). The entries with no  $\Delta E$  and no corresponding SPE failed to produce any meaningful results. As we can see, the “set maximum” step where  $c = 0$  drastically improves the performance of  $\text{GP}_{EF}$  by reducing the number of SPE calls required. We have also tested the “set maximum” step with  $c = 10$  (*e.g.*, as suggested by Denzel et al.<sup>12</sup>) and found that we improve the results over a baseline where we do not perform a “set maximum” step, but worse when  $c = 0$ . (Note that Denzel et al.’s geometry optimization<sup>12</sup> is not the same as ours and contains an over-shooting step that may explain the difference in performance, among other differences.) Thus, at least for constant 0 priors, we can improve the performance of  $\text{GP}_{EF}$  at the cost of tuning one additional parameter (*i.e.*, the constant  $c$ ) and performing a “set maximum” step during optimi-

Table 3: Comparing  $\text{GP}_F$  and  $\text{GP}_{\text{EF}}$  by varying prior choices to test impact of fitting energies. The kernel  $k$  corresponds to a Matérn kernel with  $\sigma_M = 1.0$  and  $l = 0.4$ . The prior  $\tilde{\mu}$  corresponds to setting the prior to the maximum energy found during training,  $\tilde{\mu}_{10}$  corresponds to setting the prior to the maximum energy plus 10 during training, and LJ corresponds to a Leonard-Jones potential. The entries with a  $\Delta E$  (in eV) but no corresponding SPE timed-out before reaching convergence ( $> 300$  steps).

id	$\text{GP}_F(0, k)$		$\text{GP}_{\text{EF}}(0, k)$		$\text{GP}_{\text{EF}}(\tilde{\mu}, k)$		$\text{GP}_{\text{EF}}(\tilde{\mu}_{10}, k)$		$\text{GP}_F(\text{LJ}, k)$		$\text{GP}_{\text{EF}}(\text{LJ}, k)$	
	#SPE	$\Delta E$	#SPE	$\Delta E$	#SPE	$\Delta E$	#SPE	$\Delta E$	#SPE	$\Delta E$	#SPE	$\Delta E$
1	16	0.00	146	0.00	15	0.00	26	0.00	18	0.00	147	0.00
2	25	-0.38	186	-0.38	22	-0.38	36	-0.38	33	1.69	185	-0.38
3	36	-0.01	-	0.00	33	-0.01	52	-0.01	-	-	-	0.00
4	10	-0.00	151	0.00	10	-0.00	19	0.00	49	0.00	150	0.00
5	33	0.00	-	0.03	25	0.00	53	0.01	-	-	-	0.03
6	29	0.01	-	0.06	26	-0.00	50	0.02	-	-	-	0.06
7	50	0.00	-	-	46	0.00	83	0.01	-	-	-	0.48
8	33	0.01	-	0.05	28	0.01	36	0.01	-	-	-	0.05
9	40	0.02	-	-	36	0.00	51	0.06	-	-	-	-
10	10	0.00	207	0.00	10	0.00	11	0.00	11	0.00	189	0.00
11	8	0.00	95	0.00	10	-0.00	12	0.00	40	0.00	94	0.00
12	15	0.00	178	0.00	17	0.00	33	0.00	67	-0.01	176	0.00
13	11	-0.00	220	0.00	11	0.00	21	0.00	40	0.00	222	0.00
14	18	-0.00	266	0.00	18	-0.00	32	0.00	-	-	264	0.00
15	12	-0.78	-	-	11	-0.77	15	-0.77	11	-0.77	-	-
16	34	0.00	-	0.68	26	0.00	57	0.00	-	-	-	0.67
17	36	0.11	-	-	35	0.10	41	0.13	-	-	-	-
18	22	-0.00	-	-	23	-0.00	52	0.00	-	-	-	-
19	28	0.01	-	1.93	24	0.01	57	0.02	38	0.00	-	1.93
20	37	-0.17	-	0.02	36	-0.17	57	-0.17	36	-0.17	-	0.02
21	12	0.00	167	0.00	11	0.00	11	0.00	-	-	155	0.00
22	24	0.00	-	0.04	22	-0.00	41	0.00	54	-2.16	-	0.04
23	20	0.00	193	0.00	20	0.00	32	0.00	20	0.00	192	0.00
24	21	0.00	203	0.00	21	-0.00	33	0.00	21	0.00	204	0.00
25	52	0.01	-	0.02	49	0.01	70	0.01	42	2.30	-	-

mization.

We have also included an example of a non-constant prior in the form of a simplified pairwise Leonard-Jones (LJ) potential

$$V_{\text{LJ}}(\mathbf{x}) = \sum_{i < j} 4\varepsilon_{\text{LJ}} \left[ \left( \frac{\sigma_{\text{LJ}}}{r_{i,j}} \right)^{12} - \left( \frac{\sigma_{\text{LJ}}}{r_{i,j}} \right)^6 \right] \quad (41)$$

where  $\mathbf{x} = (\mathbf{x}^1 \dots \mathbf{x}^{N_A})^T$  are the xyz coordinates of a molecular structure and  $r_{i,j} = \|\mathbf{x}^i - \mathbf{x}^j\|$  is the distance between atoms  $i$  and  $j$  (in Å). Thus we compute the LJ potential  $V_{\text{LJ}}(\mathbf{x})$  as the summation of the pairwise po-

tential of every unique pair of atoms. We do not use a cutoff radius in evaluating the summation. For our tests, we set the diameter of the cross-section  $\sigma_{\text{LJ}} = 1 \text{ \AA}$  and the depth of the potential well  $\epsilon_{\text{LJ}} = 1 \text{ eV}$ . We implement this LJ potential as a proof-of-concept to demonstrate how to implement a non-constant prior in mad-GP (see Figure S2 in the Supporting Information for more details). We are not claiming that LJ is a good choice of mean function.

The surrogate  $\text{GP}_{\text{EF}}(\text{LJ}, k)$  performs similarly to  $\text{GP}_{\text{EF}}(0, k)$  and  $\text{GP}_F(\text{LJ}, k)$  performs similarly to  $\text{GP}_F(0, k)$ . This is not surprising as the LJ potential is a very simple model that is

mostly suitable for describing non-covalent interactions between non-bonded molecules, and unsuitable for bond breaking, bond stretching, valence-angle bending, and internal rotations. In this proof-of-concept test, the LJ mean function behaves like a constant 0 mean function. Implementing and systematically testing more complex and physically inspired prior mean functions is current work-in-progress. We also believe that further exploration of the “set maximum” step and how it affects a choice of mean function is worth pursuing in future work that more thoroughly compares  $\text{GP}_F$  and  $\text{GP}_{\text{EF}}$ .

### 3.1.3 Gaussian processes with non-constant priors and hierarchical kernels

We use mad-GP to explore several combinations of non-constant priors and hierarchical kernels for  $\text{GP}_F$  and  $\text{GP}_{\text{EF}}$ . Figure 5 provides a visualization the results using wandb.<sup>82</sup>

Each curve, read from left-to-right, passes through each choice and results in an average number of steps (color-coded, darker is better) across the Baker-Chan dataset (lower is better). The choices include (1) optimizer:  $\text{GP}_F$  vs.  $\text{GP}_{\text{EF}}$ , (2) mean functions, (3) descriptors: COULOMB which gives the Coulomb matrix,<sup>14</sup> PRE-COULOMB which gives the Coulomb matrix without atomic charges  $Z_i$ , and xyz which gives Cartesian coordinates, and (4) kernel functions: squared-exponential, Matérn kernel with  $p = 3$  (three-times differentiable), and Matérn kernel with  $p = 2$  (twice differentiable).

For completeness, the Coulomb matrix without atomic charges is defined as the matrix with  $ij$ -th entry

$$P_{ij} = \|\mathbf{x}^i - \mathbf{x}^j\| \quad (42)$$

where  $\mathbf{x} = (\mathbf{x}^1 \dots \mathbf{x}^{N_A})^T$  are the xyz coordinates of a molecular structure with  $N_A$  atoms. The Coulomb matrix is defined as the matrix with  $ij$ -th entry

$$C_{ij} = \begin{cases} 0.5Z_i^{2.4} & i = j \\ \frac{Z_i Z_j}{\|\mathbf{x}^i - \mathbf{x}^j\|} & \text{otherwise} \end{cases} \quad (43)$$

where  $\mathbf{x} = (\mathbf{x}^1 \dots \mathbf{x}^{N_A})^T$  are the xyz coordinates of a molecular structure with  $N_A$  atoms.

The squared-exponential kernel is

$$k_{\text{se}}(\mathbf{x}, \mathbf{y}) = \sigma_{\text{se}}^2 \exp\left(\frac{\|\mathbf{x} - \mathbf{y}\|^2}{l^2}\right) \quad (44)$$

where  $\sigma_{\text{se}}$  is a weight and  $l$  is a length-scale parameter. Finally, the Matérn kernel with  $p = 3$  is

$$k_{\text{M3}}(\mathbf{x}, \mathbf{y}) = \sigma_{\text{M}}^2 \left(1 + \frac{\sqrt{7}\rho}{l} + \frac{14\rho^2}{5l^2} + \frac{7\sqrt{7}\rho^3}{15l^3}\right) \exp\left(\frac{-\sqrt{7}\rho}{l}\right) \quad (45)$$

where  $\rho = \|\mathbf{x} - \mathbf{y}\|$ .

For the first choice of optimizer, we see that  $\text{GP}_F$  is better at reducing the average number of steps as compared to  $\text{GP}_{\text{EF}}$ . Note that we are not performing the “set maximum” step for  $\text{GP}_{\text{EF}}$  as we hope to compare different priors. These results are consistent with those in Table 3. For prior functions, constant prior functions perform better than LJ potentials. These results are also consistent with those in Table 3.

For the choice of descriptors, xyz coordinates perform better than both the Coulomb descriptor and the pre-Coulomb descriptor without atomic charges. There are at least two points we should highlight about this result. First, AD enables us to perform gradient-based geometry optimization using descriptors of molecules without requiring us to define an inverse transformation from descriptor space back into xyz coordinates. Indeed, the geometry optimization with Coulomb descriptors converges on the Baker-Chan dataset, and thus gives us an example of a chemically-inspired descriptor that can be applied for geometry optimization. Second, while the average number of steps may be higher for a single geometry optimization, note that Coulomb descriptors are invariant to rotation, a property that xyz descriptors do not possess. Consequently, further geometry optimization involving a GP using Coulomb descriptors will be robust to rotations of a molecule while those based on xyz descriptors will not be. Note that, herein, we are referring to the *direct* usage of the xyz encoding of a structure *without* prefixing the orientation (by defining a standard

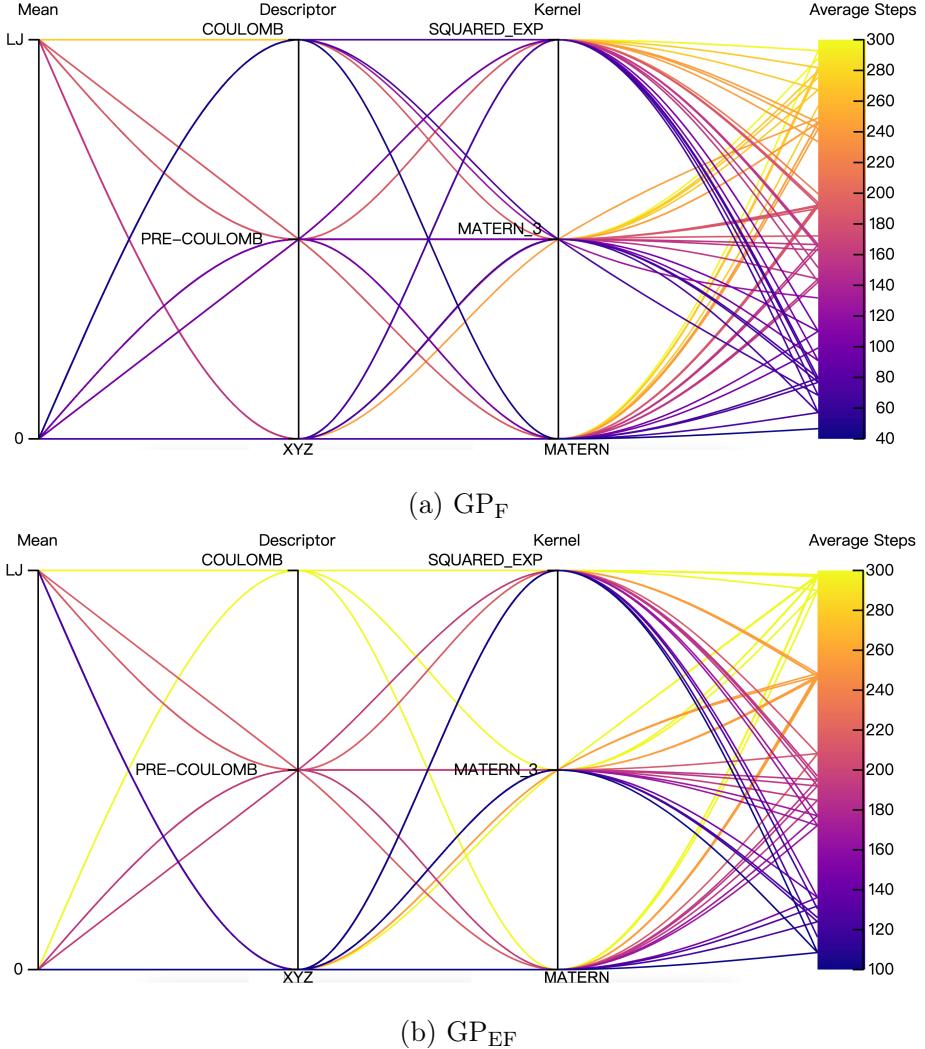


Figure 5: Comparing different choices of mean functions, descriptors, and kernel functions with respect to the average number of steps across the Baker-Chan dataset (lower is better). Each curve, read from left-to-right, passes through each choice and results in an average number of steps (darker is better). The figure was generated with wandb.<sup>82</sup>

orientation using principle axis as done in quantum chemistry codes) and the center of mass of the input structure, which certainly eliminate the issue of rotation and translation invariance of the descriptor.

Finally, for the choice of kernel functions, we see that if we use constant 0 mean function, then Matérn kernels perform the best. As a reminder, we are not performing the “set maximum” step here for constant priors so it is better to look at the results for GP<sub>F</sub> for comparison of mean and kernel functions. For GP<sub>F</sub>, we see that several combinations work well: Coulomb matrices with Matérn  $p = 2, p = 3$ , and squared exponential kernels work. Coulomb descriptors

increase the dimensionality of the representation, and so it is interesting to see that kernels that impose smoother constraints still work.

### 3.2 Using Automatic Differentiation

One of the design goals of mad-GP is to enable users to more systematically explore different priors and kernel functions for GP PES surrogate models without needing to implement first and second-order derivatives. The key technique for achieving this was AD. We summarize some of our experiences using AD in this

section.

One question we might be interested in is: "what can I do now that I could not do before?" For a standard kernel such as a Matérn kernel, the extra work that is done if AD is not used is (1) deriving or looking up the derivatives, (2) transcribing them to code, (3) and checking that the derivatives are implemented correctly. For completeness, the first-order derivatives are

$$\frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x}} = -\frac{5}{3l^3} [(l + \sqrt{5}\rho)(\mathbf{x} - \mathbf{y})] \sigma_M^2 \exp\left(\frac{-\sqrt{5}\rho}{l}\right) \quad (46)$$

and

$$\frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{y}} = -\frac{\partial k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{y}} \quad (47)$$

The second-order derivative is

$$\begin{aligned} \frac{\partial^2 k(\mathbf{x}, \mathbf{y})}{\partial \mathbf{x} \partial \mathbf{y}} = & -\frac{5}{3l^3} \left[ \frac{5}{l} ((\mathbf{x} - \mathbf{y}) \otimes (\mathbf{x} - \mathbf{y}) - \right. \\ & \left. (l + \sqrt{5}\rho) \mathbf{I} \right] \sigma_M^2 \exp\left(\frac{-\sqrt{5}\rho}{l}\right) \end{aligned} \quad (48)$$

where  $\otimes$  is an outer product.

Although this might be simple enough for a standard kernel, consider implementing the Matérn kernel for arbitrary  $p$ . This kernel is defined as

$$\begin{aligned} k_p(\mathbf{x}, \mathbf{y}) = & \sigma_M^2 \exp\left(\frac{-\sqrt{2p+1}\rho}{l}\right) \frac{p!}{(2p)!} \\ & \sum_{i=0}^p \frac{(p+i)!}{i!(p-i)!} \left(\frac{2\sqrt{2p+1}\rho}{l}\right)^{p-i} \end{aligned} \quad (49)$$

where  $\rho = \|\mathbf{x} - \mathbf{y}\|$ . We have implemented this kernel for  $1 \leq p \leq 10$  in mad-GP using 18 lines-of-code. Deriving the first and second-order derivatives with respect to  $\mathbf{x}$  and  $\mathbf{y}$  and checking that they are correct is harder. Notably, SD tools are less effective on multivariate spaces.

Beyond this, consider deriving and implementing first and second-order derivatives for a chemical descriptor that may not be differentiable in the traditional sense. Figure 6 gives an example of a descriptor implemented in mad-GP that is not differentiable because the columns of the Coulomb matrix are sorted

by their  $l_2$  norm. (It also gives an example of how to use atomic charges in a GP kernel.) AD, unlike SD, can still handle such descriptors by using the concept of sub-gradients. Thus AD presents us the opportunity to perform geometry optimization using kernels with descriptors that are non-invertible and non-differentiable.

We emphasize again that AD implements an appropriate factorization of the SD in an automatic fashion. Consequently, it implements a particular symbolic derivative and the floating-point precision is identical to implementing that symbolic derivative. The performance cost that one pays when using AD is that the first and second-order derivatives are derived when the code is executed. This is a one-time cost because we can use JIT technology to cache the results of the derivatives. Compared to the cost of SPE calculations and geometry optimization, the one-time cost to compile the first and second-order derivatives is negligible.

## 4 Discussion

As we demonstrate with our prototype mad-GP, users need only write the mean and kernel functions before being able to explore a variety of GP surrogate models that implement  $\text{GP}_E$ ,  $\text{GP}_F$ , or  $\text{GP}_{EF}$ . In the rest of this section, we will highlight a few interesting discoveries that we were able to uncover due to the flexibility of mad-GP.

**AD is crucial for atomistic kernel functions** In light of the testing results, one can see the possibility for an abundance of mean functions, kernel functions, their combinations, and their compositions that one can explore using mad-GP. To the best of our knowledge, most prior work explores simple mean functions like constant function and standard kernels such as Matérn kernels. Notably, the usage of AD along with gradient-based optimization of GP surrogates means that we can use arbitrarily complex descriptors so long as we can implement them in mad-GP. We have identified several combinations that work using mad-GP. We leave a more thorough exploration

```

1 def coulomb_sort_by_l2_kernel(k):
2     def sort(A):
3         return A[jnp.argsort(jnp.linalg.norm(A, axis=-1))]
4     def inner(atoms_x, atoms_y, x, y):
5         zs1 = jnp.array(atoms_x.get_atomic_numbers())
6         zs2 = jnp.array(atoms_y.get_atomic_numbers())
7         zs = jax.lax.map(lambda m1: jax.lax.map(lambda m2: m1 * m2, zs1), zs2)
8         return k(atoms_x, atoms_y,
9                 sort(coulomb_desc(zs1, x)).reshape(-1),
10                sort(coulomb_desc(zs2, y)).reshape(-1))
11

```

Figure 6: A kernel where we sort the columns of a Coulomb matrix by it’s  $l_2$  norm (lines 2–3 and 8–9). Line 5 gives an example of how to use the molecule metadata `atoms_x` and `atoms_y` to obtain the atomic masses for use in a descriptor. The function `jax.lax.map` is a functional looping construct (line 6). The function `coulomb_desc` (not shown) implements a Coulomb matrix using atomic masses and positions as inputs.

of this direction as future work.

**“Set maximum” step** The usage of  $\text{GP}_{\text{EF}}$  only performs well in our tests if the “set maximum” step is applied. In contrast,  $\text{GP}_F$  performs well without this additional step. Consequently, we find that  $\text{GP}_F$  is more robust compared to  $\text{GP}_{\text{EF}}$  because we do not need to modify the geometry optimization algorithm. Crucially, the “set maximum” step affects our ability to explore different mean functions for  $\text{GP}_{\text{EF}}$ . We believe that further investigation of this step and its impact on choice of mean function in comparing  $\text{GP}_{\text{EF}}$  and  $\text{GP}_F$  is a good direction of future work. These questions only became apparent to us when we tried to implement a tool that could handle a variety of GP use cases in a generic manner. At a meta-level, we hope that this provides further evidence that better tools can lead to better science.

## 5 Conclusion

In summary, we introduce mad-GP, a library for constructing GP surrogate models of PESs. A user of mad-GP only needs to write down the functional form of the mean and kernel functions, and the library handles all required derivative implementations. mad-GP accomplishes this with a technique called AD. Our

hope is that mad-GP can be used to more systematically explore the large design space of GP surrogate models for PESs.

As an initial case study, we apply mad-GP to perform geometry optimization. We compare GP surrogates that fit forces with those that fit energies and forces. In general, we find that  $\text{GP}_{\text{EF}}$  performs comparably with  $\text{GP}_F$  in terms of the number of SPE calculations required, although  $\text{GP}_F$  is more robust for optimization because it does not require an additional step to be applied during optimization. In our preliminary study on the use of non-constant priors and hierarchical kernels in GP PES surrogates, we also confirm that constant mean functions and Matérn kernels work well as reported in the literature, although our tests also identify several other promising candidates (*e.g.*, Coulomb matrices with three-times differentiable Matérn kernels). Our studies validate that AD is a viable method for performing geometry optimization with GP surrogate models on small molecules.

**Acknowledgement** This work was supported by Oracle Labs and by the Schiller Institute Grant for Exploratory Collaborative Scholarship (SIGECS). The authors thank Darius Russell Kish, Weiming Qin and Yang Wang for feedback on usage of mad-GP. Finally, the authors thank the Boston College Linux Cluster

for computing resources.

## References

- (1) Bartók, A. P.; Payne, M. C.; Kondor, R.; Csányi, G. Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons. *Phys. Rev. Lett.* **2010**, *104*, 136403.
- (2) Rowe, P.; Deringer, V. L.; Gasparotto, P.; Csányi, G.; Michaelides, A. An accurate and transferable machine learning potential for carbon. *The Journal of Chemical Physics* **2020**, *153*, 034702.
- (3) Chmiela, S.; Tkatchenko, A.; Sauceda, H. E.; Poltavsky, I.; Schütt, K. T.; Müller, K.-R. Machine learning of accurate energy-conserving molecular force fields. *Science Advances* **2017**, *3*.
- (4) Chmiela, S.; Sauceda, H. E.; Müller, K.-R.; Tkatchenko, A. Towards exact molecular dynamics simulations with machine-learned force fields. *Nature communications* **2018**, *9*, 1–10.
- (5) Sugisawa, H.; Ida, T.; Krems, R. V. Gaussian Process Model of 51-Dimensional Potential Energy Surface for Protonated Imidazole Dimer. *The Journal of Chemical Physics* **2020**, *153*, 114101.
- (6) Denzel, A.; Haasdonk, B.; Kästner, J. Gaussian Process Regression for Minimum Energy Path Optimization and Transition State Search. *The Journal of Physical Chemistry A* **2019**, *123*, 9600–9611, PMID: 31617719.
- (7) Denzel, A.; Kästner, J. Gaussian Process Regression for Transition State Search. *Journal of Chemical Theory and Computation* **2018**, *14*, 5777–5786, PMID: 30351931.
- (8) Koistinen, O.-P.; Ásgeirsson, V.; Vehtari, A.; Jónsson, H. Nudged Elastic Band Calculations Accelerated with Gaussian Process Regression Based on Inverse Interatomic Distances. *Journal of Chemical Theory and Computation* **2019**, *15*, 6738–6751, PMID: 31638795.
- (9) Koistinen, O.-P.; Ásgeirsson, V.; Vehtari, A.; Jónsson, H. Minimum Mode Saddle Point Searches Using Gaussian Process Regression with Inverse-Distance Covariance Function. *Journal of Chemical Theory and Computation* **2020**, *16*, 499–509, PMID: 31801018.
- (10) Unke, O. T.; Chmiela, S.; Sauceda, H. E.; Gastegger, M.; Poltavsky, I.; Schütt, K. T.; Tkatchenko, A.; Müller, K.-R. Machine Learning Force Fields. *Chemical Reviews* **0**, *0*, null, PMID: 33705118.
- (11) Sugisawa, H.; Ida, T.; Krems, R. V. Gaussian process model of 51-dimensional potential energy surface for protonated imidazole dimer. *The Journal of Chemical Physics* **2020**, *153*, 114101.
- (12) Denzel, A.; Kästner, J. Gaussian process regression for geometry optimization. *The Journal of Chemical Physics* **2018**, *148*, 094114.
- (13) Garijo del Río, E.; Mortensen, J. J.; Jacobsen, K. W. Local Bayesian optimizer for atomic structures. *Phys. Rev. B* **2019**, *100*, 104103.
- (14) Rupp, M.; Tkatchenko, A.; Müller, K.-R.; von Lilienfeld, O. A. Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning. *Phys. Rev. Lett.* **2012**, *108*, 058301.
- (15) Bartók, A. P.; Kondor, R.; Csányi, G. On representing chemical environments. *Phys. Rev. B* **2013**, *87*, 184115.
- (16) Sauceda, H. E.; Chmiela, S.; Poltavsky, I.; Müller, K.; Tkatchenko, A. Molecular force fields with gradient-domain machine learning: Construction and application to dynamics of small molecules with coupled

- cluster forces. *The Journal of Chemical Physics* **2019**, *150* 11, 114102.
- (17) Smith Jr, V. H.; Schaefer III, H. F.; Morokuma, K. *Applied Quantum Chemistry: Proceedings of the Nobel Laureate Symposium on Applied Quantum Chemistry in Honor of G. Herzberg, RS Mulliken, K. Fukui, W. Lipscomb, and R. Hoffman, Honolulu, HI, 16–21 December 1984*; Springer Science & Business Media, 2012.
- (18) Fukui, K. Formulation of the Reaction Coordinate. *The Journal of Physical Chemistry* **1970**, *74*, 4161–4163.
- (19) Fukui, K.; Kato, S.; Fujimoto, H. Constituent Analysis of the Potential Gradient along a Reaction Coordinate. Method and an Application to Methane + Tritium Reaction. *Journal of the American Chemical Society* **1975**, *97*, 1–7.
- (20) Ishida, K.; Morokuma, K.; Komornicki, A. The Intrinsic Reaction Coordinate. An Ab Initio Calculation for  $\text{HNC} \rightarrow \text{HCN}$  and  $\text{H} + \text{CH}_4 \rightarrow \text{CH}_4 + \text{H}$ . *The Journal of Chemical Physics* **1977**, *66*, 2153–2156.
- (21) Blais, N. C.; Truhlar, D. G.; Garrett, B. C. Improved Parametrization of Diatomics-in-molecules Potential Energy Surface for  $\text{Na}(3\text{p } 2\text{P}) + \text{H}_2 \rightarrow \text{Na}(3\text{s } 2\text{S}) + \text{H}_2$ . *The Journal of Chemical Physics* **1983**, *78*, 2956–2961.
- (22) Truhlar, D. G.; Steckler, R.; Gordon, M. S. Potential Energy Surfaces for Polyatomic Reaction Dynamics. *Chemical Reviews* **1987**, *87*, 217–236.
- (23) Varandas, A. J. C.; Brown, F. B.; Mead, C. A.; Truhlar, D. G.; Blais, N. C. A Double Many-body Expansion of the Two Lowest-energy Potential Surfaces and Nonadiabatic Coupling for  $\text{H}_3$ . *The Journal of Chemical Physics* **1987**, *86*, 6258–6269.
- (24) Tucker, S. C.; Truhlar, D. G. A Six-Body Potential Energy Surface for the  $\text{SN}_2$  Reaction  $\text{Cl}(\text{g}) + \text{CH}_3\text{Cl}(\text{g})$  and a Variational Transition-State-Theory Calculation of the Rate Constant. *Journal of the American Chemical Society* **1990**, *112*, 3338–3347.
- (25) Lynch, G. C.; Steckler, R.; Schwenke, D. W.; Varandas, A. J. C.; Truhlar, D. G.; Garrett, B. C. Use of Scaled External Correlation, a Double Many-body Expansion, and Variational Transition State Theory to Calibrate a Potential Energy Surface for  $\text{FH}_2$ . *The Journal of Chemical Physics* **1991**, *94*, 7136–7149.
- (26) Dahlke, E. E.; Truhlar, D. G. Electrostatically Embedded Many-Body Expansion for Simulations. *Journal of Chemical Theory and Computation* **2008**, *4*, 1–6.
- (27) Mezey, P. G. Reactive domains of energy hypersurfaces and the stability of minimum energy reaction paths. *Theoretica chimica acta* **1980**, *54*, 95–111.
- (28) Mezey, P. G. Catchment region partitioning of energy hypersurfaces, I. *Theoretica chimica acta* **1981**, *58*, 309–330.
- (29) Mezey, P. G. The isoelectronic and isoprotic energy hypersurface and the topology of the nuclear charge space. *International Journal of Quantum Chemistry* **1981**, *20*, 279–285.
- (30) Mezey, P. G. Manifold theory of multidimensional potential surfaces. *International Journal of Quantum Chemistry* **1981**, *20*, 185–196.
- (31) Mezey, P. G. Critical level topology of energy hypersurfaces. *Theoretica chimica acta* **1981**, *60*, 97–110.
- (32) Mezey, P. G. Lower and upper bounds for the number of critical points on energy hypersurfaces. *Chemical Physics Letters* **1981**, *82*, 100–104.

- (33) Mezey, P. G. Quantum chemical reaction networks, reaction graphs and the structure of potential energy hypersurfaces. *Theoretica chimica acta* **1982**, *60*, 409–428.
- (34) Mezey, P. G. Topology of energy hypersurfaces. *Theoretica chimica acta* **1982**, *62*, 133–161.
- (35) Mezey, P. G. The topology of energy hypersurfaces II. Reaction topology in euclidean spaces. *Theoretica chimica acta* **1983**, *63*, 9–33.
- (36) Mezey, P. Potential Energy Hypersurfaces, Studies in physical and theoretical chemistry. 1987.
- (37) Duchovic, R.; Volobuev, Y.; Lynch, G.; Truhlar, D.; Allison, T.; Wagner, A.; Garrett, B.; Corchado, J. POTLIB 2001: A Potential Energy Surface Library for Chemical Systems. *Computer Physics Communications* **2002**, *144*, 169–187.
- (38) Alış, Ö. F.; Rabitz, H. Efficient implementation of high dimensional model representations. *Journal of Mathematical Chemistry* **2001**, *29*, 127–142.
- (39) Yagi, K.; Oyanagi, C.; Taketsugu, T.; Hirao, K. Ab initio potential energy surface for vibrational state calculations of H<sub>2</sub>CO. *The Journal of chemical physics* **2003**, *118*, 1653–1660.
- (40) Yagi, K.; Hirata, S.; Hirao, K. Multiresolution potential energy surfaces for vibrational state calculations. *Theoretical Chemistry Accounts* **2007**, *118*, 681–691.
- (41) Carter, S.; Culik, S. J.; Bowman, J. M. Vibrational self-consistent field method for many-mode systems: A new approach and application to the vibrations of CO adsorbed on Cu (100). *The Journal of chemical physics* **1997**, *107*, 10458–10469.
- (42) Bowman, J. M.; Carter, S.; Huang, X. MULTIMODE: a code to calculate rovibrational energies of polyatomic molecules. *International Reviews in Physical Chemistry* **2003**, *22*, 533–549.
- (43) Bowman, J. M.; Carrington, T.; Meyer, H.-D. Variational quantum approaches for computing vibrational energies of polyatomic molecules. *Molecular Physics* **2008**, *106*, 2145–2182.
- (44) Braams, B. J.; Bowman, J. M. Permutationally invariant potential energy surfaces in high dimensionality. *International Reviews in Physical Chemistry* **2009**, *28*, 577–606.
- (45) Jäckle, A.; Meyer, H.-D. Product representation of potential energy surfaces. II. *The Journal of chemical physics* **1998**, *109*, 3772–3779.
- (46) Otto, F. Multi-layer Potfit: An accurate potential representation for efficient high-dimensional quantum dynamics. *The Journal of chemical physics* **2014**, *140*, 014106.
- (47) Avila, G.; Carrington Jr, T. Using multi-dimensional Smolyak interpolation to make a sum-of-products potential. *The Journal of chemical physics* **2015**, *143*, 044106.
- (48) Ziegler, B.; Rauhut, G. Efficient generation of sum-of-products representations of high-dimensional potential energy surfaces based on multimode expansions. *The Journal of chemical physics* **2016**, *144*, 114114.
- (49) Truhlar, D. G.; Horowitz, C. J. Functional Representation of Liu and Siegbahn's Accurate Ab Initio Potential Energy Calculations for H+H<sub>2</sub>. *The Journal of Chemical Physics* **1978**, *68*, 2466–2476.
- (50) Thompson, T. C.; Izmirlian, G.; Lemon, S. J.; Truhlar, D. G.; Mead, C. A. Consistent Analytic Representation of the Two Lowest Potential Energy Surfaces for Li<sub>3</sub>, Na<sub>3</sub>, and K<sub>3</sub>. *The Journal of Chemical Physics* **1985**, *82*, 5597–5603.

- (51) Nguyen, K. A.; Rossi, I.; Truhlar, D. G. A Dual-level Shepard Interpolation Method for Generating Potential Energy Surfaces for Dynamics Calculations. *The Journal of Chemical Physics* **1995**, *103*, 5522–5530.
- (52) Manzhos, S.; Carrington Jr, T. A random-sampling high dimensional model representation neural network for building potential energy surfaces. *The Journal of chemical physics* **2006**, *125*, 084109.
- (53) Schütt, K. T.; Kindermans, P.-J.; Sauceda, H. E.; Chmiela, S.; Tkatchenko, A.; Müller, K.-R. SchNet: A Continuous-Filter Convolutional Neural Network for Modeling Quantum Interactions. Proceedings of the 31st International Conference on Neural Information Processing Systems. Red Hook, NY, USA, 2017; p 992–1002.
- (54) Schütt, K. T.; Sauceda, H. E.; Kindermans, P.-J.; Tkatchenko, A.; Müller, K.-R. Schnet—a deep learning architecture for molecules and materials. *The Journal of Chemical Physics* **2018**, *148*, 241722.
- (55) Schutt, K.; Kessel, P.; Gastegger, M.; Nicoli, K.; Tkatchenko, A.; Müller, K.-R. SchNetPack: A deep learning toolbox for atomistic systems. *Journal of chemical theory and computation* **2018**, *15*, 448–455.
- (56) Behler, J.; Parrinello, M. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett.* **2007**, *98*, 146401.
- (57) Behler, J. Representing potential energy surfaces by high-dimensional neural network potentials. *Journal of Physics: Condensed Matter* **2014**, *26*, 183001.
- (58) Behler, J. Perspective: Machine learning potentials for atomistic simulations. *The Journal of chemical physics* **2016**, *145*, 170901.
- (59) Zhang, L.; Han, J.; Wang, H.; Saidi, W.; Car, R.; E, W. End-to-end Symmetry Preserving Inter-atomic Potential Energy Model for Finite and Extended Systems. Advances in Neural Information Processing Systems. 2018.
- (60) Smith, J. S.; Isayev, O.; Roitberg, A. E. ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost. *Chemical science* **2017**, *8*, 3192–3203.
- (61) Unke, O. T.; Meuwly, M. PhysNet: a neural network for predicting energies, forces, dipole moments, and partial charges. *Journal of chemical theory and computation* **2019**, *15*, 3678–3693.
- (62) Anderson, B.; Hy, T. S.; Kondor, R. Cormorant: Covariant Molecular Neural Networks. *Advances in Neural Information Processing Systems* **2019**, *32*, 14537–14546.
- (63) Klicpera, J.; Groß, J.; Günnemann, S. Directional Message Passing for Molecular Graphs. International Conference on Learning Representations. 2019.
- (64) Wood, M. A.; Thompson, A. P. Extending the accuracy of the SNAP interatomic potential form. *The Journal of chemical physics* **2018**, *148*, 241721.
- (65) Williams, C. K.; Rasmussen, C. E. *Gaussian Processes for Machine Learning*; MIT press Cambridge, MA, 2006; Vol. 2.
- (66) De, S.; Bartók, A. P.; Csányi, G.; Cerotti, M. Comparing molecules and solids across structural and alchemical space. *Phys. Chem. Chem. Phys.* **2016**, *18*, 13754–13769.
- (67) Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Millman, K. J.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C. J.; Polat, İ.; Feng, Y.;

- Moore, E. W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; SciPy 1.0 Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **2020**, *17*, 261–272.
- (68) Lawson, C. L.; Hanson, R. J. *Solving least squares problems*; SIAM, 1995.
- (69) Meyer, R.; Hauser, A. W. Geometry optimization using Gaussian process regression in internal coordinate systems. *The Journal of Chemical Physics* **2020**, *152*, 084112.
- (70) Himanen, L.; Jäger, M. O. J.; Morooka, E. V.; Federici Canova, F.; Ranawat, Y. S.; Gao, D. Z.; Rinke, P.; Foster, A. S. Dscribe: Library of descriptors for machine learning in materials science. *Computer Physics Communications* **2020**, *247*, 106949.
- (71) Kuhn, H. W. The Hungarian method for the assignment problem. *Naval research logistics quarterly* **1955**, *2*, 83–97.
- (72) Baydin, A. G.; Pearlmutter, B. A.; Radul, A. A.; Siskind, J. M. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research* **2018**, *18*, 1–43.
- (73) Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; Chintala, S. In *Advances in Neural Information Processing Systems 32*; Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R., Eds.; Curran Associates, Inc., 2019; pp 8024–8035.
- (74) Bradbury, J.; Frostig, R.; Hawkins, P.; Johnson, M. J.; Leary, C.; Maclaurin, D.; Necula, G.; Paszke, A.; VanderPlas, J.; Wanderman-Milne, S.; Zhang, Q. JAX: composable transformations of Python+NumPy programs. 2018; <http://github.com/google/jax>.
- (75) Baker, J.; Chan, F. The location of transition states: A comparison of Cartesian, Z-matrix, and natural internal coordinates. *Journal of Computational Chemistry* **1996**, *17*.
- (76) Larsen, A. H.; Mortensen, J. J.; Blomqvist, J.; Castelli, I. E.; Christensen, R.; Dułak, M.; Friis, J.; Groves, M. N.; Hammer, B.; Haragus, C.; Hermes, E. D.; Jennings, P. C.; Jensen, P. B.; Kermode, J.; Kitchin, J. R.; Kolsbjerg, E. L.; Kubal, J.; Kaasbørg, K.; Lysgaard, S.; Maronsson, J. B.; Maxson, T.; Olsen, T.; Pastewka, L.; Peterson, A.; Rostgaard, C.; Schiøtz, J.; Schütt, O.; Strange, M.; Thygesen, K. S.; Vegge, T.; Vilhelmsen, L.; Walter, M.; Zeng, Z.; Jacobsen, K. W. The atomic simulation environment—a Python library for working with atoms. *Journal of Physics: Condensed Matter* **2017**, *29*, 273002.
- (77) Stewart, J. J. Optimization of parameters for semiempirical methods VI: more modifications to the NDDO approximations and re-optimization of parameters. *Journal of molecular modeling* **2013**, *19*, 1–32.
- (78) Stewart, J. J. P. MOPAC2016. *Stewart Computational Chemistry, Colorado Springs, CO, USA* **2016**,
- (79) Fdez. Galván, I.; Raggi, G.; Lindh, R. Restricted-Variance Constrained, Reaction Path, and Transition State Molecular Optimizations Using Gradient-Enhanced Kriging. *Journal of Chemical Theory and Computation* **2021**, *17*, 571–582, PMID: 33382621.

- (80) Walker, M. W.; Shao, L.; Volz, R. A. Estimating 3-D location parameters using dual number quaternions. *CVGIP: Image Understanding* **1991**, *54*, 358–367.
- (81) Kromann, J. C. Calculate Root-mean-square deviation (RMSD) of Two Molecules Using Rotation. 2021; Software available from <http://github.com/charnley/rmsd,v1.4>.
- (82) Biewald, L. Experiment Tracking with Weights and Biases. 2020; <https://www.wandb.com/>, Software available from wandb.com.

## Supporting Information Available

### List of Listings in Supporting Information

S1	Initial geometries in Baker-Chan dataset . . . . .	S1
----	----------------------------------------------------	----

### List of Figures in Supporting Information

S1	The script we use to calculate the RMSD between geometries obtained from MOPAC with those obtained from GP surrogates in Table 2. We use the rmsd package from <a href="http://github.com/charnley/rmsd">http://github.com/charnley/rmsd</a> . . . . .	5
S2	A prototype implementation of the simplified Leonard-Jones (LJ) potential used in Table 3. We consider all pairwise interactions and we do not use a cutoff radius. . . . .	6

### List of Tables in Supporting Information

S1	Comparing different noise levels for $\text{GP}_{\text{EF}}$ and $\text{GP}_{\text{F}}$ . We use an energy noise of $\sigma_e = 2 \times 10^{-n}$ and a force noise $\sigma_f = 5 \times 10^{-n}$ for varying $n$ . . . . .	5
----	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---

Listing S1: Initial geometries in Baker-Chan dataset

```

==> Baker_25/01.xyz <==
3
HCN <--> HNC
    C      0.00000      0.00000      0.00000
    N      1.14838      0.00000      0.00000
    H      1.14838      1.58536      0.00000

==> Baker_25/02.xyz <==
5
HCCH <--> CCH2
    C      0.00000      0.00000      0.00000
    C      1.24054      0.00000      0.00000

```

X 0.00000 1.00000 0.00000 H 0.17678 -0.96865 2.00849  
 H 0.81952 -1.44008 -0.00000 H -0.92403 0.52585 0.00000  
 ==> Baker\_25/08.xyz <==  
 10  
 8. BETA-(FORMYLOXY) ETHYL  
 C 0.00000 0.00000 0.00000  
 C 1.48700 0.00000 0.00000  
 O 1.94238 1.38506 0.00000  
 C 0.90421 2.25068 0.19004  
 O -0.15162 1.69459 0.62349  
 H -0.52584 -0.52311 0.80552  
 H -0.51623 0.06744 -0.96446  
 H 1.92286 -0.47194 0.91622  
 H 1.93776 -0.43941 -0.92636  
 H 1.04450 3.26680 -0.19030  
 ==> Baker\_25/09.xyz <==  
 20  
 9. PARENT DIELS-ALDER  
 X 0.00000 0.00000 0.00000  
 X 1.00000 0.00000 0.00000  
 X 0.00000 1.20000 0.00000  
 X -1.73205 -1.00000 -0.00000  
 C 0.00000 0.00000 1.40000  
 C 0.00000 0.00000 -1.40000  
 C 0.00000 1.20000 -0.77000  
 C 0.00000 1.20000 0.77000  
 C -1.73205 -1.00000 0.70000  
 C -1.73205 -1.00000 -0.70000  
 H 0.08152 2.11122 -1.34396  
 H 0.08152 2.11122 1.34396  
 H 0.08311 -0.00735 2.47677  
 H 0.08311 -0.00735 -2.47677  
 H -0.09930 -0.89587 0.80324  
 H -0.09930 -0.89587 -0.80324  
 H -2.27850 -0.21959 1.20656  
 H -2.27850 -0.21959 -1.20656  
 H -1.34601 -1.86707 1.21533  
 H -1.34601 -1.86707 -1.21533  
 ==> Baker\_25/10.xyz <==  
 8  
 10. S-TETRAZINE <--> 2HCN + N2  
 N 0.00000 0.00000 0.00000  
 N 1.20000 0.00000 0.00000  
 C -0.75000 1.29904 0.00000  
 C 1.95000 1.29904 0.00000  
 N -0.15000 2.33827 0.00000  
 N 1.35000 2.33827 0.00000  
 H -1.83000 1.29904 0.00000  
 H 3.03000 1.29904 -0.00000  
 ==> Baker\_25/11.xyz <==  
 10  
 11. ROTATIONAL TS IN BUTADIENE  
 C 0.00000 0.00000 0.00000  
 C 1.46700 0.00000 0.00000  
 C -0.73813 1.09433 0.00000  
 C 2.20513 -1.02833 0.37428  
 ==> Baker\_25/03.xyz <==  
 4  
 H2CO <--> H2 + CO  
 C 0.00000 0.00000 0.00000  
 O 1.25000 0.00000 0.00000  
 H -0.22574 1.28025 0.00000  
 H -1.29904 0.75000 0.00000  
 ==> Baker\_25/04.xyz <==  
 5  
 CH3O <--> CH2OH  
 O 0.00000 0.00000 0.00000  
 C 1.42300 0.00000 0.00000  
 H 0.33239 1.00639 0.00000  
 H 1.92492 -0.27223 0.92495  
 H 1.92492 -0.27223 -0.92495  
 ==> Baker\_25/05.xyz <==  
 8  
 5. RING OPENING CYCLOPROPYL  
 C 0.00000 0.00000 0.00000  
 C 1.45400 0.00000 0.00000  
 C 1.20152 1.43191 0.00000  
 H -0.54630 0.73668 -0.61814  
 H -0.54630 -0.90367 0.32891  
 H 1.39931 2.02151 0.91459  
 H 1.39931 2.02151 -0.91459  
 H 2.15071 -0.79836 0.09654  
 ==> Baker\_25/06.xyz <==  
 10  
 6. BICYCLO[1.1.0] BUTANE TS 1  
 C 0.00000 0.00000 0.00000  
 C 1.49500 0.00000 0.00000  
 C 1.54696 1.41705 0.00000  
 C 0.68458 -0.65428 1.11519  
 H -0.78624 0.75702 -0.05825  
 H 2.10820 -0.60536 -0.70132  
 H 1.84711 1.99477 -0.88416  
 H 1.47876 2.04447 0.89727  
 H 0.73503 -0.29612 2.16460  
 H 0.68720 -1.75990 1.14386  
 ==> Baker\_25/07.xyz <==  
 10  
 7. BICYCLO[1.1.0] BUTANE TS 2  
 C 0.00000 0.00000 0.00000  
 C 1.48900 0.00000 0.00000  
 C 2.48236 0.94927 0.00000  
 C 0.36986 -0.10696 1.32937  
 H -0.53209 -0.84588 -0.47858  
 H 1.85885 -0.98195 -0.38878  
 H 2.38170 1.96390 0.40743  
 H 3.52822 0.70563 -0.22407  
 H 0.64874 0.73642 1.98829

H -0.55624 -0.92574 0.00000  
 H 2.02324 0.86991 -0.31662  
 H -1.81502 1.00011 -0.00000  
 H -0.28086 2.07495 0.00000  
 H 3.28202 -0.93980 0.34206  
 H 1.74786 -1.94982 0.70968

==> Baker\_25/12.xyz <==  
 8

**12. CH<sub>3</sub>CH<sub>3</sub> <--> CH<sub>2</sub>CH<sub>2</sub> + H<sub>2</sub>**  
 C 0.00000 0.00000 0.00000  
 C 1.43000 0.00000 0.00000  
 H -0.75000 1.29904 0.00000  
 H 0.68000 1.29904 0.00000  
 H -0.46065 -0.23899 -0.95853  
 H -0.46065 -0.23899 0.95853  
 H 1.89065 -0.23899 0.95853  
 H 1.89065 -0.23899 -0.95853

==> Baker\_25/13.xyz <==  
 8

**13. CH<sub>3</sub>CH<sub>2</sub>F <--> CH<sub>2</sub>CH<sub>2</sub> + HF**  
 C 0.00000 0.00000 0.00000  
 C 1.43000 0.00000 0.00000  
 H -0.26047 1.47721 0.00000  
 F 1.75993 1.87113 0.00000  
 H -0.45643 -0.25334 -0.94546  
 H -0.45643 -0.25334 0.94546  
 H 1.88643 -0.25334 0.94546  
 H 1.88643 -0.25334 -0.94546

==> Baker\_25/14.xyz <==  
 7

**14. CH<sub>2</sub>CHOH <--> CH<sub>3</sub>CHO**  
 C 0.00000 0.00000 0.00000  
 C 1.43000 0.00000 0.00000  
 O 2.08000 1.12583 0.00000  
 H -0.37280 -0.43287 -0.92830  
 H -0.37280 -0.51213 0.88704  
 H 1.97500 -0.94038 0.08227  
 H 0.73385 1.12361 0.10190

==> Baker\_25/15.xyz <==  
 4

**15. HCOCl <--> HCl + CO**  
 O 0.00000 0.00000 0.00000  
 C 1.17000 0.00000 0.00000  
 Cl 1.17000 2.33500 0.00000  
 H 2.29700 0.00000 -0.00000

==> Baker\_25/16.xyz <==  
 7

**16. H<sub>2</sub>O + PO<sub>3</sub><sup>-</sup> <--> H<sub>2</sub>PO<sub>4</sub><sup>-</sup>**  
 P 0.00000 0.00000 0.00000  
 O 1.46000 0.00000 0.00000  
 O -0.48664 1.37651 0.00000  
 O -0.65996 -0.93339 -1.61668  
 O -0.88375 -1.24996 0.16789  
 H 0.07857 -1.24213 -2.12829

==> Baker\_25/17.xyz <==  
 14

**17. CLAISEN REARRANGEMENT**  
 C 0.00000 0.00000 0.00000  
 C 1.42000 0.00000 0.00000  
 C 2.12250 1.21677 0.00000  
 O 1.64355 1.93968 -1.81110  
 C 0.33402 1.96918 -1.79207  
 C -0.39798 0.76439 -1.76446  
 H -0.46065 0.85553 0.49394  
 H -0.46065 -0.98788 -0.00000  
 H 1.96500 -0.94397 0.00000  
 H 3.20835 1.12177 0.00000  
 H 1.61192 2.04347 0.49394  
 H -1.48132 0.88382 -1.75004  
 H 0.08664 -0.08038 -2.25394  
 H -0.18972 2.92509 -1.79759

==> Baker\_25/18.xyz <==  
 11

**18. SLYENE INSERTION**  
 C 0.00000 0.00000 0.00000  
 C 1.52000 0.00000 0.00000  
 Si 2.64418 2.00654 0.00000  
 H -0.45039 0.42206 0.91050  
 H -0.46169 0.45944 -0.87399  
 H -0.35460 -1.05165 -0.01983  
 H 1.85900 -0.54759 0.87939  
 H 1.87685 -0.55983 -0.87707  
 H 2.18743 2.81279 -1.15402  
 H 1.73047 1.22083 0.89319  
 H 4.09462 1.74959 -0.14350

==> Baker\_25/19.xyz <==  
 6

**19. HNCCS <--> HNC + CS**  
 H 0.00000 0.00000 0.00000  
 N 1.01000 0.00000 0.00000  
 X 1.01000 1.00000 0.00000  
 C 2.00485 0.67103 -0.00000  
 C 2.15078 2.52530 -0.00000  
 S 3.50493 3.31977 -0.00000

==> Baker\_25/20.xyz <==  
 7

**20. HCONH<sub>3</sub><sup>+</sup> <--> NH<sub>4</sub><sup>+</sup> + CO**  
 N 0.00000 0.00000 0.00000  
 C 2.00000 0.00000 0.00000  
 O 2.56500 0.97861 0.00000  
 H -0.34382 0.97092 0.00000  
 H 1.46000 -0.93531 0.00000  
 H -0.34382 -0.48546 -0.84084  
 H -0.34382 -0.48546 0.84084

==> Baker\_25/21.xyz <==  
 8

**21. ROTATIONAL TS IN ACROLEIN**

C	0.00000	0.00000	0.00000
C	1.34000	0.00000	0.00000
C	2.06500	1.25574	0.00000
O	2.37000	1.78401	1.05655
H	-0.54000	-0.93531	-0.00000
H	-0.54000	0.93531	0.00000
H	1.88000	-0.93531	0.00000
H	2.33500	1.72339	-0.93531

==> Baker\_25/22.xyz <==

7

22. HCONHOH <--> HCONHNHO

O	0.00000	0.00000	0.00000
C	1.30000	0.00000	0.00000
N	1.74463	1.22160	0.00000
O	3.12336	1.46471	0.00000
H	2.00707	-0.84265	0.00000
H	3.29013	2.45070	0.00000
H	0.25033	1.35233	0.00000

==> Baker\_25/23.xyz <==

5

23. HNC + H2 <--> H2CNH

H	0.00000	0.00000	0.00000
N	1.00000	0.00000	0.00000
C	1.60000	1.03923	0.00000
H	2.45945	1.54303	0.08682
H	3.06820	0.50927	0.05933

==> Baker\_25/24.xyz <==

5

24. H2CNH <--> HCNH2

H	0.00000	0.00000	0.00000
N	1.00000	0.00000	0.00000
C	1.60000	1.03923	0.00000
H	1.76250	0.19486	-0.97500
H	1.10000	1.90526	0.00000

==> Baker\_25/25.xyz <==

5

25. HCNH2 <--> HCN + H2

C	0.00000	0.00000	0.00000
N	1.35000	0.00000	0.00000
H	-0.25882	0.96593	0.00000
H	1.71235	-1.17112	0.67615
H	1.82883	-1.23623	-0.44995

Table S1: Comparing different noise levels for  $\text{GP}_{\text{EF}}$  and  $\text{GP}_{\text{F}}$ . We use an energy noise of  $\sigma_e = 2 \times 10^{-n}$  and a force noise  $\sigma_f = 5 \times 10^{-n}$  for varying  $n$ .

id	steps								
	$n = 2$		$n = 3$		$n = 4$		$n = 5$		
	$\text{GP}_{\text{EF}}$	$\text{GP}_{\text{F}}$	$\text{GP}_{\text{EF}}$	$\text{GP}_{\text{F}}$	$\text{GP}_{\text{EF}}$	$\text{GP}_{\text{F}}$	$\text{GP}_{\text{EF}}$	$\text{GP}_{\text{F}}$	
1	15	16	15	16	15	16	15	16	
2	22	25	22	25	21	25	21	25	
3	33	37	33	36	32	36	37	36	
4	10	10	10	10	10	10	10	10	
5	25	30	25	33	25	33	25	33	
6	27	29	26	29	26	29	26	29	
7	47	49	46	50	45	50	45	50	
8	32	30	28	33	26	33	26	33	
9	35	36	36	40	73	40	59	40	
10	10	10	10	10	10	10	10	10	
11	11	8	10	8	10	8	10	8	
12	18	15	17	15	17	15	17	15	
13	12	11	11	11	11	11	11	11	
14	18	18	18	18	18	18	18	18	
15	12	13	11	12	11	12	11	12	
16	25	33	26	34	26	34	26	34	
17	36	37	35	36	45	36	47	36	
18	24	22	23	22	23	22	23	22	
19	24	29	24	28	23	28	23	28	
20	38	39	36	37	36	37	36	37	
21	13	14	11	12	11	12	11	12	
22	22	24	22	24	22	24	22	24	
23	20	20	20	20	20	20	20	20	
24	21	22	21	21	21	21	21	21	
25	49	51	49	52	47	52	51	52	
average	24.9	25.1	23.4	25.2	24.9	25.2	24.8	25.2	

```

1 def get_rmsd(mol1: np.ndarray,
2                 mol2: np.ndarray) -> float:
3     mol1 -= rmsd.centroid(mol1)
4     mol2 -= rmsd.centroid(mol2)
5     return rmsd.quaternion_rmsd(mol1, mol2)

```

Figure S1: The script we use to calculate the RMSD between geometries obtained from MOPAC with those obtained from GP surrogates in Table 2. We use the rmsd package from <http://github.com/charnley/rmsd>.

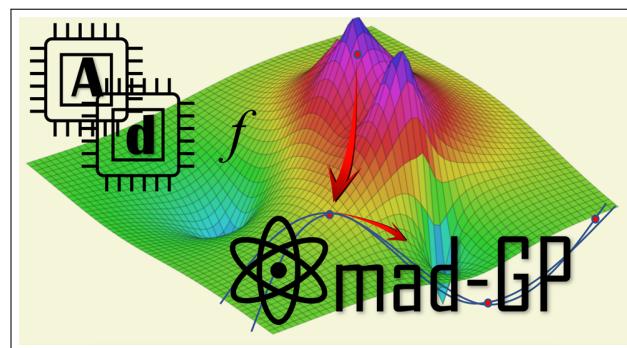
```

1 def leonard_jones(atoms_x, x):
2     x = jnp.reshape(x, (-1, 3))
3     n = int(x.size / 3)
4     dist_exp_2 = ((x[:, None, :] - x) ** 2).sum(-1)
5     dist_exp_2 = dist_exp_2[jnp.triu_indices(n, 1)]
6     dist_exp_m6 = dist_exp_2 ** -3
7     dist_exp_m12 = dist_exp_m6 ** 2
8     E = (dist_exp_m12 - 2 * dist_exp_m6).sum()
9     return E

```

Figure S2: A prototype implementation of the simplified Leonard-Jones (LJ) potential used in Table 3. We consider all pairwise interactions and we do not use a cutoff radius.

# TOC Graphic



# Geometry Meta-Optimization

Daniel Huang,<sup>1</sup> Junwei Lucas Bao,<sup>2</sup> and Jean-Baptiste Tristan<sup>3</sup>

<sup>1)</sup>*Department of Computer Science, San Francisco State University, San Francisco, California 94132, United States*

<sup>2)</sup>*Department of Chemistry, Boston College, Chestnut Hill, Massachusetts 02467, United States*

<sup>3)</sup>*Department of Computer Science, Boston College, Chestnut Hill, Massachusetts 02467, United States*

(\*Electronic mail: tristanj@bc.edu)

(\*Electronic mail: lucas.bao@bc.edu)

(\*Electronic mail: danehuang@sfsu.edu)

(Dated: March 6, 2022)

Recent work has demonstrated the promise of using machine-learned surrogates, in particular Gaussian process (GP) surrogates, in reducing the number of electronic structure calculations (ESCs) needed to perform surrogate model based (SMB) geometry optimization. In this paper, we study *geometry meta-optimization* with GP surrogates where a SMB optimizer additionally learns from its past “experience” performing geometry optimization. To validate this idea, we start with the simplest setting where a geometry meta-optimizer learns from previous optimizations of the *same* molecule with different initial-guess geometries.

We give empirical evidence that geometry meta-optimization with GP surrogates is effective and requires less tuning compared to SMB optimization with GP surrogates on the ANI-1 dataset of off-equilibrium initial structures of small organic molecules. Unlike SMB optimization where a surrogate should be immediately useful for optimizing a given geometry, a surrogate in geometry meta-optimization has more flexibility because it can distribute its ESC savings across a set of geometries. Indeed, we find that GP surrogates that preserve rotational invariance provide increased marginal ESC savings across geometries. As a more stringent test, we also apply geometry meta-optimization to conformational search on a hand-constructed dataset of hydrocarbons and alcohols. We observe that while SMB optimization and geometry meta-optimization do save on ESCs, they also tend to miss higher energy conformers compared to standard geometry optimization. We believe that further research into characterizing the divergence between GP surrogates and PESs is critical not only for advancing geometry meta-optimization, but also for exploring the potential of machine-learned surrogates in geometry optimization in general.

## I. INTRODUCTION

Geometry optimization is a fundamental task in theoretical chemistry. Importantly, it is used to discover stationary points on a potential energy surface (PES) of a collection of atoms (*e.g.*, a molecule). A PES is a  $3N - 6$  hypersurface for an  $N$  atom system and gives the system’s energy as a function of its geometry (*i.e.*, each atom’s nuclear coordinates). The stationary points of a PES correspond to compounds and substances relevant to chemistry (*e.g.*, reactants, products, transition-state structures), and thus, information about a PES of an atomistic system can be used to study its chemical properties (*e.g.*, thermodynamics, reaction mechanisms, reaction rates).

An ab initio geometry optimizer employs electronic structure calculations (ESCs) to solve the electronic Schrödinger equation and obtain information useful for exploring a PES (*e.g.*, energy and/or forces at a given geometry). However, ESCs can be computationally intensive—*e.g.*, self-consistent field (SCF) methods such as Kohn-Sham density functional theory (KS-DFT) have  $O(K^3)$  to  $O(K^4)$  scaling where  $K$  is the number of atomic basis functions<sup>1,2</sup>. Consequently, it is critical to design optimizers that minimize the number of ESCs if we hope to scale geometry optimization to larger systems.

One promising approach for reducing the cost of geometry optimization is to use surrogate model based (SMB) geometry optimization. In SMB (geometry) optimization, information

obtained from a surrogate of a PES (which is designed to be cheap to evaluate) is used to explore that PES as opposed to information obtained from the PES directly (which may involve an expensive ESC). Recent work<sup>3–21</sup> demonstrates that machine learning (ML) models that are trained on energies and/or forces obtained from ESCs for a given geometry are effective PES surrogates for SMB optimization. Many promising results use kernel ridge regression, or Gaussian processes (GPs) more generally, due to their ability to constrain the model to conserve energy (*i.e.*, the prediction of the force is the negative gradient of the prediction of the energy), although different ML models can be used as surrogates as well.

In typical SMB optimization, the surrogate model (or surrogate models) constructed during optimization is discarded after it is completed. This raises the following natural question: can we leverage the artifacts of previous optimizations for accelerating future geometry optimization? We might imagine the following series of increasingly difficult tasks.

1. Leverage past optimizations to improve optimization of the *same* atomistic system from similar initial geometries. This would give us another method for accelerating a single geometry optimization like SMB optimization.
2. Leverage past optimizations to improve optimization involving *varying numbers* of fixed types of elements. This would enable us to scale geometry optimization to

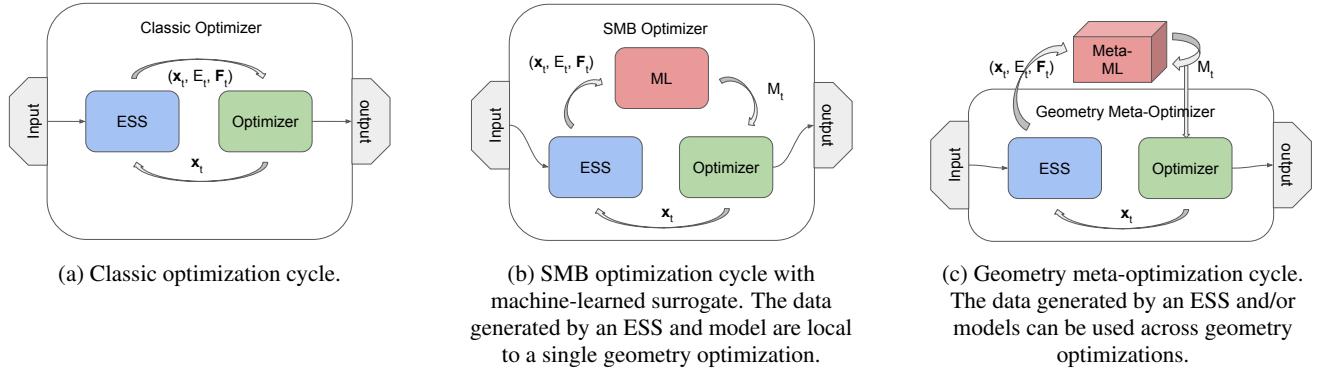


Figure 1: A comparison of the classic optimization, SMB optimization with machine-learned surrogate, and geometry meta-optimization.

larger systems involving the same elements.

3. Leverage past optimizations to improve optimization of atomistic systems involving elements that are *chemically similar* (*e.g.*, elements in the same column in the periodic table). This would enable us to improve optimization of an atomistic system by transferring knowledge from an atomistic system that is similar.

The approach to ML where an algorithm learns from its “experience” with a task is called meta-learning<sup>22–24</sup>, and shares some important concepts with reinforcement learning<sup>25</sup>, Bayesian optimization<sup>26,27</sup>, and active learning<sup>28</sup>. In the context of geometry optimization, we might thus be interested in exploring the concept of a *geometry meta-optimizer* that learns from its past “experience” performing geometry optimizations. While the idea of geometry meta-optimization is straightforward in principle, it is also a bold one to tackle given the complexity of PESs. Our goal in this paper is to take the first step and assess whether geometry meta-optimization based on SMB optimization for the same atomistic system can be competitive with classic and state-of-the-art SMB geometry optimization with machine-learned surrogates.

Towards this end, we introduce a baseline geometry meta-optimizer for SMB optimization that use GP surrogates, describe its properties including the importance of molecular representations (our computer experiments involve molecules), and introduce an extension of the baseline geometry meta-optimizer to manage computational complexity (Section II). We empirically characterize the behavior of these geometry meta-optimizers with two tests. First, we perform a series of experiments on the ANI-1 dataset<sup>29,30</sup>, which contains many off-equilibrium initial-guess geometries of small organic molecules generated by normal-mode displacement, to validate that learning across geometry optimizations of very similar initial geometries is possible for the same molecule. Second, we perform a more stringent test by using geometry meta-optimization to perform conformational search on a hand-constructed dataset of small hydrocarbons and alcohols where initial geometries are generated by rotating salient dihedral bonds (*i.e.*, internal torsions that exclude the methyl groups since their rotations do not lead to new unique con-

formers) to investigate how learning occurs across a more diverse set of initial geometries (Section III).

We leave with at least two important takeaways from our exploration of geometry meta-optimization. First, we observe that geometry meta-optimizers are not only competitive in terms of reducing the number of ESCs, but they also do not require clever exploration of the surrogate model (Section IV). Thus geometry meta-optimization can be used as a robust method for accelerating any single geometry optimization similar to SMB optimization. Second, surrogates play different roles in SMB optimization compared to geometry meta-optimization: whereas a surrogate in SMB optimization is designed to reduce the number of ESCs for a single geometry optimization, a surrogate in geometry meta-optimization can distribute the reduction of ESCs across a set of geometry optimizations. Indeed, one reason we might hope that geometry meta-optimization is possible at all is to use this flexibility to generalize features of a PES that are helpful for future optimization. For example, we find that rotational invariance is beneficial across optimizations, but not necessary for a single optimization (Section IV).

Perhaps surprisingly, in our experiment with using geometry meta-optimization to search for conformers, we find that optimization with GP surrogates (*i.e.*, with either geometry meta-optimizers or SMB optimization), although they do save on ESCs, tend to miss higher energy conformers compared to a traditional geometry optimizer (Section IV). We believe that further research into characterizing the divergence between GP surrogates and PESs is an interesting direction for future work, and critical not only for advancing geometry meta-optimization, but also for exploring the potential of machine-learned surrogates in geometry optimization in general.

## II. METHODS

Before we introduce a geometry meta-optimization (Section II B), we first introduce background on SMB optimization as a reference point for geometry meta-optimization (Section II A).

### A. Surrogate Model Based Optimization

The geometry optimization algorithms that are implemented in quantum chemistry software can be thought of as an interaction between (1) an electronic structure solver (ESS) and (2) an optimizer (Figure 1a). The ESS performs an ESC by solving the electronic Schrödinger equation under a fixed-nuclei approximation (*e.g.*, with a SCF method), and can provide information of the energy and forces at a given geometry. Popular optimizers in chemistry applications include direct inversion in the iterative subspace for geometry optimization<sup>31</sup> (GDIIS), energy-represented DIIS<sup>32</sup> (GEDIIS), and eigenfollowing optimization<sup>33</sup> (EF). The ESS and optimizer work together in lockstep: (1) the optimizer chooses a geometry  $\mathbf{x}_t$  and (2) the ESS computes the energy  $E_t$  and/or forces  $\mathbf{F}_t$  that can now be used by the optimizer to propose the next geometry  $\mathbf{x}_{t+1}$ .

Recent work<sup>3–16,19</sup> has shown that it is possible to improve on classic optimization by using SMB optimization with machine learned surrogate models (Figure 1b). In this approach, the optimizer is augmented with a machine learned surrogate model  $\mathcal{M}_t$  that aids it in choosing a geometry  $\mathbf{x}_t$ . The surrogate model is typically trained on all previous geometries  $(\mathbf{x}_1, \dots, \mathbf{x}_{t-1})$  and their energies  $(E_1, \dots, E_{t-1})$  and forces  $(\mathbf{F}_1, \dots, \mathbf{F}_{t-1})$  encountered during optimization. That is, we perform supervised learning on the dataset  $\mathcal{D} = (\mathbf{x}_t, (E_t, \mathbf{F}_t))_{t \in 1\dots T}$  of input/output pairs where the geometries  $\mathbf{x}_t$  form the inputs, the energies  $E_t$  and forces  $\mathbf{F}_T$  form the outputs, and  $T$  is the length of the current optimization. Note that it is possible to only train on energies or only on forces as well<sup>16</sup>. A trained surrogate leads to a model of the PES  $\mathcal{M}_t$  that can predict an energy and forces given an input geometry, and can be used by the optimizer to choose the next geometry to explore.

An ideal machine-learned surrogate model  $\mathcal{M}_t$  is (1) simple to train, ideally in an on-the-fly fashion (*i.e.*, do not need to retrain from scratch on all data each iteration), (2) cheap to evaluate, and (3) easy to search for stationary points. Many ML models have been explored as candidates for SMB optimization, each with their own sets of trade-offs, including neural networks<sup>30,34–44</sup> and kernel methods/GPs<sup>10–12,15–17</sup>. Some ML models are designed specifically for modeling PESs (*e.g.*, Gaussian Approximation Potentials<sup>11,12</sup> and Behler-Parrinello Neural Networks<sup>36,37</sup>). In our work, we restrict attention to GP surrogates that are trained on both energy and forces, which are used in many state-of-the-art work on SMB optimization.

A GP is a non-parametric regression model, notated  $\text{GP}(\mu, k)$ , with two tunable ‘‘parameters’’: (1) a *covariance* function  $k$  and (2) a *mean* function  $\mu$ . The covariance function (also called kernel function) enables us to enforce constraints on the model: continuity, differentiability, symmetries (*e.g.*, rotational invariance), and capture conservation of energy by constraining the gradient of the prediction of the energy to be the negative of the prediction of the forces acting on the nuclei. The mean function (also called prior function) enables us to express beliefs about the functional form of a PES. In the context of geometry meta-optimization, the mean function can

be used to integrate surrogate models obtained from previous geometry optimization.

Training a GP model  $\text{GP}(\mu, k)$  on a dataset  $\mathcal{D} = (\mathbf{x}_t, (E_t, \mathbf{F}_t))_{t \in 1\dots T}$  (*i.e.*, performing GP regression), notated  $\text{TRAIN}(\text{GP}(\mu, k), \mathcal{D})$ , gives a PES surrogate  $f_{\mathcal{D}, \mu, k}^*(\mathbf{y})$ , or simply  $f_{\mathcal{D}}^*(\mathbf{y})$  when the mean  $\mu$  and covariance  $k$  can be inferred from context. It has a closed form solution<sup>3</sup>

$$f_{\mathcal{D}}^*(\mathbf{y}) = \sum_{t=1}^T \begin{pmatrix} k(\mathbf{y}, \mathbf{x}_t) & \frac{\partial k}{\partial \mathbf{x}_t}(\mathbf{y}, \mathbf{x}_t) \\ \frac{\partial k}{\partial \mathbf{y}}(\mathbf{y}, \mathbf{x}_t) & \frac{\partial^2 k}{\partial \mathbf{y} \partial \mathbf{x}_t}(\mathbf{y}, \mathbf{x}_t) \end{pmatrix} \begin{pmatrix} \alpha_t \\ \beta_t \end{pmatrix} + \begin{pmatrix} \mu(\mathbf{y}) \\ \frac{\partial}{\partial \mathbf{y}} \mu(\mathbf{y}) \end{pmatrix} \quad (1)$$

where  $\alpha_t$  and  $\beta_t$  are model weights obtained during training by solving a system of linear equations involving the dataset  $\mathcal{D}$ . A GP surrogate (1) has a training complexity of  $O(T^3 D^3)$  (*i.e.*, the complexity of solving a system of  $T(D+1)$  linear equations where each point generates  $D+1$  equations due to also fitting forces) where  $T$  is the number of examples in the dataset and  $D$  is the dimension of  $\mathbf{x}_t$  (*e.g.*,  $D = 3N$  for Cartesian coordinates), (2) can be evaluated efficiently with matrix multiplication (see Equation 1), and (3) is differentiable when  $\mu$  and  $k$  are differentiable (see Equation 1) so it is easy to search for stationary points via gradient-based methods.

One important property of GPs to keep in mind for geometry meta-optimization with GP surrogates is that a trained GP model  $f_{\mathcal{D}}^*(\mathbf{y})$  is itself a GP model with the following relationship:

$$\begin{aligned} \text{TRAIN}(\text{GP}(\mu, k), \{(\mathbf{x}_t, (E_t, \mathbf{F}_t))\} \cup \mathcal{D}) = \\ \text{TRAIN}(f_{\mathcal{D}, \mu, k}^*, \{(\mathbf{x}_t, (E_t, \mathbf{F}_t))\}) \end{aligned} \quad (2)$$

For more background on GPs and GP regression, we refer the reader to standard references<sup>19,45</sup>.

### B. Geometry Meta-Optimization

The high-level idea of geometry meta-optimization is that the optimizer learns across *several* geometry optimizations in addition to learning within a *single* geometry optimization as in traditional SMB optimization. Figure 1c depicts this graphically by lifting the (meta) ML algorithm out of the geometry optimizer and showing that it can train on data and models across various geometry optimizations. Although the design space of a geometry meta-optimizer is large, what we conceptually need to accomplish is to share ‘‘information’’ between geometry optimizations.

For SMB optimization with machine-learned surrogates, there are at least two mediums at our disposal for sharing information: (1) via an ‘‘amalgamated dataset’’ consisting of all previous data generated by an ESS on previous geometry optimizations or (2) via the surrogate models obtained from previous geometry optimizations which implicitly encode the data.

The former approach enables us to formulate the problem of meta-optimization in terms of how to use the ‘‘amalgamated dataset’’ to train a surrogate model and also inform the choice of new examples for future SMB geometry optimization. More concretely for GP surrogates, which examples do

we train on, and can we combine this information with the existing GP to inform the selection of the next example to query? Thus we have fine-grained control over the training set and can adopt a traditional SMB optimization framework at the cost of increasing the difficulty of training on a larger dataset. Techniques such as active learning<sup>28</sup> where a learner can choose which new examples for the current geometry optimization given the existing “amalgamated dataset” to train on can be an interesting direction to pursue to scale geometry meta-optimization based on sharing data. These strategies have already been applied with success for structure search with machine-learned surrogates<sup>46–50</sup>.

The latter approach enables us to formulate the problem of meta-optimization in terms of how to adapt models learned from previous optimizations to be useful for future optimization. More concretely for GP surrogates, can we adapt a GP surrogate fit from previous optimizations for a future optimization or use it as a prior? Thus we constrain the family of models that can be used as surrogates to those that can incorporate past surrogates in an on-the-fly fashion, but can scale better, in principle, because the surrogate model implicitly encodes features of the previous data it is trained on. We believe that both mediums are worth exploring, although for the case of SMB optimization based on GP surrogates that we consider in this paper, we find it more advantageous to adopt the latter approach because the surrogate is trained at each step of optimization in SMB optimization. Consequently, we need to balance the time it takes to train a surrogate compared to the cost of an ESC, and sharing models can take advantage of on-the-fly training.

The simple idea of geometry meta-optimization is that we reuse the surrogate model obtained from a previous geometry optimization as the initial surrogate model for a new geometry optimization as opposed to throwing it away and starting from scratch. This geometry meta-optimizer relies on the fact that a trained GP surrogate is itself a GP (see Equation 2).

The computational complexity of training a GP is a pragmatic reason to take the perspective of using models as the medium of geometry meta-optimization as opposed to the data directly. As a reminder, the complexity of training a GP fit to energies and forces is in  $O(T^3 D^3)$ . Consequently, learning from an “amalgamated” dataset would be impractical for large  $T$ . However, an existing GP surrogate can be incrementally updated with an additional data point in time-complexity  $O(T^2 D^3)$  with a procedure such as incremental Cholesky decomposition (which is implemented in our code as the linear-equation solver), which improves upon the previous situation, but is still practically challenging for large  $T$ . This incremental computation is also used within a single geometry optimization to more efficiently train a GP surrogate in an on-the-fly fashion.

Note that SMB optimization based on GP surrogates forms a special setting where sharing data versus sharing models give mathematically equivalent GP surrogates under the assumption that the mean and covariance functions are held constant. More precisely, it is equivalent to train a GP surrogate on an “amalgamated” dataset or incrementally fit a resulting GP surrogate to new data. Thus there is a sense in which

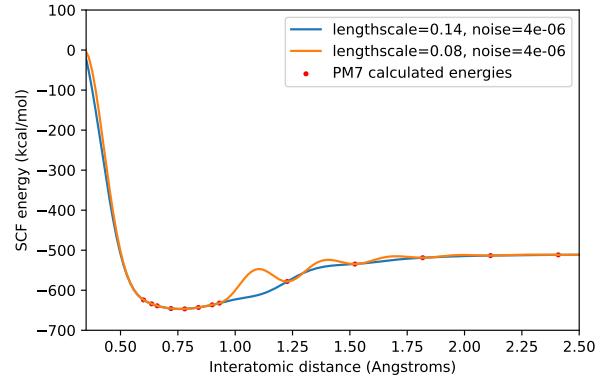


Figure 2: The effect of lengthscale on a GP surrogate fitting the potential-energy curve of the  $\text{H}_2$  molecule as a function of interatomic distance ( $\text{\AA}$ ). A smaller lengthscale (encountered at end of SMB optimization with adaptive lengthscale) adds bumps to regions of the surrogate with larger distances between them (encountered at beginning of SMB optimization with adaptive lengthscale). The noise parameter ensures stability of solving the GPR and is added for completeness.

this geometry meta-optimizer is a baseline. The mathematical equivalence of learning from the data directly versus leveraging the surrogate model is a special property of GP surrogates, and in general, will not hold for other families of ML models.

One key difference between a SMB optimizer and geometry meta-optimizer concerns how the surrogate model is used. SMB optimizers tailor the use of a surrogate model so that it can save the most ESCs for a single geometry optimization. For example, a common strategy in SMB optimization is to reduce the lengthscale of the covariance function as the optimization gets closer to a local minimum. (Note that an adaptive lengthscale for SMB optimization breaks the constant covariance function requirement for equivalence between learning from an “amalgamated” dataset and leveraging past GP surrogates.) This is a sensible heuristic from an optimization perspective because we expect that the distance between geometries to become closer as we approach a local minimum. (Note that the “distance” here is not the physical distance but rather defined as a metric in the kernel function  $k$  to describe the similarity between two geometries during optimization using the Euclidean norm.) At the same time, such a heuristic can also make the resulting surrogate PES nonphysical in regions away from a local minimum (Figure 2). For the purposes of a single optimization, this is not an issue provided that the optimization saves ESCs.

In contrast, because a geometry meta-optimizer shares surrogate models across geometry optimizations, we have the flexibility to distribute the savings of ESCs across a set of geometries. Indeed, one reason we might hope that geometry meta-optimization is possible at all is to use this flexibility to generalize features of a PES that are helpful for future optimization. An example property that is beneficial across

optimizations, but not necessary for a single optimization, is maintaining rotational invariance. We will see evidence of this when we compare the ESC savings of GP surrogates that maintain rotational invariance with those that do not (Section IV).

The idea that we can use the surrogate to generalize features of the PES across geometry optimizations from different initial geometries as opposed to simply being a useful construct for a single geometry optimization also reduces the risk of overfitting to the metric of reducing the number of ESC calculations. In fact, this reveals a problem with some prior work<sup>3</sup> on evaluating SMB optimization. In order to make SMB optimizer competitive, many clever proposals have been made to reduce the number of ESCs (*e.g.*, adaptive length scale and overshooting<sup>3</sup>). But because these are often tested on benchmarks that only contain a single initial geometry of an atomistic system, the distribution of energies generated (*e.g.*, useful for conformational analysis) or robustness across various initial geometries is not assessed. Our experiments with conformational analysis indicate a bias of GP surrogates towards finding lower energy conformers (Section IV).

### C. Representation of Molecules

GP models, like many ML models, operate on vector encodings of molecules instead of molecules directly<sup>37,51</sup>. These vectors encodings may lose invariances that molecules possess such as rotational invariance or translational invariance. Consequently, the model will be blind to these invariances and may use some of its modeling capacity to relearn these invariances. This is more likely to affect geometry meta-optimizers as opposed to SMB optimizers because global invariances should not matter as much at any local point in a single optimization.

A popular representation of molecules in atomistic ML is the Coulomb representation<sup>52</sup>. For a molecule of  $N$  atoms with nuclear coordinates  $X \in \mathbb{R}^{N \times 3}$  and charges  $Z \in \mathbb{R}^{N \times 1}$ , the Coulomb representation is the  $N$  by  $N$  matrix

$$\Phi(X, Z)_{ij} = \begin{cases} 0.5Z_i^{2.4} & \text{for } i = j \\ \frac{Z_i Z_j}{|X_i - X_j|} & \text{otherwise} \end{cases} \quad (3)$$

where off-diagonal elements capture the repulsion between nuclei, and the diagonal is a polynomial fit of atomic energies to nuclear charge. The Coulomb representation is a non-additive, molecule-wise, 2-body representation that is invariant to rotation and translation, but it is not invariant to permutation. Even though it is possible to extend a molecular representation to be invariant to permutation<sup>53–55</sup>, it makes geometry optimization more difficult and we chose not to use them in this work. In an application such as searching for conformers, we can fix a permutation of the atoms as we construct various initial geometries to search from so we can also practically work around this issue. We will see that a similar workaround for rotational invariance where we reorient a molecule with respect to a given molecule does not work as well as enforcing global rotational invariance (Section IV A).

Although a Coulomb representation enforces certain invariances, one issue that arises with using a Coulomb representation for geometry optimization is how to recover Cartesian coordinates from the Coulomb representation. The idea is that we optimize the composition of a PES surrogate with the representation so that we get gradients in Cartesian coordinates instead of optimizing the PES surrogate directly in the space of Coulomb matrices and back-mapping the result to get Cartesian coordinates.

More concretely, Let  $f : \mathcal{C} \rightarrow \mathbb{R}$  be a PES surrogate defined on the space of Coulomb matrices  $\mathcal{C}$  and  $C : \mathbb{R}^{3N} \rightarrow \mathcal{C}$  be a function that maps Cartesian coordinates into Coulomb matrices. Then the approach we take is to optimize  $f \circ C$  (*i.e.*,  $(f \circ C)(x) = f(C(x))$ ) so that the chain rule gives a derivative

$$\frac{\partial f \circ C}{\partial x_i}(x) = \frac{\partial f}{\partial C(x)}(C(x)) \frac{\partial C}{\partial x_i}(x) \quad (4)$$

in Cartesian coordinates so that we can compute gradients in Cartesian coordinates.

Alternatively, we could also optimize the PES surrogate  $f$  directly in the space of Coulomb matrices to find an optimal Coulomb matrix and define a back-mapping  $C^{-1} : \mathcal{C} \rightarrow \mathbb{R}^{3N}$  that takes a Coulomb matrix back into Cartesian coordinates. We choose the former approach because (1) we can use the technique of automatic differentiation to handle the chain-rule seamlessly for us (see mad-GP<sup>56</sup>) and (2) we do not have to construct a back-mapping.

### D. Scaling Geometry Meta-Optimization with Priors

Managing computational complexity is a significant challenge for geometry meta-optimization. More concretely, geometry meta-optimization introduces several computational overheads compared to traditional optimization (*e.g.*, EF). First, the meta-optimizer requires an additional  $O(T^2 D^3)$  calculation for incremental Cholesky at each step of optimization. (Technically, our implementation is  $O(T^2 D^{2.8})$  because we use a block-form of incremental Cholesky that is dominated by  $O(T^2)$  matrix multiplications via Strassen's algorithm.) This comes with an additional  $O(T^2 D^2)$  amount of memory to store the auxiliary matrices for incremental Cholesky. Second, the meta-optimizer requires an additional  $O(T^2 D^2)$  computation time and memory for computing the GP energy-force kernel (see Equation 1), which is dominated by the computation of the Hessian of the kernel, assuming that the base GP kernel  $k$  can be computed in  $O(1)$ . These overheads are also present in standard SMB optimization, the difference being that  $T$  grows across optimizations in geometry meta-optimization.

The problem of scaling GP regression has been thoroughly studied and several solutions exist<sup>57–62</sup>, many of which essentially are clever methods for discarding the least informative data to train a GP on. In this work, we use a simpler strategy: after we have optimized a certain number of geometries, we initialize a fresh GP surrogate model using the previous surrogate model as a mean (*i.e.*, prior) function. Thus this strategy implicitly keeps the entire dataset it is trained on. Denzel et.

al<sup>3</sup> propose this strategy (called a multi-level GP) in their original work on SMB optimization for reducing the complexity of a single geometry optimization. In our case, we apply this strategy across geometry optimizations. We believe that there are many interesting ideas that can be explored in future work that carefully discards data that a GP is trained on over time in geometry meta-optimization (*e.g.*, use the distance between examples and their gradients). For the purposes of this paper, we start with the simpler one where we implicitly keep all the data.

More precisely, suppose we set a threshold  $M$  for the maximum number of geometries to optimize before reinitializing a fresh GP surrogate for geometry  $M$ . Then the following equations give what GP surrogate is used to optimize the  $m$ -th geometry:

$$f_{\mathcal{D}^m}^*, \mathbf{y}^m \leftarrow \begin{cases} \text{GP-SMB-OPT}(\text{GP}_0(f_{\mathcal{D}^{m-1}}^*, k), \mathbf{x}_m) & m \text{ multiple of } M \\ \text{GP-SMB-OPT}(f_{\mathcal{D}^{m-1}}^*, \mathbf{x}_m) & \text{otherwise} \end{cases} \quad (5)$$

where GP-SMB-OPT performs SMB optimization with the given GP surrogate and initial points, and produces a GP surrogate artifact and final geometry.  $\text{GP}_0(\mu, k)$  is notation for a GP surrogate that is fit to 0 data points. When we reach  $M$  geometries, or any multiple of  $M$ , we reinitialize a fresh GP surrogate, the difference being that we use the previous surrogate as a mean function ( $\text{GP}_0(f_{\mathcal{D}^{m-1}}^*, k)$ ). Thus this strategy enables us to implicitly keep the entire dataset while capping the training complexity at  $O(T^2 D^3)$  where  $T$  is the number of examples required to optimize  $M$  geometries. There is a cost for implicitly retaining the data: the complexity of prediction now queries  $O(M)$  surrogate model to evaluate the mean function.

### III. DATA & COMPUTATION INFORMATION

We choose MOPAC 2016<sup>63</sup> as the ESS and use the semiempirical PM7 Hamiltonian<sup>64</sup> within the unrestricted Hartree-Fock formalism<sup>65</sup>. The SCF energy convergence criterion was set to  $10^{-8}$  kcal/mol. All our experiments are run on small organic molecules (Section III A and Section III B), and so we expect PM7 to be sufficient.

We use mad-GP<sup>56</sup>, a Python software package that implements SMB optimization based on GPs for molecules and materials, to implement our geometry meta-optimizers. There are two reasons why we use this software package. First, it supports many features that we need for geometry meta-optimization including (1) directly reusing a GP surrogate used in one optimization for another optimization and (2) using trained surrogates as prior functions. Second and more importantly, it enables us to experiment with various covariance functions without the need to implement first or second-order derivatives manually via the technique of automatic differentiation<sup>66</sup>. In particular, we use this feature to implement a composition of Gaussian kernels with Coulomb representations.

#### A. ANI-1 Experiments

The set of experiments we describe in this section are aimed at providing initial *quantitative* evidence that geometry meta-optimization can work in the setting of using previous optimizations to accelerate optimization of the same atomistic system. Towards this end, we require a dataset of displaced geometries of various molecules that can act as a proxy in our experiments for the various initial geometries a computational chemist might supply to a geometry meta-optimizer. We use the ANI-1 dataset<sup>29</sup> for this purpose.

The ANI-1 dataset<sup>29</sup> contains 20 million off-equilibrium geometries of small organic molecules with up to 8 heavy atoms of oxygen, carbon, and nitrogen. In the published ANI-1 dataset, all structures are set to be neutral (*i.e.*, they carry zero charges) and singlet (*i.e.*, the spin multiplicity is 1); however, this does not mean that, with a more reliable high-level electronic structure method, these structures have singlet ground states. The reliability of the electronic structure method used for generating these off-equilibrium structures is beyond the scope of this paper. Herein, we simply use them as initial-guess structures for geometry optimizations. All the molecules were chosen from the GDB-11 database of chemically viable molecules<sup>67</sup>. Each molecule in ANI-1 has a varying number of geometries generated by first optimizing a molecule at DFT level of theory (with a particular density functional approximation and a small atomic basis set) and then displacing them along their normal modes at some finite temperature.

We select a subset of the ANI-1 dataset in our experiments to validate the concept of geometry meta-optimization. We use all molecules with 1 and 2 heavy atoms in the ANI-1 dataset. For molecules with a number of heavy atoms ranging from 3 to 8, we select uniformly at random 200 molecular formulas. For each formula, we select uniformly at random 18 geometries.

We set up three experiments to answer three questions concerning geometry meta-optimization.

1. How does the baseline geometry meta-optimizer compare with classic and SMB optimization? In particular, how does the number of ESC calculations scale as a function of number of previous geometry optimizations?
2. How much does rotational invariance affect the number of ESC calculations required for geometry optimization? We might expect that rotational invariance is important for geometry meta-optimization but less so for a single geometry optimization.
3. Given that training GP regression is computationally intensive, how does the baseline strategy of reusing previous surrogates as prior functions in future optimization affect geometry meta-optimization?

We use the following 6 geometry optimization strategies to answer these questions. In the definitions of GP surrogates,  $\mathbf{0}$  is a constant 0 mean function, rbf is a Gaussian kernel, and rbf  $\circ$  coulomb is a Gaussian kernel on a Coulomb representation of a molecule:

1. classic optimization using the eigenfollow optimizer in MOPAC<sup>63</sup>;
2. SMB optimization with surrogate GP( $\mathbf{0}$ , rbf);
3. geometry meta-optimization with surrogate GP( $\mathbf{0}$ , rbf);
4. geometry meta-optimization with surrogate GP( $\mathbf{0}$ , rbf  $\circ$  coulomb) so that it globally maintains rotational invariance;
5. geometry meta-optimization with surrogate GP( $\mathbf{0}$ , rbf) where geometries are reoriented at each step of optimization with respect to a given fixed geometry via Kabasch's algorithm<sup>68,69</sup> to (locally) approximate rotational invariance; and
6. geometry meta-optimization with surrogate GP( $\mathbf{0}$ , rbf  $\circ$  coulomb) where it is reinitialized after 9 optimizations and the previous surrogate is used as the mean function in the fresh GP surrogate for optimizing the remaining 9 geometries.

We design one experiment for each question.

1. The first experiment compares geometry optimization on the dataset with strategies 1, 2, and 3 to benchmark classic, SMB optimization, and geometry meta-optimization to address the first question.
2. The second experiment compares geometry optimization on the dataset with strategies 3, 4 and 5 to study the impact of rotational invariance in the surrogate PES for SMB optimization and geometry meta-optimization to address the second question.
3. The third experiment compares geometry optimization on the dataset with strategies 4 and 6 to study the effect of using another geometry meta-optimization strategy to reduce the computational cost of training to address the third question.

In all experiments reported in this paper, the GPs jointly predict energy and force. We also considered GPs that predict only forces and Matérn kernels (considered state-of-the-art), but the results were similar to those based on Gaussian kernels. For all the experiments, we use vanilla SMB optimization with GP surrogates: (1) use the surrogate to predict the next geometry, (2) set the prior to be the maximum energy seen during optimization unless the GP is using a previous GP surrogate as prior, (3) train the surrogate with the geometry, and energy and forces obtained from an ESS, (4) test convergence and that the energy has decreased compared to the previous geometry, and (5) repeat if not converged.

The convergence criterion we use is determined by the ESC and not by the surrogate. The optimizer converges if the maximum component of the absolute value of the forces is less than  $0.03\text{eV}/\text{\AA}$  and that the norm of the forces is less than  $2/3 \cdot \sqrt{3N} \cdot 0.03\text{eV}/\text{\AA}$ . Note that this convergence criterion is tighter compared to optimization with MOPAC where we enforce the same condition on the norm of the forces using MOPAC's "GNORM" keyword. Prior work<sup>3</sup> has shown that SMB optimization is competitive with classic optimization, and so we use MOPAC here as a correctness check rather than as a direct head-to-head comparison.

All these experiments are run without any hyperparameter tuning. The lengthscale for GPs with the Gaussian kernel

was set at 0.3 for every molecule and during geometry meta-optimization on all geometries. Similarly, the lengthscale for GPs with the Gaussian kernel composed with a Coulomb kernel was set at 8.0 for every molecule and during geometry meta-optimization on all geometries.

## B. Conformational Analysis

As a more stringent test of the behavior of geometry meta-optimization, we also perform an experiment involving conformation search. Towards this end, we use a hand-constructed dataset of 10 small alcohols and hydrocarbons (see Table I in results) where we manually identify and rotate salient dihedral bonds.

The alcohols and hydrocarbons contain between 4 and 8 heavy atoms and all the bonds are single bonds. For each alcohol or hydrocarbon, we identify the salient dihedral bonds as those that involve a C-C-C-C bond or C-C-O-H bond. For each of these dihedral bonds, we generate all combinations of geometries obtainable by rotations of those bonds by  $0^\circ, 120^\circ, 240^\circ$ . This strategy obviously does not scale when there are a large number of dihedral angles to consider. We filter out those geometries whose smallest inter-atomic distance is smaller than  $0.74\text{\AA}$  (*i.e.*, the H-H bond length in the  $\text{H}_2$  molecule). We leave a more sophisticated study of geometry meta-optimization (*e.g.*, refined grid of dihedral angles, tighter convergence criterion, better criteria for checking whether two structures are identical, a more extensive test on using GP/meta for conformational search) for conformation search for future work. Conformational analysis is an area of application where reducing the cumulative number of ESCs is a better metric compared to reducing the ESCs for a single geometry optimization, and so would be an interesting application of a geometry meta-optimizer.

For each alcohol or hydrocarbon, we run geometry meta-optimization with strategy 6, *i.e.*, with surrogate GP( $\mathbf{0}$ , rbf  $\circ$  coulomb) where the GP surrogate is reinitialized every 9 optimizations and the previous surrogate is used as the mean function in the fresh GP surrogate for optimizing the remaining geometries. We do not test other strategies because our experimentation with ANI-1 indicates the effectiveness of this GP surrogate and the potentially large number of conformers constrains us to use a computationally efficient strategy. For all the molecules, we use a lengthscale of 4.0.

The convergence criterion is more critical in conformational search. We use the convergence criterion that the norm of the forces is less than  $2/3 \cdot \sqrt{3N} \cdot 0.03\text{eV}/\text{\AA}$  for both MOPAC and geometry meta-optimization. Note that this convergence criterion is too loose for actual conformational analysis<sup>70</sup>. For the current experiment, we can take the final geometries obtained from both MOPAC and geometry meta-optimization and further optimize it with MOPAC to tighter convergence criterion using MOPAC's "PRECISE" keyword.

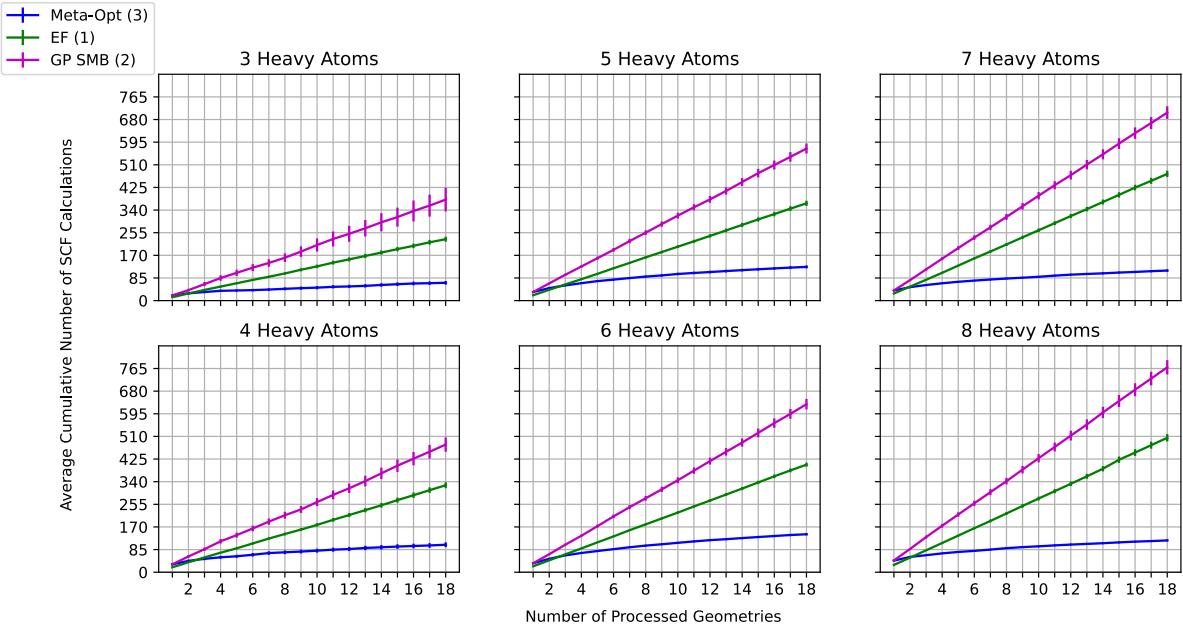


Figure 3: Comparison of optimization cycles for classic and ML optimizers. The number in the parentheses in each strategy refers to the indices of the strategies in Section III A. For example, EF is the classic (eigenfollowing) optimization which corresponds to strategy 1.

## IV. RESULTS

Our results indicate that geometry meta-optimizers with GP surrogates work and are competitive with classic optimizers and SBM optimization. Section IV A contains the results of the experiments described in Section III A. Section III B contains the results of the experiment described in Section IV B.

### A. ANI-1 Experiments

In each of the following plots, we report the average cumulative number of SCF calculations after optimization of each of the 18 initial geometries. The error bars represent the standard deviations of the number of SCF calculations of different molecules (which have the same number of heavy atoms). We separate the results by number of heavy atoms, ranging from 3 to 8, to provide some insight on scalability to large molecules.

We verify that the energy of the molecules optimized by the SMB optimizers and geometry meta-optimizers are comparable to that of MOPAC. The average energy difference between the baseline geometry meta-optimizer and MOPAC is  $-0.21$  kcal/mol, a standard deviation of 1.27, and a 99th percentile of 0.2 kcal/mol. Recall that the convergence criteria of the ML-based optimizers are tighter so it is expected that they would find lower energy geometries.

In some rare cases, the SBM optimizer or geometry meta-optimizer fails. This typically happens on smaller molecules and for the first few geometries when there is still little information about the PES. The SBM optimizer fails on one exper-

iment. The baseline geometry meta-optimizer fails on 9 experiments. The geometry meta-optimizer with Coulomb representation fails on 25 experiments. In contrast, the MOPAC optimizer never failed. We believe that there are simple ideas that would prevent such failures which are essentially due to a bad guess at a PES, but our goal is to study an off-the-shelf geometry meta-optimizer with no tuning of hyperparameters or engineering. These failures would also be avoided by taking following the pragmatic approach and having the very first optimization done by MOPAC rather than the geometry meta-optimizer.

*a. How does the baseline geometry meta-optimizer compare with classic and SMB optimization?* We compare the results of running the MOPAC optimizer (EF) against an SBM optimizer with and without geometry meta-optimization (Figure 3).

There are several points to keep in mind for interpreting the comparison of SMB optimization (strategy 2) with MOPAC EF (strategy 1). First, previous studies of SMB optimization with GP surrogates benchmarks against L-BFGS and demonstrates that it is superior<sup>3</sup>. In our case, we compare against a more advanced EF optimizer. The EF optimizer has access to an estimate of the Hessian during optimization for controlling the step size and determining convergence (*e.g.*, minima or transition state) whereas the SMB optimizer uses the Hessian of the kernel to fit the force obtained from a ESC. Consequently, we have an unfair comparison for SMB optimization because the EF optimizer has access to more information and a more challenging test of geometry meta-optimization. Second, we emphasize that the SMB optimizer

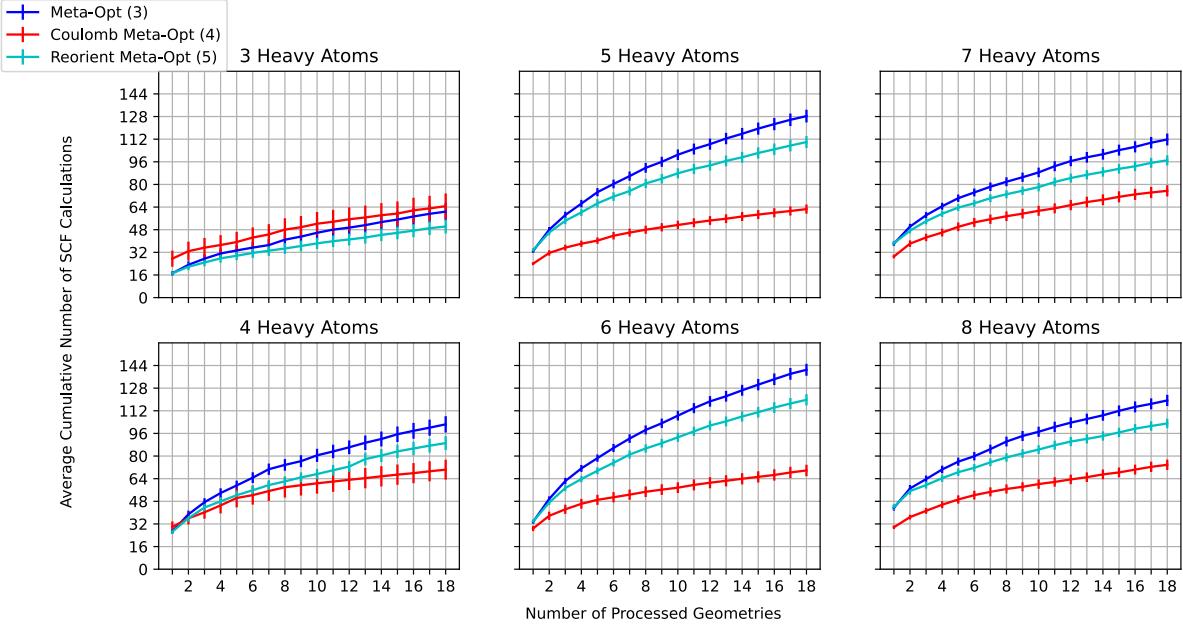


Figure 4: Effect of symmetry representation. The number in the parentheses in each strategy refers to the indices of the strategies in Section III A.

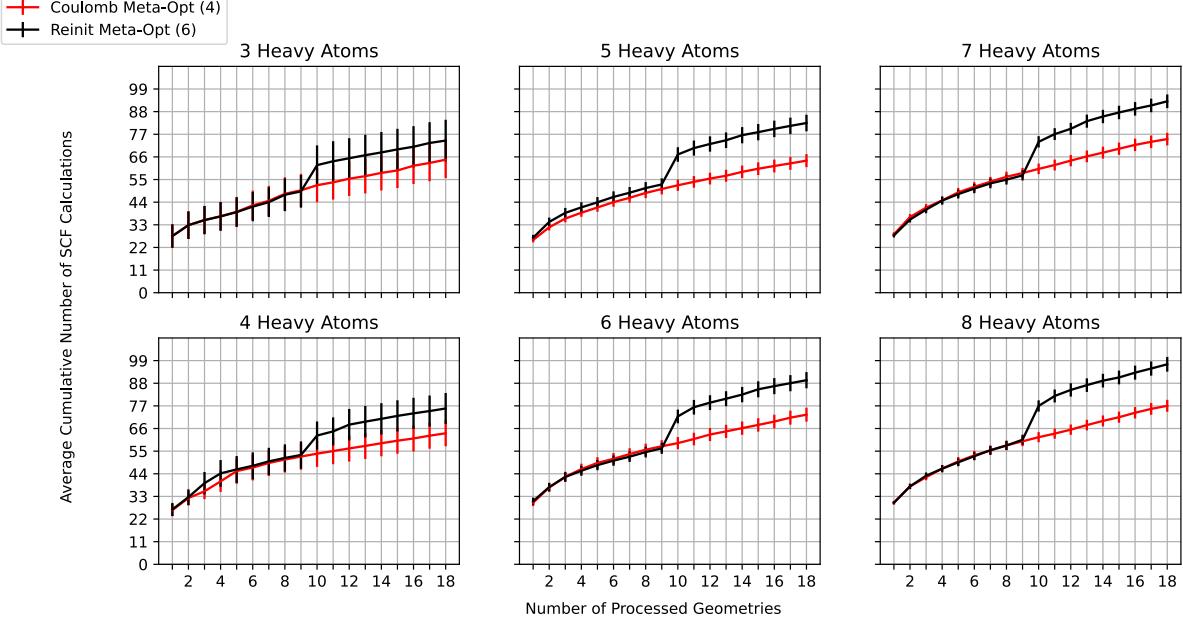


Figure 5: Effect of discarding data and learning from past model. The number in the parentheses in each strategy refers to the indices of the strategies in Section III A.

in this paper is a vanilla optimizer in that it does not use any advanced optimization techniques that are typically used in SMB geometry optimization. We believe that we could improve the performance of SMB optimization by implementing adaptive length-scales, overshooting, and fine-tuning the hy-

perparameters<sup>3</sup>. Third, the SBM optimizer and the geometry meta-optimizer use a tighter convergence criterion compared to MOPAC (MOPAC only checks that the norm of the gradient is small). With all of these points in mind, we observe that SMB optimization (strategy 2) is less competitive against

MOPAC EF optimization.

In contrast, the geometry meta-optimizer (strategy 3) is very competitive against MOPAC, even with the tighter convergence criterion, and its curve appears to be sub-linear. Like the SMB optimizer, the geometry meta-optimizer also does not use any clever optimization heuristics typically applied in standard SBM optimization. Geometry meta-optimization thus seems to alleviate the need for advanced engineering of the SBM optimizer. This is crucial not only because optimizers need to work “off-the-shelf”, but also because careful engineering of an SBM optimizer inevitably leads to overfitting. Note that the geometry meta-optimizer’s only difference compared to SMB optimization is the sharing of surrogate models across optimizations. Thus the reduction in the cumulative number of ESCs required for optimizing a set of geometries between the two is solely due to adapting surrogate models from previous geometry optimizations to the next.

*b. How much does rotational invariance affect the number of ESC calculations required for geometry meta-optimization?* We now compare the simplest geometry meta-optimizer (strategy 3) to one that reorient the molecule at every step of the optimization (strategy 5) and one that uses a Coulomb representation (strategy 4) (Figure 4). Both methods improve on the simplest geometry meta-optimizer, and the Coulomb representation method appears to have a significant advantage as the number of heavy atoms increases. Note further that the Coulomb representation, which guarantees global rotational and translation invariance, is also superior to reorientation, which only locally approximates rotational invariance. In particular, the marginal increase in ESCs required decreases as a function of geometries optimized more for the Coulomb representation as compared to reorientation. This offers evidence that geometry meta-optimizers benefit more from physically accurate surrogates compared to surrogates that are only used once. Indeed, at time step 1, the geometry meta-optimizer is simply a SMB optimizer, and the performance of a Gaussian kernel and reoriented kernel is essentially identical.

*c. Given that training GP regression is computationally intensive, how does the baseline strategy of reusing previous surrogates as prior functions in future optimization affect geometry meta-optimization?* Finally, we compare the baseline geometry meta-optimization with Coulomb representation (strategy 4) to one that reinitializes a fresh GP surrogate every  $M$  steps and captures the previous surrogate through its mean function for the purposes of managing computational complexity (strategy 6) (Figure 5). Observe that at the geometry indexed 9 in each plot, there is a jump in the curve for additional number of SCF calculations required. This jump occurs because we have reinitialized a fresh GP surrogate. Note that the jump is not as large as the value for the first geometry, *i.e.*, standard SMB optimization. This suggests that reusing previous surrogates as a prior, which implicitly encodes previously seen data, is helpful for reducing the number of SCF calculations. However, the performance is worse than that of the baseline geometry meta-optimizer where we train with all the data points. Thus this strategy is beneficial in making geometry meta-optimization more computationally tractable without

sacrificing much performance.

## B. Conformational Analysis Experiments

Table I gives the results of our conformational analysis experiments on the alcohol and hydrocarbon dataset. We observe that the number of SCF calculations required for geometry meta-optimization is indeed less than that of a baseline MOPAC strategy. The column labeled “Original” gives the number of conformers found after the experiment is run with a standard convergence criterion (which is loose for conformational analysis). To compute a conformer, we check that the energy difference is less than  $10^{-3}$  kcal/mol. Note that we have not at this point optimized with a tight convergence criterion so that many of these conformers may end up being the same structure. We take the final geometries for both MOPAC and geometry meta-optimization and optimize them both with MOPAC using the “PRECISE” keyword. This result is reported in the column labeled “Precise”.

Figure 6 gives a more detailed energy ladder of each final structure found by MOPAC and geometry meta-optimization. Perhaps the most surprising result is that geometry meta-optimization followed by MOPAC precise (M-P) tends to miss higher energy conformers compared to MOPAC (S-P). Moreover, MOPAC appears to find a larger diversity of conformers compared to geometry meta-optimization. We have verified that SMB optimization with GP surrogates produces similar results to geometry meta-optimization, and so the bias towards lower energy conformers appears to be a feature of using a GP surrogate. We believe that further investigation of the divergence between the conformers produced by MOPAC and geometry meta-optimization warrants further investigation and can shed light on how exactly surrogates are modeling PESs in a manner useful for geometry optimization.

## V. CONCLUSION

In this paper, we explored the concept of geometry meta-optimization in the context of using past optimizations to improve optimization of the *same* atomistic system from similar initial geometries. We show that geometry meta-optimization is competitive with classic and SMB optimization for accelerating a single geometry optimization. This setting of geometry meta-optimization is the first step in exploring the possibility of using geometry meta-optimization to accelerate geometry optimization of similar atomistic systems (*e.g.*, varying numbers of elements or replacing elements with chemically similar ones).

We emphasize that surrogates play different roles in SMB optimization compared to geometry meta-optimization: whereas a surrogate in SMB optimization is designed to reduce the number of ESCs for a single geometry optimization, a surrogate in geometry meta-optimization can distribute the reduction of ESCs across a set of geometry optimizations. This difference is why we may hope that geometry meta-optimization can generalize features of a PES that are helpful

Molecule	Original SCF MOPAC	Original Mopac	Additional SCF Mopac	Precise Mopac				
	Meta	Meta	Meta	Meta				
<chem>CH3CH2CH(OH)CH2CH2CH2(OH)</chem>	6255	839	191	181	29719	20840	174	113
<chem>CH3CH2CH2CH2CH(OH)CH3</chem>	2130	330	74	69	8072	5762	64	29
<chem>CH3CH2CH2CH(OH)CH2CH3</chem>	2483	346	74	67	8735	5523	64	41
<chem>CH3CH2CH2CH2CH2CH2(OH)</chem>	1367	216	51	42	5549	3815	38	21
<chem>CH3CH(OH)CH2CH2CH3</chem>	569	85	16	12	2574	1121	10	3
<chem>CH3CH2CH2CH2CH2(OH)</chem>	637	119	25	24	2335	1752	23	15
<chem>CH3CH2CH2CH2(OH)</chem>	612	105	26	24	2003	1595	14	8
<chem>CH3CH(OH)CH2CH3</chem>	151	54	9	7	776	514	7	6
<chem>CH3CH2CH(OH)CH2CH3</chem>	129	86	4	4	760	429	4	4
<chem>CH3CH2CH2CH3</chem>	31	11	3	3	148	150	2	2

Table I: Conformational analysis results. The columns labeled “Original” and “Precise” give the number of unique conformers found after optimizing to a local minima (which is loose for conformational search) and extremely-tight convergence criterion respectively. The column labeled “Original SCF” gives the number of SCF calculations required for optimization to a standard convergence criterions and the column labeled “Additional SCF” gives the additional number of SCF calculations required for precise convergence.

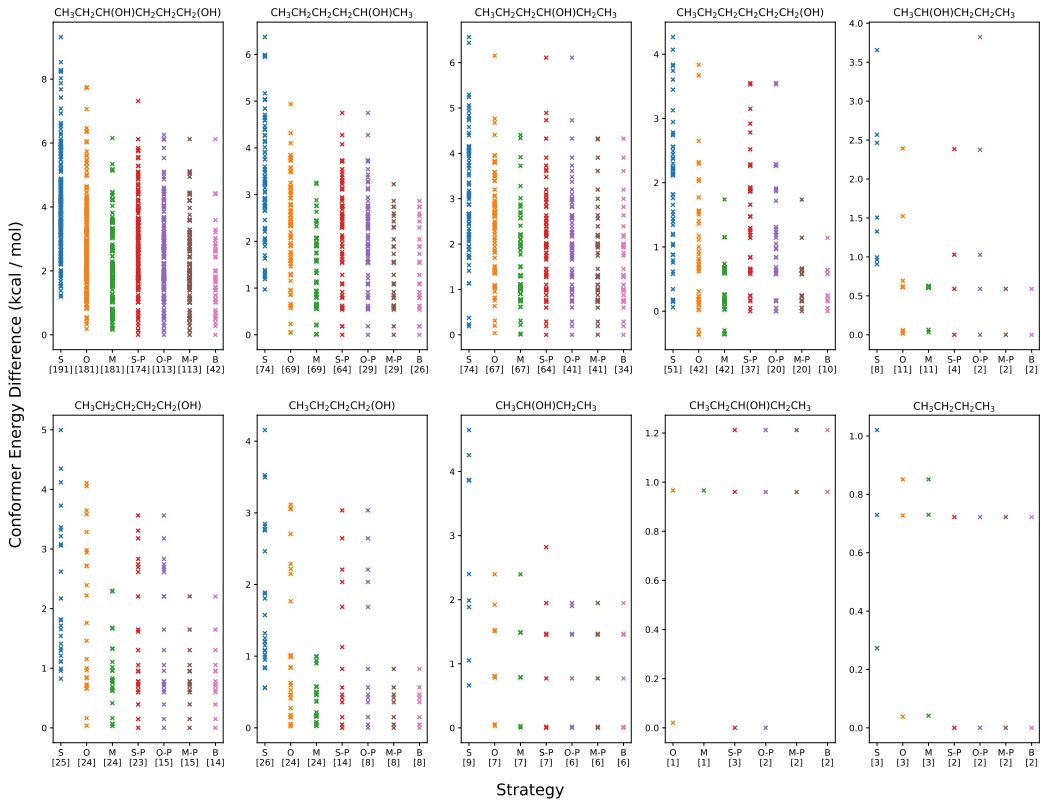


Figure 6: Comparing conformation search for alcohols and hydrocarbons. The relative energies (in kcal/mol) of all conformers of a given molecule are computed with respect to the absolute electronic-structure energy of the lowest-energy conformer found across all strategies. The number in the square brackets indicates the number of conformers found by that strategy. Strategies with looser convergence are MOPAC (S), SMB optimization (O), and geometry meta-optimization (M). Strategies with precise convergence are MOPAC-precise (S-P), SMB optimization + MOPAC-precise (O-P), and geometry meta-optimization + MOPAC-precise (M-P). The column labled (B) highlights those conformers that are found by both S-P and M-P.

for future optimization. As a reminder, we can conceptually accomplish geometry meta-optimization either by sharing all the data generated from previous geometry optimizations or by sharing previous surrogate models, which implicitly incorporate the data. The reason why we focus on sharing surrogate models in this paper is due to the complexity of training: in geometry meta-optimization, we need to retrain the surrogate model during each step of optimization which is impractical if we were to retrain on an already large dataset from scratch. Thus the application of a surrogate to geometry meta-optimization shifts the focus from constructing a good fit of ESC data to how to efficiently adapt previous surrogates for the current geometry optimization.

Our initial experiment with conformational analysis raises an interesting question concerning SMB optimization and geometry meta-optimization with GP surrogates. Notably, we observe on our alcohol and hydrocarbon dataset that while GP surrogates are effective at reducing the number of ESCs, they also tend to miss higher energy conformers. We believe that further research in characterizing the divergence of GP surrogates from classic optimizers is interesting to pursue.

## ACKNOWLEDGMENTS

This work was supported by Oracle Labs and by the Schiller Institute Grant for Exploratory Collaborative Scholarship (SIGECS). We thank the Systems Team in Academic Technology at San Francisco State University and the Boston College Linux Cluster Center for cluster computing resources. We thank Julian Asilis and Weiming Qin for helpful discussions.

## DATA AVAILABILITY

The data that support the findings of this study are available within the article and its supplementary material.

## REFERENCES

- <sup>1</sup>W. Kohn and L. J. Sham, "Self-Consistent Equations Including Exchange and Correlation Effects," *Physical Review* **140**, A1133–A1138 (1965).
- <sup>2</sup>W. Kohn, A. D. Becke, and R. G. Parr, "Density Functional Theory of Electronic Structure," *The Journal of Physical Chemistry* **100**, 12974–12980 (1996).
- <sup>3</sup>A. Denzel and J. Kästner, "Gaussian process regression for geometry optimization," *The Journal of Chemical Physics* **148**, 094114 (2018).
- <sup>4</sup>A. Denzel, B. Haasdonk, and J. Kästner, "Gaussian Process Regression for Minimum Energy Path Optimization and Transition State Search," *The Journal of Physical Chemistry A* **123**, 9600–9611 (2019).
- <sup>5</sup>A. Denzel and J. Kästner, "Gaussian Process Regression for Transition State Search," *Journal of Chemical Theory and Computation* **14**, 5777–5786 (2018).
- <sup>6</sup>A. Denzel and J. Kästner, "Hessian Matrix Update Scheme for Transition State Search Based on Gaussian Process Regression," *Journal of Chemical Theory and Computation* **16**, 5083–5089 (2020).
- <sup>7</sup>R. Meyer and A. W. Hauser, "Geometry optimization using Gaussian process regression in internal coordinate systems," *The Journal of Chemical Physics* **152**, 084112 (2020).
- <sup>8</sup>O.-P. Koistinen, V. Ásgeirsson, A. Vehtari, and H. Jónsson, "Nudged Elastic Band Calculations Accelerated with Gaussian Process Regression Based on Inverse Interatomic Distances," *Journal of Chemical Theory and Computation* **15**, 6738–6751 (2019).
- <sup>9</sup>O.-P. Koistinen, V. Ásgeirsson, A. Vehtari, and H. Jónsson, "Minimum Mode Saddle Point Searches Using Gaussian Process Regression with Inverse-Distance Covariance Function," *Journal of Chemical Theory and Computation* **16**, 499–509 (2020).
- <sup>10</sup>H. Sugisawa, T. Ida, and R. V. Krems, "Gaussian process model of 51-dimensional potential energy surface for protonated imidazole dimer," *The Journal of Chemical Physics* **153**, 114101 (2020).
- <sup>11</sup>P. Rowe, V. L. Deringer, P. Gasparotto, G. Csányi, and A. Michaelides, "An accurate and transferable machine learning potential for carbon," *The Journal of Chemical Physics* **153**, 034702 (2020).
- <sup>12</sup>A. P. Bartók, M. C. Payne, R. Kondor, and G. Csányi, "Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons," *Physical Review Letters* **104**, 136403 (2010).
- <sup>13</sup>O. T. Unke, S. Chmiela, H. E. Sauceda, M. Gastegger, I. Poltavsky, K. T. Schütt, A. Tkatchenko, and K.-R. Müller, "Machine Learning Force Fields," *Chemical Reviews* **121**, 10142–10186 (2021).
- <sup>14</sup>E. Garijo del Río, J. J. Mortensen, and K. W. Jacobsen, "Local Bayesian optimizer for atomic structures," *Physical Review B* **100**, 104103 (2019).
- <sup>15</sup>S. Chmiela, A. Tkatchenko, H. E. Sauceda, I. Poltavsky, K. T. Schütt, and K.-R. Müller, "Machine learning of accurate energy-conserving molecular force fields," *Science Advances* (2017), 10.1126/sciadv.1603015.
- <sup>16</sup>H. E. Sauceda, S. Chmiela, I. Poltavsky, K.-R. Müller, and A. Tkatchenko, "Molecular force fields with gradient-domain machine learning: Construction and application to dynamics of small molecules with coupled cluster forces," *The Journal of Chemical Physics* **150**, 114102 (2019).
- <sup>17</sup>S. Chmiela, H. E. Sauceda, I. Poltavsky, K.-R. Müller, and A. Tkatchenko, "sGML: Constructing accurate and data efficient molecular force fields using machine learning," *Computer Physics Communications* **240**, 38–45 (2019).
- <sup>18</sup>A. Glielmo, C. Zeni, and A. De Vita, "Efficient nonparametric \$n\$-body force fields from machine learning," *Physical Review B* **97**, 184307 (2018).
- <sup>19</sup>V. L. Deringer, A. P. Bartók, N. Bernstein, D. M. Wilkins, M. Ceriotti, and G. Csányi, "Gaussian Process Regression for Materials and Molecules," *Chemical Reviews* **121**, 10073–10141 (2021).
- <sup>20</sup>G. Schmitz and O. Christiansen, "Gaussian process regression to accelerate geometry optimizations relying on numerical differentiation," *The Journal of Chemical Physics* **148**, 241704 (2018).
- <sup>21</sup>R. Archibald, J. T. Krogel, and P. R. C. Kent, "Gaussian process based optimization of molecular geometries using statistically sampled energy surfaces from quantum Monte Carlo," *The Journal of Chemical Physics* **149**, 164116 (2018).
- <sup>22</sup>C. Finn, P. Abbeel, and S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *Proceedings of the 34th International Conference on Machine Learning (PMLR)*, 2017 pp. 1126–1135.
- <sup>23</sup>A. Nichol, J. Achiam, and J. Schulman, "On First-Order Meta-Learning Algorithms," arXiv:1803.02999 [cs] (2018), arXiv:1803.02999 [cs].
- <sup>24</sup>J. Yoon, T. Kim, O. Dia, S. Kim, Y. Bengio, and S. Ahn, "Bayesian Model-Agnostic Meta-Learning," in *Advances in Neural Information Processing Systems*, Vol. 31 (Curran Associates, Inc., 2018).
- <sup>25</sup>R. S. Sutton and A. G. Barto, *Reinforcement Learning, Second Edition: An Introduction* (MIT Press, 2018).
- <sup>26</sup>J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," in *Advances in Neural Information Processing Systems*, Vol. 25 (Curran Associates, Inc., 2012).
- <sup>27</sup>B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the Human Out of the Loop: A Review of Bayesian Optimization," *Proceedings of the IEEE* **104**, 148–175 (2016).
- <sup>28</sup>B. Settles, "Active Learning Literature Survey," Technical Report (University of Wisconsin-Madison Department of Computer Sciences, 2009).
- <sup>29</sup>J. S. Smith, O. Isayev, and A. E. Roitberg, "ANI-1, A data set of 20 million calculated off-equilibrium conformations for organic molecules," *Scientific Data* **4**, 170193 (2017).
- <sup>30</sup>J. S. Smith, O. Isayev, and A. E. Roitberg, "ANI-1: An extensible neural network potential with DFT accuracy at force field computational cost," *Chemical Science* **8**, 3192–3203 (2017).

- <sup>31</sup>Ö. Farkas and H. B. Schlegel, "Methods for optimizing large molecules. Part III. An improved algorithm for geometry optimization using direct inversion in the iterative subspace (GDIIS)," *Physical Chemistry Chemical Physics* **4**, 11–15 (2002).
- <sup>32</sup>X. Li and M. J. Frisch, "Energy-Represented Direct Inversion in the Iterative Subspace within a Hybrid Geometry Optimization Method," *Journal of Chemical Theory and Computation* **2**, 835–839 (2006).
- <sup>33</sup>A. Banerjee, N. Adams, J. Simons, and R. Shepard, "Search for stationary points on surfaces," *The Journal of Physical Chemistry* **89**, 52–57 (1985).
- <sup>34</sup>S. Manzhos and T. Carrington, "A random-sampling high dimensional model representation neural network for building potential energy surfaces," *The Journal of Chemical Physics* **125**, 084109 (2006).
- <sup>35</sup>J. Behler and M. Parrinello, "Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces," *Physical Review Letters* **98**, 146401 (2007).
- <sup>36</sup>J. Behler, "Representing potential energy surfaces by high-dimensional neural network potentials," *Journal of Physics. Condensed Matter: An Institute of Physics Journal* **26**, 183001 (2014).
- <sup>37</sup>J. Behler, "Perspective: Machine learning potentials for atomistic simulations," *The Journal of Chemical Physics* **145**, 170901 (2016).
- <sup>38</sup>K. Schütt, P.-J. Kindermans, H. E. Sauceda Felix, S. Chmiela, A. Tkatchenko, and K.-R. Müller, "SchNet: A continuous-filter convolutional neural network for modeling quantum interactions," in *Advances in Neural Information Processing Systems*, Vol. 30 (Curran Associates, Inc., 2017).
- <sup>39</sup>K. T. Schütt, H. E. Sauceda, P.-J. Kindermans, A. Tkatchenko, and K.-R. Müller, "SchNet – A deep learning architecture for molecules and materials," *The Journal of Chemical Physics* **148**, 241722 (2018).
- <sup>40</sup>K. T. Schütt, P. Kessel, M. Gastegger, K. A. Nicoli, A. Tkatchenko, and K.-R. Müller, "SchNetPack: A Deep Learning Toolbox For Atomistic Systems," *Journal of Chemical Theory and Computation* **15**, 448–455 (2019).
- <sup>41</sup>L. Zhang, J. Han, H. Wang, W. Saidi, R. Car, and W. E, "End-to-end Symmetry Preserving Inter-atomic Potential Energy Model for Finite and Extended Systems," in *Advances in Neural Information Processing Systems*, Vol. 31 (Curran Associates, Inc., 2018).
- <sup>42</sup>O. T. Unke and M. Meuwly, "PhysNet: A Neural Network for Predicting Energies, Forces, Dipole Moments, and Partial Charges," *Journal of Chemical Theory and Computation* **15**, 3678–3693 (2019).
- <sup>43</sup>B. Anderson, T. S. Hy, and R. Kondor, "Cormorant: Covariant Molecular Neural Networks," in *Advances in Neural Information Processing Systems*, Vol. 32 (Curran Associates, Inc., 2019).
- <sup>44</sup>J. Klicpera, J. Groß, and S. Günnemann, "Directional Message Passing for Molecular Graphs," *International Conference on Learning Representations* (2020).
- <sup>45</sup>C. K. I. Williams and C. E. Rasmussen, *Gaussian Processes for Machine Learning*, Vol. 2 (MIT press Cambridge, MA, 2006).
- <sup>46</sup>A. A. Peterson, R. Christensen, and A. Khorshidi, "Addressing uncertainty in atomistic machine learning," *Physical Chemistry Chemical Physics* **19**, 10978–10985 (2017).
- <sup>47</sup>E. V. Podryabinkin and A. V. Shapeev, "Active learning of linearly parametrized interatomic potentials," *Computational Materials Science* **140**, 171–180 (2017).
- <sup>48</sup>N. Bernstein, G. Csányi, and V. L. Deringer, "De novo exploration and self-guided learning of potential-energy surfaces," *npj Computational Materials* **5**, 1–9 (2019).
- <sup>49</sup>R. Jinnouchi, F. Karsai, and G. Kresse, "On-the-fly machine learning force field generation: Application to melting points," *Physical Review B* **100**, 014105 (2019).
- <sup>50</sup>M. K. Bisbo and B. Hammer, "Efficient Global Structure Optimization with a Machine-Learned Surrogate Model," *Physical Review Letters* **124**, 086102 (2020).
- <sup>51</sup>F. Musil, A. Grisafi, A. P. Bartók, C. Ortner, G. Csányi, and M. Ceriotti, "Physics-Inspired Structural Representations for Molecules and Materials," *Chemical Reviews* **121**, 9759–9815 (2021).
- <sup>52</sup>M. Rupp, A. Tkatchenko, K.-R. Müller, and O. A. von Lilienfeld, "Fast and Accurate Modeling of Molecular Atomization Energies with Machine Learning," *Physical Review Letters* **108**, 058301 (2012).
- <sup>53</sup>B. J. Braams and J. M. Bowman, "Permutationally invariant potential energy surfaces in high dimensionality," *International Reviews in Physical Chemistry* **28**, 577–606 (2009).
- <sup>54</sup>Z. Xie and J. M. Bowman, "Permutationally Invariant Polynomial Basis for Molecular Energy Surface Fitting via Monomial Symmetrization," *Journal of Chemical Theory and Computation* **6**, 26–34 (2010).
- <sup>55</sup>B. Jiang and H. Guo, "Permutation invariant polynomial neural network approach to fitting potential energy surfaces," *The Journal of Chemical Physics* **139**, 054112 (2013).
- <sup>56</sup>D. Huang, C. Teng, J. L. Bao, and J.-B. Tristan, "mad-gp: Automatic differentiation of gaussian processes for molecules and materials," *Journal of Mathematical Chemistry* (in press).
- <sup>57</sup>J. Quiñonero-Candela and C. E. Rasmussen, "A Unifying View of Sparse Approximate Gaussian Process Regression," *The Journal of Machine Learning Research* **6**, 1939–1959 (2005).
- <sup>58</sup>E. Snelson and Z. Ghahramani, "Sparse Gaussian Processes using Pseudo-inputs," in *Advances in Neural Information Processing Systems*, Vol. 18 (MIT Press, 2006).
- <sup>59</sup>J. Hensman, N. Fusi, and N. D. Lawrence, "Gaussian processes for Big data," in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI'13* (AUAI Press, Arlington, Virginia, USA, 2013) pp. 282–290.
- <sup>60</sup>Q. Le, T. Sarlos, and A. Smola, "Fastfood - Computing Hilbert Space Expansions in loglinear time," in *Proceedings of the 30th International Conference on Machine Learning* (PMLR, 2013) pp. 244–252.
- <sup>61</sup>A. Wilson and H. Nickisch, "Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)," in *Proceedings of the 32nd International Conference on Machine Learning* (PMLR, 2015) pp. 1775–1784.
- <sup>62</sup>D. Eriksson, K. Dong, E. Lee, D. Bindel, and A. G. Wilson, "Scaling Gaussian Process Regression with Derivatives," in *Advances in Neural Information Processing Systems*, Vol. 31 (Curran Associates, Inc., 2018).
- <sup>63</sup>J. J. P. Stewart, "Mopac2016," Stewart Computational Chemistry, Colorado Springs, CO, USA (2016).
- <sup>64</sup>J. J. P. Stewart, "Optimization of parameters for semiempirical methods VI: More modifications to the NDDO approximations and re-optimization of parameters," *Journal of Molecular Modeling* **19**, 1–32 (2013).
- <sup>65</sup>J. A. Pople and R. K. Nesbet, "Self-Consistent Orbitals for Radicals," *The Journal of Chemical Physics* **22**, 571–572 (1954).
- <sup>66</sup>A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research* **18**, 1–43 (2018).
- <sup>67</sup>T. Fink and J.-L. Reymond, "Virtual exploration of the chemical universe up to 11 atoms of C, N, O, F: Assembly of 26.4 million structures (110.9 million stereoisomers) and analysis for new ring systems, stereochemistry, physicochemical properties, compound classes, and drug discovery," *Journal of Chemical Information and Modeling* **47**, 342–353 (2007 Mar-Apr).
- <sup>68</sup>W. Kabsch, "A solution for the best rotation to relate two sets of vectors," *Acta Crystallographica Section A: Crystal Physics, Diffraction, Theoretical and General Crystallography* **32**, 922–923 (1976).
- <sup>69</sup>J. C. Kromann, "Calculate root-mean-square deviation (rmsd) of two molecules using rotation," (2021), software available from <http://github.com/charnley/rmsd>, v1.4.
- <sup>70</sup>J. L. Bao, L. Xing, and D. G. Truhlar, "Dual-Level Method for Estimating Multistructural Partition Functions with Torsional Anharmonicity," *Journal of Chemical Theory and Computation* **13**, 2511–2522 (2017).
- <sup>71</sup>P. O. Dral, "Quantum Chemistry in the Age of Machine Learning," *The Journal of Physical Chemistry Letters* **11**, 2336–2347 (2020).
- <sup>72</sup>M. J. Burn and P. L. A. Popelier, "Creating Gaussian process regression models for molecular simulations using adaptive sampling," *The Journal of Chemical Physics* **153**, 054111 (2020).
- <sup>73</sup>O.-P. Koistinen, F. B. Dagbjartsdóttir, V. Ásgeirsson, A. Vehtari, and H. Jónsson, "Nudged elastic band calculations accelerated with Gaussian process regression," *The Journal of Chemical Physics* **147**, 152720 (2017).
- <sup>74</sup>I. Fdez. Galván, G. Raggi, and R. Lindh, "Restricted-Variance Constrained, Reaction Path, and Transition State Molecular Optimizations Using Gradient-Enhanced Kriging," *Journal of Chemical Theory and Computation* **17**, 571–582 (2021).
- <sup>75</sup>A. P. Bartók, R. Kondor, and G. Csányi, "On representing chemical environments," *Physical Review B* **87**, 184115 (2013).
- <sup>76</sup>S. De, A. P. Bartók, G. Csányi, and M. Ceriotti, "Comparing molecules and solids across structural and alchemical space," *Physical Chemistry Chemical Physics* **18**, 13754–13769 (2016).

- <sup>77</sup>L. Himanen, M. O. J. Jäger, E. V. Morooka, F. Federici Canova, Y. S. Ranawat, D. Z. Gao, P. Rinke, and A. S. Foster, “DScribe: Library of descriptors for machine learning in materials science,” *Computer Physics Communications* **247**, 106949 (2020).
- <sup>78</sup>M. W. Walker, L. Shao, and R. A. Volz, “Estimating 3-d location parameters using dual number quaternions,” *CVGIP: Image Understanding* **54**, 358–367 (1991).
- <sup>79</sup>A. H. Larsen, J. J. Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dulak, J. Friis, M. N. Groves, B. Hammer, C. Hargus, E. D. Hermes, P. C. Jennings, P. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjer, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, and K. W. Jacobsen, “The atomic simulation environment—a Python library for working with atoms,” *Journal of Physics: Condensed Matter* **29**, 273002 (2017).
- <sup>80</sup>A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett (Curran Associates, Inc., 2019) pp. 8024–8035.
- <sup>81</sup>P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods* **17**, 261–272 (2020).
- <sup>82</sup>J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, “JAX: composable transformations of Python+NumPy programs,” (2018), software available from <http://github.com/google/jax>, v0.2.5.
- <sup>83</sup>L. Biewald, “Experiment tracking with weights and biases,” (2020), software available from [wandb.com](http://wandb.com).
- <sup>84</sup>J. Baker and F. Chan, “The location of transition states: A comparison of Cartesian, Z-matrix, and natural internal coordinates,” *Journal of Computational Chemistry* **17**, 888–904 (1996).

# A Formal Proof of PAC Learnability for Decision Stumps

*Dedicated to Olivier Danvy on the occasion of his sixtieth birthday.*

Joseph Tassarotti  
tassarot@bc.edu  
Boston College  
USA

Anindya Banerjee  
anindya.banerjee@imdea.org  
IMDEA Software Institute  
Spain

Koundinya Vajjha  
kov5@pitt.edu  
University of Pittsburgh  
USA

Jean-Baptiste Tristan<sup>\*</sup>  
tristanj@bc.edu  
Boston College  
USA

## Abstract

We present a formal proof in Lean of probably approximately correct (PAC) learnability of the concept class of decision stumps. This classic result in machine learning theory derives a bound on error probabilities for a simple type of classifier. Though such a proof appears simple on paper, analytic and measure-theoretic subtleties arise when carrying it out fully formally. Our proof is structured so as to separate reasoning about deterministic properties of a learning function from proofs of measurability and analysis of probabilities.

**CCS Concepts:** • General and reference → Verification;  
• Theory of computation → Sample complexity and generalization bounds.

**Keywords:** interactive theorem proving, probably approximately correct, decision stumps

## ACM Reference Format:

Joseph Tassarotti, Koundinya Vajjha, Anindya Banerjee, and Jean-Baptiste Tristan. 2021. A Formal Proof of PAC Learnability for Decision Stumps. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3437992.3439917>

<sup>\*</sup>Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. CPP '21, January 18–19, 2021, Virtual, Denmark

© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8299-1/21/01...\$15.00  
<https://doi.org/10.1145/3437992.3439917>

## 1 Introduction

Machine learning (ML) has achieved remarkable success in a number of problem domains. However, the often opaque nature of ML has led to concerns about its use in important contexts such as medical diagnosis and fraud detection. To address these concerns, researchers have developed a number of algorithms with proved guarantees of robustness, privacy, fairness, and accuracy.

One essential property for an ML algorithm is to *generalize* well to unseen data. That is, after an algorithm has been trained on some data, it should be possible to present it with new data and expect it to classify or analyze it correctly. Valiant introduced the framework of Probably Approximately Correct (PAC) learnability [39], which gives a mathematical characterization of what it means for an algorithm to generalize well. This framework has become an essential part of the study of computational and statistical learning theory, and a large body of theoretical results has been developed for proving that an algorithm generalizes.

However, at present, the vast majority of these and other proofs in ML theory are pencil-and-paper arguments about idealized versions of algorithms. There is considerable room for error when real systems are built based on these algorithms. Such errors can go unnoticed for long periods of time, and are often difficult to diagnose with testing, given the randomized behavior of ML systems. Moreover, the original pencil-and-paper proofs of correctness may have errors. Because machine learning algorithms often involve randomized sampling of continuous data, their formal analysis usually requires measure-theoretic reasoning, which is technically subtle.

Formal verification offers a way to eliminate bugs from analyses of algorithms and close the gap between theory and implementation. However, the mathematical subtleties that complicate rigorous pencil-and-paper reasoning about ML algorithms also pose a serious obstacle to verification. In particular, while there has been great progress in recent

years in formal proofs about randomized programs, this work has often been restricted to *discrete* probability theory. In contrast, machine learning algorithms make heavy use of both discrete and continuous data, a mixture that requires measure-theoretic probability.

Thus, to even begin formally verifying ML algorithms in a theorem prover, results from measure theory must be formalized first. In recent years, some libraries of measure-theoretic results have been developed in various theorem provers [1, 20, 27, 36, 38]. However, it can be challenging to tell which results needed for ML algorithms are missing from these libraries. The difficulty is that, on the one hand, standard textbooks on the theoretical foundations of machine learning [28, 29, 35] omit practically all measure-theoretic details. Meanwhile, research monographs that give completely rigorous accounts [16] present results in maximal generality, well beyond what appears to be needed for many ML applications. This generality comes at the cost of mathematical prerequisites that go beyond even a first or second course in measure theory.

This paper describes the formal verification in Lean [15] of a standard result of theoretical machine learning which illustrates the complexities of measure-theoretic probability. We consider a simple type of classifier called a *decision stump*. A decision stump classifies real-numbered values into two groups with a simple rule: all values  $\leq$  some threshold  $t$  are labeled 1, and those above  $t$  are labeled 0. We show that decision stumps are PAC learnable by proving a *generalization* bound, which bounds the number of training examples needed to obtain a chosen level of classification accuracy with high probability.

We describe more precisely below how decision stumps are trained and the bound that we have proved (Section 2). Although decision stumps appear simple, they are worth considering because they are the 1-dimensional version of a classifying algorithm for axis-aligned rectangles that is used as a motivating example in all of the standard textbooks in this field [28, 29, 35].

In spite of the seeming simplicity of this example, all three of the cited textbooks either give an incorrect proof of this result or omit what we found to be the most technically challenging part of the proof (Section 3). In noting this, we do not wish to exaggerate the importance of these errors. The proofs can be fixed, and in each book, the results are correctly re-proven later as consequences of general theorems. Rather our point is to emphasize that even basic results in this area can touch on subtle issues, and errors can evade notice despite much review and scrutiny by a wide audience. We believe this further motivates the need for machine-checked proofs of such results if they are to be deployed in high-importance settings.

A key component of our work is that we structure our formal proof in a manner that lets us separate the high-level reasoning found in textbook descriptions from the low-level

details about measurability. We outline the structure used to achieve this separation of concerns in Section 4. We then describe some preliminaries about measure-theoretic probability in Section 5. The Giry monad [18] allows us to give a precise description of the sources of randomness involved in training and evaluating the performance of classifiers (Section 6). We exploit this to split deterministic reasoning about basic properties of the stump learning algorithm (Section 7) from proofs of measurability (Section 8) and the analysis of bounds on probabilities (Section 9).

The proof is publicly available at <https://github.com/jtristan/stump-learnable>.

## 2 Decision Stumps and PAC Learnability

To motivate decision stumps and the result that we have formalized, consider the following scenario. Suppose a scientist has developed a test to measure levels of some protein in blood in order to diagnose a disease. Assume there is some (unknown) threshold  $t \in \mathbb{R}$  such that if the protein level is  $\leq t$ , then the patient has the disease, and otherwise does not. Given a random sample of patients whose disease status is known, the scientist wants to estimate the threshold  $t$  so that the test can be used to screen and diagnose future patients whose disease statuses are unknown.

In other words, the scientist wants to find a decision stump to classify whether patients have the disease or not. We can model the blood test as returning a nonnegative real number. There is some distribution  $\mu$  on the interval  $[0, \infty)$  representing levels of the protein in the population. The scientist has samples  $x_1, \dots, x_n \in [0, \infty)$  independently drawn from  $\mu$  giving the results of the test on a collection of  $n$  patients, along with labels  $y_1, \dots, y_n \in \{0, 1\}$  giving each patient's disease status. A label of 1 means a patient has the disease, while 0 means they do not, so that  $y_i = 1$  if and only if  $x_i \leq t$ . The  $(x_1, y_1), \dots, (x_n, y_n)$  are called *training examples*. The scientist is trying to pick some value  $\hat{t}$  as an estimate of  $t$  to use to classify future patients. In particular, she will use her estimate to define a decision stump classifier. To state this formally, we first define a function *LABEL* that assigns a label to a point  $x$  according to a threshold  $d$ :

$$\text{LABEL}(d, x) = \begin{cases} 1 & \text{if } x \leq d \\ 0 & \text{if } x > d \end{cases} \quad (1)$$

The scientist will pick some threshold  $\hat{t}$  and then use the classifier  $\lambda x. \text{LABEL}(\hat{t}, x)$  to label future patients. We call such a classifier a *hypothesis*.

How should the scientist select  $\hat{t}$ ? One idea is to take  $\hat{t}$  to be the maximum of the  $x_i$  that have label 1. (If no  $x_i$  has label 1, she can take  $\hat{t}$  to be 0.) This estimate, at least, would correctly label all of the training examples. This corresponds to the following *learning algorithm*  $\mathcal{A}$ , which returns a classifier

using this estimate:

$$\begin{aligned} \mathcal{A} &([ (x_1, y_1), \dots, (x_n, y_n) ]) \\ &= \text{let } \hat{t} = \max\{x_i \mid y_i = 1\} \text{ in} \\ &\quad \lambda x. \text{LABEL}(\hat{t}, x) \end{aligned} \tag{2}$$

where  $\max$  of the empty set is defined to be 0.<sup>1</sup> Of course, the estimate  $\hat{t}$  used in the classifier returned by this algorithm is not going to be exactly the value of  $t$ , especially if the number of training examples,  $n$ , is small. But if  $n$  is large enough, we might hope that a good estimate can be produced. The key question then becomes, how many training examples should the scientist use?

To answer this more precisely, we need to decide how to evaluate the quality of the classifier returned by  $\mathcal{A}$ . At first, one might think that the goal should be to minimize  $|\hat{t} - t|$ , that is, to get an estimate that is as close as possible to the true threshold  $t$ . While minimizing the distance between  $\hat{t}$  and  $t$  is useful, our primary concern should be how well we classify future examples. The **ERROR** of a classifier  $h$  is the probability that  $h$  mislabels a test example  $x$  randomly sampled from  $\mu$ :

$$\text{ERROR}(h) = \Pr_{x \sim \mu} (h(x) \neq \text{LABEL}(t, x)) \tag{3}$$

We write  $x \sim \mu$  in the above to indicate that the random variable  $x$  has distribution  $\mu$ . This  $x$  is independent of the training examples used by the scientist. While the definition of **ERROR** refers to this randomized scenario of drawing a test sample, for a fixed hypothesis  $h$  the quantity **ERROR**( $h$ ) is a real number.

Because the training examples the scientist uses are randomly selected, the **ERROR** of the hypothesis she selects using  $\mathcal{A}$  is a random variable. In practice, the scientist does not know either the distribution  $\mu$  or the target  $t$ , so she cannot compute the exact **ERROR** of the classifier she obtains. Nevertheless, she might want to try to ensure that the **ERROR** of the classifier she defines using  $\mathcal{A}$  will be below some small  $\epsilon$  with high probability.

To talk about the **ERROR** of the selected hypothesis precisely, let us first introduce a helper function which takes an unlabeled list of examples and returns a list where each example has been paired with its true label:

$$\begin{aligned} \text{LLIST} &([x_1, \dots, x_n]) \\ &= \text{MAP } (\lambda x. (x, \text{LABEL}(t, x))) [x_1, \dots, x_n] \end{aligned} \tag{4}$$

Then through her choice of  $n$ , the scientist can bound the following probability:

$$\Pr_{(x_1, \dots, x_n) \sim \mu^n} (\text{ERROR}(\mathcal{A}(\text{LLIST}([x_1, \dots, x_n]))) \leq \epsilon)$$

<sup>1</sup>We call this function an *algorithm* here, following Shalev-Shwartz and Ben-David [35], although the operations on real numbers involved are non-computable.

where  $(x_1, \dots, x_n) \sim \mu^n$  indicates that the variables  $x_i$  are drawn independently from the same distribution  $\mu$ . The following theorem, which is the central result that we have formalized, tells the scientist how to select  $n$  to achieve a desired bound on this probability:

**Theorem 2.1.** For all  $\epsilon$  and  $\delta$  in the open interval  $(0, 1)$ , if  $n \geq \frac{\ln(\delta)}{\ln(1-\epsilon)} - 1$  then

$$\Pr_{(x_1, \dots, x_n) \sim \mu^n} (\text{ERROR}(\mathcal{A}(\text{LLIST}([x_1, \dots, x_n]))) \leq \epsilon) \geq 1 - \delta \tag{5}$$

Before giving an informal sketch of how this theorem is proved, we briefly describe how this result fits into the framework of PAC learnability [39].

**PAC Learnability.** PAC learning theory gives an abstract way to explain scenarios like the one with the scientist described above. In this general set up, there is some set  $\mathcal{X}$  of possible examples and a set  $C$  of hypotheses  $h : \mathcal{X} \rightarrow \{0, 1\}$ , which are possible classifiers we might select. The set  $C$  is also called a *concept class*. In the case of decision stumps,  $\mathcal{X} = \mathbb{R}^{\geq 0}$ , and  $C = \{\lambda x. \text{LABEL}(d, x) \mid d \in \mathbb{R}^{\geq 0}\}$ .

As in the stump example, there is assumed to be some unknown distribution  $\mu$  over  $\mathcal{X}$ . Additionally, there is some function  $f : \mathcal{X} \rightarrow \{0, 1\}$  that maps examples to their true labels. The **ERROR** of a hypothesis is the probability that it incorrectly labels an example drawn according to  $\mu$ . The function  $f$  is said to be *realizable* if there is some hypothesis  $h \in C$  that has error 0. The goal is to select a hypothesis with minimal **ERROR** when given a collection of training examples that have been labeled according to  $f$ . A concept class is said to be PAC learnable if there is some algorithm to select a hypothesis for which we can compute the number of training examples needed to achieve error bounds with high probability as in the stump example:

**Definition 2.2.** A concept class  $C$  is PAC learnable if there exists an algorithm  $\mathcal{A} : \text{List}(\mathcal{X} \times \{0, 1\}) \rightarrow C$  and a function  $g : (0, 1)^2 \rightarrow \mathbb{N}$  such that for all distributions  $\mu$  on  $\mathcal{X}$  and realizable label functions  $f$ , when  $\mathcal{A}$  is run on a list of at least  $g(\epsilon, \delta)$  independently sampled examples from  $\mu$  with labels computed by  $f$ , the returned hypothesis  $h$  has error  $\leq \epsilon$  with probability  $\geq 1 - \delta$ .

By “there exists” in the above definition, one typically conveys a constructive sense: one can exhibit the algorithm and compute  $g$ .<sup>2</sup> The function  $g$  is a bound on the *sample complexity* of the algorithm, telling us how many training examples are needed to achieve a bound on the **ERROR** with a given probability. There is a large body of theoretical results for showing that a concept class is PAC learnable and for bounding the sample complexity by analyzing the VC-dimension [41] of the concept class.

<sup>2</sup>Some authors require further that the algorithm  $\mathcal{A}$  has polynomial running time, but we will not do so.

**Theorem 2.1** states that the concept class of decision stumps is PAC learnable. We next turn to how this theorem is proved.

### 3 Informal Proof of PAC Learnability

The PAC learnability of decision stumps follows from the general VC-dimension theory alluded to above. However, mechanizing that underlying theory in all its generality is beyond the scope of this paper.

Instead this paper formalizes a more elementary proof, based on a description from three textbooks [28, 29, 35]. The proofs in the textbooks are for a slightly different result—learning a 2-dimensional rectangle instead of a decision stump—but the idea is essentially the same. We first sketch how the proof is presented in two of the textbooks [28, 35]. As we shall see, this argument has a flaw.

*Proof sketch of Theorem 2.1.* Given  $\mu$  and  $\epsilon$ , consider labeled samples  $(x_1, y_1), \dots, (x_n, y_n)$ . Recall that the true threshold is some unknown value  $t$ , and the learning algorithm  $\mathcal{A}$  here returns a classifier that uses the maximum of the positively labeled examples as the decision threshold  $\hat{t}$ .

We start by noting some deterministic properties about this classifier. Observe that  $\mathcal{A}$  ensures  $\hat{t} \leq t$ , because all positively labeled training examples must be  $\leq t$ . Furthermore, a test example  $x$  is misclassified only if  $\hat{t} < x \leq t$ . Thus the only errors that the classifier selected by  $\mathcal{A}$  can make is by incorrectly assigning the label 0 to an example that should have label 1.

With this in mind, the proof goes by cases on the probability that a randomly sampled example  $x \sim \mu$  will have true label 1. First assume  $\Pr_{x \sim \mu}(x \leq t) \leq \epsilon$ . That is, this case assumes that examples with true label 1 are rare. Then the classifier returned by  $\mathcal{A}$  must have **ERROR** that is  $\leq \epsilon$ . In other words, if the returned classifier can only misclassify examples whose true label is 1, and those are sufficiently rare, i.e., have probability  $\leq \epsilon$  by the above assumption, then the classifier has the desired error bound.

Next assume  $\Pr_{x \sim \mu}(x \leq t) > \epsilon$ . The idea for this case is to find an interval  $\mathcal{I}$  such that, so long as at least one of the training examples  $x_i$  falls into  $\mathcal{I}$ , the classifier returned by  $\mathcal{A}$  will have error  $\leq \epsilon$ . Then we find a bound on the probability that *none* of the  $x_i$  fall into  $\mathcal{I}$ .

In particular, set  $\mathcal{I} = [\theta, t]$ , choosing  $\theta$  so that

$$\Pr_{x \sim \mu}(x \in \mathcal{I}) = \epsilon$$

That is, we want  $\mathcal{I}$  to enclose exactly probability  $\epsilon$  under  $\mu$ . Let  $E$  be the event that at least one of the training examples falls in  $\mathcal{I}$ . If  $E$  occurs, then the threshold  $\hat{t}$  selected by  $\mathcal{A}$  is in  $\mathcal{I}$ . To see this, observe that if for some  $x_i$  we have  $\theta \leq x_i \leq t$ , then we know  $y_i = 1$ , and hence  $\theta \leq x_i \leq \hat{t} \leq t$ .

In that case, for a test example  $x$  to be misclassified, we must have  $\hat{t} < x \leq t$ , meaning  $x$  must also lie in  $\mathcal{I}$ . Thus, the event of misclassifying  $x$  is a subset of the event that  $x$  lies in  $\mathcal{I}$ . Hence, if  $E$  occurs, the probability of misclassifying  $x$  is at

most the probability that  $x$  lies in  $\mathcal{I}$ . But the probability that  $x$  lies in  $\mathcal{I}$  is  $\epsilon$  by the way we defined  $\theta$ . Therefore if  $E$  occurs, the probability that a randomly selected example  $x$  will be misclassified is  $\leq \epsilon$ , meaning the **ERROR** of the classifier will be  $\leq \epsilon$ .

So for the error to be above  $\epsilon$  means that *none* of our training examples  $x_i$  came from  $\mathcal{I}$ . For each  $i$ , we have

$$\Pr_{(x_1, \dots, x_n)}(x_i \notin \mathcal{I}) = 1 - \epsilon$$

Because each  $x_i$  is sampled independently from  $\mu$ , the probability that none of the  $x_i$  lie in  $\mathcal{I}$  is  $(1 - \epsilon)^n$ . Thus the probability of  $E$ , the event that *at least* one  $x_i$  is in  $\mathcal{I}$ , is  $1 - (1 - \epsilon)^n$ . Since we have shown that if  $E$  occurs, then the **ERROR** is  $\leq \epsilon$ , this means that the probability that the **ERROR** is  $\leq \epsilon$  is *at least* the probability of  $E$ . The rest of the proof follows by choosing  $n$  to ensure  $1 - (1 - \epsilon)^n \geq 1 - \delta$ .  $\square$

The careful reader may notice that there is one subtle step in the above: how do we choose  $\theta$  to ensure that “ $\mathcal{I}$  encloses exactly probability  $\epsilon$  under  $\mu$ ”? The phrasing “encloses exactly” comes from Kearns and Vazirani [28] (page 4), which does not say how to prove that  $\theta$  exists, beyond giving some geometric intuition in which we visualize shifting the left edge of  $\mathcal{I}$  until the enclosed amount has the specified probability. Shalev-Shwartz and Ben-David [35] similarly instructs us to select  $\theta$  so that the probability “is exactly”  $\epsilon$ .<sup>3</sup>

Unfortunately, the argument is not correct, because such a  $\theta$  may not exist.<sup>4</sup> The following counterexample demonstrates this.

**Counterexample 3.1.** Take  $\mu$  to be the Bernoulli distribution which returns 1 with probability .5 and 0 otherwise. Let  $t = .5$ , and  $\epsilon = .25$ . Then for all  $a$  we have:

$$\Pr_{x \sim \mu}(x \in [a, t]) = \begin{cases} .5 & \text{if } a \leq 0 \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

so this means that no matter how we select  $\theta$ , we cannot have  $\Pr_{x \sim \mu}(x \in [\theta, t]) = \epsilon$ , so the desired  $\theta$  does not exist.

The issue in the proof is that the distribution function  $\mu$  has been assumed to be *continuous*, whereas in the above counterexample  $\mu$  is a *discrete* distribution. In particular, if there is some point  $y$  such that  $\Pr_{x \sim \mu}(x = y) > 0$  then this introduces a *jump discontinuity* in the distribution function.

However, the statement of PAC learnability says that the error bound should be achievable for *any* distribution  $\mu$ . In order to fix the proof to work for any  $\mu$ , we need to consider

<sup>3</sup>The cited references address the more general problem of axis-aligned rectangles instead of stumps, so more specifically they describe shifting the edge of a rectangle until the enclosed probability is  $\epsilon/4$ .

<sup>4</sup>We are not the first to observe this error. The errata for the first printing of Mohri et al. [29] points out the issue in the proof of Kearns and Vazirani [28].

the following revised definition of  $\theta$ , which will ensure it exists:

$$\theta = \sup \left\{ d \in \mathcal{X} \mid \Pr_{x \sim \mu}(x \in [d, t]) \geq \epsilon \right\} \quad (7)$$

In this definition, the set (say  $S$ ) over which we are taking the supremum might be infinite. However, recall that we only need to construct the point  $\theta$  in the sub-case of the proof where we assume that  $\Pr_{x \sim \mu}(x \leq t) > \epsilon$ . This assumption implies that the supremum exists, because it means that  $S$  is nonempty, and furthermore we know that  $S$  is bounded above by  $t$ . The existence of the supremum then follows from the fact that the real numbers are Dedekind complete.

The idea behind this definition of  $\theta$  is that, if the distribution function is continuous, then the definition picks a  $\theta$  that has the property required in the erroneous proof. Instead if there is a discontinuity that causes a jump in the distribution function past the value  $\epsilon$ , then the definition selects the point at that discontinuity. In particular, we can show that with this definition

$$\Pr_{x \sim \mu}(x \in [\theta, t]) \geq \epsilon \quad (8)$$

and also that

$$\Pr_{x \sim \mu}(x \in (\theta, t]) \leq \epsilon \quad (9)$$

Note in [Equation 8](#) we have the closed interval  $[\theta, t]$ , while [Equation 9](#) is about the half-open interval  $(\theta, t]$ . This means that if  $\Pr_{x \sim \mu}(x = \theta) = 0$ , as we would have in a continuous probability distribution, then  $\Pr_{x \sim \mu}(x \in [\theta, t]) = \epsilon$ . Whereas if  $\Pr_{x \sim \mu}(x = \theta) \neq 0$ , as can occur in a discrete distribution, the probabilities of lying in  $[\theta, t]$  and  $(\theta, t]$  will differ. For example, for the discrete distribution in [Counterexample 3.1](#), the definition of  $\theta$  in [Equation 7](#) would yield  $\theta = 0$ . Observe that  $\Pr_{x \sim \mu}(x = 0) = .5 \neq 0$  while  $\Pr_{x \sim \mu}(x = v) = 0$  for any  $v \in (0, 1)$ .

The original proof of PAC learnability of the class of rectangles [\[13\]](#) did give a correct definition of  $\theta$ , as does the textbook by Mohri et al. [\[29\]](#), although neither gives a proof for why the point defined this way has the desired properties. Indeed, [Mohri et al.](#) say that it is “not hard to see” that these properties hold.

In fact, this turned out to be the most difficult part of the whole proof to formalize. While it only requires some basic results in measure theory and topology, it is nevertheless the most technical step of the argument. There were two other parts of the proof that seemed obvious on paper but turned out to be much more technically challenging than expected, having to do with showing that various functions are measurable. Often, details about measurability are elided in pencil-and-paper proofs. This is understandable because these measurability concerns can be tedious and trivial, and checking that everything is measurable can clutter an otherwise insightful proof. However, many important results in

statistical learning theory do not hold without certain measurability assumptions, as discussed by Blumer et al. [\[13\]](#) and Dudley [\[16, chapter 5\]](#).

Now that we have seen some intuition for this result and some of the pitfalls in proving it, we describe the structure of our formal proof and how it addresses these challenges.

## 4 Structure of Formal Proof

When we examine the informal proof sketched above, we can see that there are several distinct aspects of reasoning. Instead of intermingling these reasoning steps as in the proof sketch, we structure our formal proof to separate these components. As we will see, this decomposition is enabled by features of Lean. We believe that this proof structure applies more generally to other proofs of PAC learnability and related results in ML theory.

Specifically, we identify the following four components. We describe each briefly in the paragraphs below. The remaining sections of the paper then elaborate on each of these parts of the formal proof, after giving some background on measure theory in [Section 5](#).

**Specifying sources of randomness:** As we saw in [Section 2](#), there are two randomized scenarios under consideration in the statement of [Theorem 2.1](#). First, there is the randomized choice of the training examples that are given as input to  $\mathcal{A}$ . These are sampled independently and from the same distribution  $\mu$ . This first source of randomness is explicit since it appears in the statement of the probability appearing in [Equation 5](#). The second kind of randomization is in the definition of **ERROR**. Recall that we defined **ERROR** in [Equation 3](#) as the probability that an example randomly sampled from  $\mu$  would be misclassified. This random sampling is entirely separate from the sampling of the training examples, although both samplings utilize the same  $\mu$ .

Finally, the theorem statement is quantifying over the distribution  $\mu$ . As we saw in the counterexample to the informal proof, inadvertently considering only certain classes of distributions (such as continuous ones) leads to erroneous arguments.

All of these details must be represented formally in the theorem prover. To handle these issues, we make use of the Giry monad [\[18\]](#) which allows us to represent sampling from distributions as monadic computations. [Section 6](#) explains how this provides a convenient way to model the training and testing of a learning algorithm in order to formally state [Theorem 2.1](#).

**Deterministic properties of the algorithm:** In the beginning of the proof sketch of [Theorem 2.1](#) we started by noting certain *deterministic* properties of the learning algorithm  $\mathcal{A}$ , such as the fact that the threshold value  $\hat{t}$  in the classifier returned by  $\mathcal{A}$  must be  $\leq$  the true unknown threshold  $t$ . These deterministic properties were the only details

about  $\mathcal{A}$  upon which we relied when later establishing the ERROR bound. This means that an analogue of [Theorem 2.1](#) will hold for any other stump learning algorithm with those properties.

As we will see, the Giry monad enables us to encode  $\mathcal{A}$  as a purely functional Lean term that selects the maximum of the positively labeled training examples. This means we can prove these preliminary deterministic properties in the usual way one reasons about pure functions in Lean. The Lean statements of these deterministic properties are described in [Section 7](#).

**Measurability of maps and events:** One detail missing from the informal proof was any consideration of *measurability* of functions and events. In measure-theoretic probability, probability spaces are equipped with a collection of *measurable sets*. We can only speak of the probability of an event if we show that the set corresponding to the event is *measurable*, meaning that it belongs to this collection. Similarly, random variables, such as the learning algorithm  $\mathcal{A}$  itself, must be *measurable functions*.

While these facts are necessary for a rigorous proof, they risk cluttering a formal proof and obscuring all of the intuition that the informal proof gave. However, with Lean's typeclass mechanism and other proof automation, we can mostly separate the parts of the proof concerning measurability from the rest of the argument, as we describe in [Section 8](#).

**Quantitative reasoning about probabilities:** The last step of the proof involves constructing the point  $\theta$  described above and showing bounds on the probability that a sampled example lies in the interval  $[\theta, t]$ . Other than correcting the issue involved in the definition of  $\theta$ , this stage of reasoning is similar to the proof style found in informal accounts of this result. Our goal is that this portion of the proof should resemble the kind of probabilistic reasoning that is familiar to experts in ML theory. This portion of the argument, and the final proof of [Theorem 2.1](#) are described in [Section 9](#).

## 5 Preliminaries

In this section we describe some basic background on measure-theoretic probability and how measure theory has been formalized in Lean as part of the `mathlib` library [38].

**Measure theory.** The starting point for probability theory is a set  $\Omega$  called a *sample space*. Elements of  $\Omega$  are called outcomes, and represent possible results of some randomized situation. For example, if the randomized situation is the roll of a six-sided die, we would have  $\Omega = \{1, 2, 3, 4, 5, 6\}$ . In naive probability theory, subsets of  $\Omega$  are called events, and a probability function  $P$  on  $\Omega$  is a function mapping events to real numbers in the interval  $[0, 1]$ , satisfying some axioms. While this naive approach works so long as  $\Omega$  is a finite or countable set, attempting to assign probabilities to *all*

subsets of  $\Omega$  runs into technical difficulties when  $\Omega$  is an uncountable set such as  $\mathbb{R}$ .

Measure-theoretic probability theory resolves this issue by only assigning probabilities to a collection  $\mathcal{F}$  of subsets of  $\Omega$ . The elements of  $\mathcal{F}$  are called *measurable sets*. This collection  $\mathcal{F}$  must be a *sigma-algebra*, which means that it must be closed under certain operations (e.g. taking countable unions). We call the pair  $(\Omega, \mathcal{F})$  a *measurable space*. A probability measure  $\mu$  is then a function of type  $\mathcal{F} \rightarrow [0, 1]$  satisfying the following axioms:

- $\mu(\emptyset) = 0$
- $\mu(\Omega) = 1$
- If  $A_1, A_2, \dots$  is a countable collection of measurable sets such that  $A_i \cap A_j = \emptyset$  for  $i \neq j$ , then

$$\mu\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \mu(A_i)$$

For the reader familiar with topology, the notion of a measurable space is analogous to the situation in topology, where a topological space is a pair  $(X, \mathcal{V})$  where  $X$  is a set and  $\mathcal{V}$  is a collection of subsets of  $X$  called the open sets, and  $\mathcal{V}$  must be closed under various set operations. Indeed, for every topological space there is a minimal sigma-algebra containing all open sets, which is called the *Borel sigma-algebra*. We use the Borel sigma-algebra on the real numbers throughout the following.

A function  $f : (\Omega_1, \mathcal{F}_1) \rightarrow (\Omega_2, \mathcal{F}_2)$  between two measurable spaces is said to be a *measurable function* if, for all  $A \in \mathcal{F}_2$ , we have  $f^{-1}(A) \in \mathcal{F}_1$ . Again, there is an analogy to topology, where a function between topological spaces is continuous if inverse images of open sets are open. In fact, if  $f$  is a continuous function between two topological spaces, then  $f$  is measurable when those spaces are equipped with their Borel sigma-algebras. Continuity implies that many standard arithmetic operations on the reals are measurable. Other examples which are measurable but not continuous include functions for testing whether a real number is  $=, \leq$  or  $\geq$  to some value.

The general measure theory in `mathlib` allows the measures of events to be greater than 1. To obtain just probability measures, we restrict these general definitions to require that if  $\mu$  is a probability measure on  $X$ , then  $\mu(X) = 1$ . In [Section 2](#) we subscripted the `Pr` notation to indicate the distributions that we were considering. In the context of a theorem prover, which obliges us to be precise in this manner, we forgo the `Pr` notation altogether. Instead, for the probability of an event  $E$  with respect to some measure  $\mu$ , we simply write  $\mu(E)$ .

Above we have used the traditional mathematical notation of writing  $f(x)$  for the application of a function  $f$  to an argument  $x$ . However, to more closely match notation in Lean, the sequel uses  $f x$  when referring to definitions from our Lean development.

**Typeclasses.** In mathematical writing, we often associate a particular mathematical structure, such as a topology or sigma-algebra with a given set, with the convention that the structure should be used throughout. For example, when talking about continuous functions from  $\mathbb{R} \rightarrow \mathbb{R}$ , we do not constantly clarify that we mean continuous functions with respect to the topology generated by the Euclidean metric on  $\mathbb{R}$ .

This style of mathematical writing can be mimicked with Lean's typeclasses. After defining a typeclass, the user can declare instances of that typeclass, which associate a default structure with a given type. This mechanism is used throughout `mathlib` to supply default topologies, ring structures, and so on with particular types.

For example, the commands below first introduce the notation `H` to refer to the type `nnreal` of nonnegative real numbers from `mathlib`. We will use this notation when referring to the type of training examples and thresholds used for classification. After declaring this notation, an instance of `measurable_space` is defined on this type:

```
notation `H` := nnreal
instance meas_H: measurable_space H := ...
```

where we have omitted the definition after the `:=` sign. After this instance is declared, any time we refer to `H` in a context where we need a sigma-algebra, this instance will be used.

The `mathlib` library comes with lemmas to automatically derive instances of `measurable_space` from other instances. For example, if a type has been associated with a topology, we can automatically derive the Borel sigma-algebra as an instance of `measurable_space` for that type. We use this Borel sigma-algebra on `H` above. Similarly, we can derive a product sigma-algebra on the product `A × B` of two types from existing instances for `A` and `B`, as in the following example:

```
instance meas_lbl: measurable_space (H × bool)
```

**Measurable spaces for stump training.** At this point, considerations about the sigma-algebras with which a type is equipped introduce the first discrepancy between the informal set-up in [Section 2](#) and our formalization. As we described there, it is common to treat the learning algorithm as if it returned a *function* of type  $\mathbb{H} \rightarrow \{0, 1\}$ , mapping examples to labels. Since one wants to speak about probabilities involving these classifiers, this means the type of classifiers must be equipped with a sigma-algebra. What sigma-algebra should be chosen? While there are canonical choices for types that have a topology (the Borel sigma-algebra) and for various operations on spaces such as products, there is no such standard choice for function types. In particular, the category of measurable spaces is not Cartesian closed [4]. Hence, there is no generic sigma-algebra on function types that would also make evaluation measurable. The textbook by [Shalev-Shwartz and Ben-David](#) points out that the PAC

learnability framework requires the existence of a sigma-algebra on the class of hypotheses that makes classification measurable [35, Remark 3.1], but the *construction* of this sigma-algebra is not typically explained in examples.

Fortunately, the subset of decision stump classifiers has a simpler structure than the type of *all* functions from  $\mathbb{H} \rightarrow \{0, 1\}$ . In particular, the behavior of a decision stump classifier is entirely determined by the threshold used as a cut-off when assigning labels. These thresholds have type `H`, which is equipped with the Borel sigma-algebra. In particular, given a threshold `t`, the function  $\lambda x. \text{LABEL}(t, x)$  is measurable, and this is the evaluation function for a decision stump classifier. Thus, as we will see in the next section, we formalize the learning algorithm  $\mathcal{A}$  such that it directly returns a threshold instead of a classifier. Similarly we adjust the definitions of `ERROR` (and associated functions) to take a threshold as input instead of a classifier.

A similar concern arises with how we represent the collection of training examples passed to the learning algorithm  $\mathcal{A}$ . In the earlier informal presentation,  $\mathcal{A}$  takes a list of labeled examples as input. However, the construction of a sigma-algebra on variable-length lists is not commonly discussed in measure theory texts. We therefore work with dependently typed vectors of a specified length. Given a type `A` and natural `n`, the type `vec A n` represents vectors of size `n+1` of values of type `A`. When `A` is a topological space, `vec A n` can be given the  $(n + 1)$ -ary product topology, and we can then make it into a measurable space by equipping it with the Borel sigma-algebra.

## 6 Specifying Randomized Processes with the Giry Monad

Now that we have described some of the preliminaries of measure-theoretic probability, we turn to the question of how to formally represent the learning algorithm in the theorem prover.

In traditional probability theory, it is common to fix some sample space  $\Omega$  and then work with a collection of *random variables* on this sample space. If  $V$  is a measurable space, a  $V$ -valued random variable is a measurable function of type  $\Omega \rightarrow V$ . One can think of the elements of the sample space  $\Omega$  as some underlying source of randomness, and then the random variables encode how that randomness is transformed into an observable value. For example,  $\Omega$  could be a sequence of random coin flips, and a random variable  $f$  might be a randomized algorithm that uses those coin flips.

In fact, at a certain point most treatments of probability theory start to leave the sample space  $\Omega$  completely abstract. One simply postulates the existence of some  $\Omega$  on which a collection of random variables with various distributions are said to exist. To ensure that the resulting theory is not vacuous, a theorem is proven to show that there exists an  $\Omega$

and a measure  $\mu$  on  $\Omega$  for which a suitably rich collection of random variables can be constructed.

While this pencil-and-paper approach could be used in formalization, it is inconvenient in several ways. First, while random variables are formally functions on the sample space, in practice we often treat them as if they were elements of their codomain. For example if  $X$  and  $Y$  are two real-valued random variables, then one writes  $X + Y$  to mean the random variable  $\lambda\omega. X(\omega) + Y(\omega)$ . Similarly, if  $f : \mathbb{R} \rightarrow \mathbb{R}$ , we write  $f(X)$  for the random variable  $(\lambda\omega. f(X(\omega)))$ . While this kind of convention is well understood on paper, trying to overload notations in a theorem prover to support it seems difficult.

The Giry monad [18] solves this problem by providing a syntactic sugar to describe stochastic procedures concisely.

## 6.1 Definition of the Giry Monad

The Giry monad is a triple  $(\text{Meas}(\cdot), \text{bind}, \text{ret})$ . For any measurable space  $X$ ,  $\text{Meas}(X)$  is the space of probability measures over  $X$ , that is, functions from measurable subsets of  $X$  to  $[0, 1]$  that satisfy the additional axioms of probability measures. The function  $\text{bind}$  is of type  $\text{Meas}(X) \rightarrow (X \rightarrow \text{Meas}(Y)) \rightarrow \text{Meas}(Y)$ . That is, it takes a probability measure on  $X$ , a function that transforms values from  $X$  into probability measures over  $Y$ , and returns a probability measure on  $Y$ . The return function  $\text{ret}$  is of type  $X \rightarrow \text{Meas}(X)$ . It takes a value from  $X$  and returns a probability measure on  $X$ . The `mathlib` library defines this monad for general measures, which we then restrict to probability measures.

Functions  $\text{bind}$  and  $\text{ret}$  construct probability measures, so their definitions say what probability they assign to an event. Letting  $A$  be an event we define:

$$\text{bind } \mu f A = \int_{x \in X} f(x)(A) d\mu \quad (10)$$

$$\text{ret } x A = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

While we give the definitions here using standard mathematical notation, the formalization in `mathlib` uses the Lean definition of the integral. Here,  $\text{ret}(x)$  is the distribution that always returns  $x$  with probability 1. The definition of  $\text{bind } \mu f$  corresponds to first sampling from  $\mu$  to obtain some value  $x$ , and then continuing with the probability measure  $f(x)$ . Then,  $\text{bind}$  and  $\text{ret}$  satisfy the usual monad laws

$$\text{bind } (\text{ret } x) f = \text{ret } (f x) \quad (12)$$

$$\text{bind } \mu (\lambda x. \text{ret } x) = \mu \quad (13)$$

$$\text{bind } (\text{bind } \mu f) g = \text{bind } \mu (\lambda x. \text{bind } (f x) g) \quad (14)$$

However, these laws only hold when  $f$  and  $g$  are measurable functions. We will use the usual `do`-notation, writing `do x ← μ; g(x)` for  $\text{bind } \mu g$ .

As with any monad, we can define a function  $\text{map}$  which lifts a function  $f : A \rightarrow B$ , to a function of type  $\text{Meas}(A) \rightarrow \text{Meas}(B)$ . Given  $\mu : \text{Meas}(A)$ , we interpret  $\text{map } f \mu$  as the

probability distribution that first samples from  $\mu$  to obtain a value of type  $A$ , and then applies  $f$  to it. Concretely, this is defined in terms of `bind` and `ret` as:

$$\text{map } f \mu = \text{do } x \leftarrow \mu; \text{ret } (f x) \quad (15)$$

As an example, we show how to construct a distribution that samples  $n$  independent times from a distribution  $\mu$  and returns the result as a tuple. That is, we formally define the  $\mu^n$  distribution that we used in Section 2. Let  $(X, \mathcal{F})$  be a measurable space and  $(X^n, \mathcal{F}^n)$  be the measurable space where  $X^n$  is the cartesian product of  $X$  ( $n$  times) and  $\mathcal{F}^n$  is the product of  $\mathcal{F}$  ( $n$  times). Let  $\mu$  be a probability measure on  $(X, \mathcal{F})$ . Then, we define the measure  $\mu^n$  recursively on  $n$  as

$$\mu^1 = \mu \quad (16)$$

$$\mu^n = \text{do } v \leftarrow \mu^{n-1}; \text{do } s \leftarrow \mu; \text{ret } (s, v) \quad (17)$$

Instead of nested tuples, as in the above, or lists of training examples, as we did in Section 2, we will use dependently typed vectors in Lean. The following term (definition omitted) gives the probability measure corresponding to taking  $n+1$  independent samples from a distribution and assembling them in a vector:

```
def vec.prob_measure
  (n : ℕ) (μ : probability_measure A)
  : probability_measure (vec A n) := ...
```

## 6.2 Modeling Stump Training and Testing

Let us now see how the decision stump training and testing procedures can be described using the Giry monad. Lean has a sectioning mechanism and a way to declare local variables. In our formalization, the lines below declare probability measures over the class of examples, and an arbitrary target threshold value for labeling samples:

```
variables (μ: probability_measure ℙ) (target: ℙ)
```

When we write definitions that use these variables, Lean will interpret the definition to treat these variables as if they were additional parameters on which the function depends.<sup>5</sup>

The following pure function labels a sample according to the target.

```
def label (target: ℙ): ℙ → ℙ × bool :=
  λ x: ℙ, (x, rle x target)
```

where `rle x target` returns true if  $x \leq \text{target}$  and false otherwise.

Finally, we can define the event that an example is misclassified, and the `ERROR` function:

```
def error_set (h: ℙ) :=
  {x: ℙ | label h x ≠ label target x}

def error := λ h, μ (error_set h target)
```

<sup>5</sup>This is different from the behavior in Coq, where a definition that depends on section variables is only generalized to take additional arguments *outside* the section where the variables are declared.

The learning function  $\mathcal{A}$  will take a vector of labeled examples as input and output a hypothesis. The function `vec_map` takes a function as an argument and applies it pointwise to the elements of the vector. We use this to label the inputs to the learning function:

```
def label_sample := vec_map (label target)
```

Our learning algorithm starts by transforming any negative example to 0 and then stripping off the labels, with the following function:

```
def filter :=
  vec_map (λ p, if p.snd then p.fst else 0)
```

This is safe since, if there were no positive examples, the learning algorithm should return 0 as we described in [Section 2](#). Finally, we can define the learning function  $\mathcal{A}$  that we will use. We call this function `choose` in the formalization. It selects the largest example after the above filtering process:

```
def choose (n: ℕ):vec (ℍ × bool) n → ℍ :=
```

```
λ data: (vec (ℍ × bool) n), max n (filter n data)
```

We then use the Giry monad to describe the measure on classifiers that results from running the algorithm:

```
def denot: probability_measure ℍ :=
  let η := vec.prob_measure n μ in
  let ν := map (label_sample target n) η in
  let γ := map (choose n) ν in
  γ
```

Note that `map` here is the monadic map operation defined in [Equation 15](#), not `vec_map`. Thus,  $\eta$  is a distribution on training examples which is then transformed to  $\nu$ , a distribution on labeled training examples. Then  $\nu$  is transformed into a distribution  $\gamma$  on thresholds by lifting `choose`.

Finally, we have the following formal version of [Theorem 2.1](#):

`theorem choose_PAC:`

```
∀ ε: nnreal, ∀ δ: nnreal, ∀ n: ℕ,
ε > 0 → ε < 1 → δ > 0 → δ < 1 →
n > (complexity ε δ) →
(denot μ target n) {h: ℍ | error μ target h ≤ ε}
  ≥ 1 - δ
```

where `complexity` is the following function:

```
def complexity (ε: ℝ) (δ: ℝ) : ℝ :=
(log(δ) / log(1 - ε)) - (1: nat)
```

The following sections outline how we prove this theorem in Lean.

## 7 Deterministic Reasoning

The overall proof builds on two simple assumptions that must be satisfied by the learning algorithm. First, the algorithm must return an estimate that is  $\leq \text{target}$ . Formally,

```
lemma choose_property_1:
  ∀ n: ℕ, ∀ S: vec ℍ n,
  choose n (label_sample target n S) ≤ target
```

Our implementation satisfies this property since after the `filter` step in `choose`, all of the examples will have been mapped to a value  $\leq \text{target}$ , and we then select the maximum value from the vector.

Second, the algorithm must return an estimate that is greater or equal to any positive example. This is because the proof uses the assumption that no examples lie in the region between the estimate and the target. Formally,

`lemma choose_property_2:`

```
  ∀ n: ℕ, ∀ S: vec ℍ n,
  ∀ i,
  ∀ p = kth_projn (label_sample target n S) i,
  p.snd = tt →
  p.fst ≤ choose n (label_sample target n S)
```

where `kth_projn l i` is an expression giving component  $i$  of the vector  $l$ . Our implementation satisfies this property because `choose` calls `filter` which leaves positive examples unchanged, and then we select the maximum from the filtered vector.

These proofs are trivial and account for only a very small fraction of the overall proof. Yet, these are the two specific properties of the algorithm that we need.

## 8 Measurability Considerations

About a quarter of the formalization consists in proving that various sets and functions are measurable. The predicate `is_measurable S` states that the set  $S$  is a measurable set, while `measurable f` states that the function  $f$  is measurable. These proofs can be long but are generally routine, with a few notable exceptions.

We will need to divide up the sample space into various intervals. If  $a$  and  $b$  are two reals, then we write `Ioo a b`, `Ioc a b`, `Ico a b`, `Icc a b` in Lean to refer to the intervals  $(a, b)$ ,  $[a, b]$ ,  $[a, b)$ , and  $[a, b]$ , respectively. To start, we observe that the error of a hypothesis is the measure of the interval between it and the target.

`lemma error_interval_1:`

```
  ∀ h, h ≤ target →
  error μ target h = μ (Ioc h target)
```

`lemma error_interval_2:`

```
  ∀ h, h > target →
  error μ target h = μ (Ioc target h)
```

We next use these lemmas to prove that the function that computes the `ERROR` of a hypothesis is measurable:

`lemma error_measurable:`

```
  measurable (error μ target)
```

*Proof.* First, note that if  $A$  and  $B$  are measurable subsets such that  $A \subseteq B$ , then  $\mu(B \setminus A) = \mu B - \mu A$ . If  $h \leq \text{target}$ , then

$$\begin{aligned} \text{error } μ \text{ target } h &= μ(h, \text{target}) \\ &= μ[0, \text{target}] - μ[0, h] \end{aligned}$$

Likewise, if  $h > \text{target}$  then  $\text{error } \mu \text{ target } h = \mu [0, h] - \mu [0, \text{target}]$ . Subtraction is measurable and testing whether a value is  $\leq \text{target}$  is measurable. Therefore, since measurability is closed under composition, it suffices to show that the function  $\lambda x. \mu [0, x]$  is Borel measurable. Because this function is monotone, its measurability is a standard result, though this fact was missing from `mathlib`.  $\square$

Next, one must show that the learning algorithm `choose` is measurable, after fixing the number of input examples:

```
lemma choose_measurable: measurable (choose n)
```

*Proof.* To prove that `choose` is a measurable function, we must prove that `max` is a measurable function. Because `max` is continuous, it is Borel measurable.  $\square$

Although the previous proof is straightforward, it hinges on the fact that the sigma-algebra structure we associate with `vec H n` is the Borel sigma-algebra. But, because we define a vector as an iterated product, another possible sigma-algebra structure for `vec H n` is the  $n+1$ -ary product sigma-algebra.

Recall from the previous section that our development uses Lean's typeclass mechanism to automatically associate product sigma-algebras with product spaces, and Borel sigma-algebras with topological spaces. As the preceding paragraph explains, for `vec H n` there are two possible choices. Which choice should be used? In programming languages with typeclasses, the problem of having to select between two potentially different instances of a typeclass is called a *coherence* problem [31]. Because of this ambiguity, `mathlib` is careful to only enable certain instances by default. Of course, this same potential ambiguity arises in normal mathematical writing, when we omit mentioning the associated sigma-algebra.

Fortunately, in the case of `vec H n`, these two sigma-algebras happen to be the same. In general, if  $X$  and  $Y$  are topological spaces with a *countable basis*, then the Borel sigma-algebra on  $X \times Y$  is equal to the product of the Borel sigma-algebras on  $X$  and  $Y$ . The standard topology on the nonnegative reals has a countable basis, so the equivalence holds. Thus, although the proof of measurability for `max` can be simple, it uses a subtle fact that resolves the ambiguity involved in referring to sets without constantly mentioning their sigma-algebras.

## 9 Probabilistic Reasoning

The remainder of the proof involves the construction of the point  $\theta$  and explicitly bounding the probability of various events.

Recall from the informal sketch in Section 3 that we first case split on whether  $\Pr_{x \sim \mu}(x \leq \text{target}) \leq \epsilon$  or not. In the language of measures, this is equivalent to a case split on whether  $\mu [0, \text{target}] \leq \epsilon$ . In the formalization, it simplifies slightly the application of certain lemmas if we instead split on whether  $\mu (0, \text{target}] \leq \epsilon$ . The following lemma is the key property in the case where the weight between 0 and

target is  $\leq \epsilon$ . In that case, the learning algorithm always chooses a hypothesis with error at most  $\epsilon$ .

lemma always\_succeed:

$$\begin{aligned} \forall \epsilon: \text{nnreal}, \epsilon > 0 \rightarrow \forall n: \mathbb{N}, \\ \mu (\text{Ioc } 0 \text{ target}) \leq \epsilon \rightarrow \\ \forall S: \text{vec } \mathbb{H} n, \\ \text{error } \mu \text{ target} \\ (\text{choose } n (\text{label\_sample target } n S)) \\ \leq \epsilon \end{aligned}$$

*Proof.* By `error_interval_1`, we know that the error is going to be equal to the measure of the interval

$$(\text{Ioc } (\text{choose } n (\text{label\_sample target } n S)) \text{ target})$$

Because we know `choose` must return a threshold between 0 and `target`, this interval is a subset of  $(\text{Ioc } 0 \text{ target})$ . Since measures are monotone, this means the measure of that interval must be  $\leq \mu (\text{Ioc } 0 \text{ target})$ , which is  $\leq \epsilon$  by assumption.  $\square$

For the case where  $\mu (0, \text{target}] > \epsilon$ , the informal sketch selected a point  $\theta$  such that  $\mu [\theta, \text{target}] = \epsilon$ . However, as we saw in Counterexample 3.1, such a  $\theta$  may not exist when  $\mu$  is not continuous. Instead, we construct  $\theta$  so that  $\mu [\theta, \text{target}] \geq \epsilon$ , and  $\mu (\theta, \text{target}] \leq \epsilon$ . The following theorem states the existence of such a point:

theorem extend\_to\_epsilon\_1:

$$\begin{aligned} \forall \epsilon: \text{nnreal}, \epsilon > 0 \rightarrow \\ \mu (\text{Ioc } 0 \text{ target}) > \epsilon \rightarrow \\ \exists \theta: \text{nnreal}, \mu (\text{Icc } \theta \text{ target}) \geq \epsilon \wedge \\ \mu (\text{Ioc } \theta \text{ target}) \leq \epsilon \end{aligned}$$

*Proof.* We take  $\theta$  to be  $\sup\{x \in X \mid \mu [x, \text{target}] \geq \epsilon\}$ . The supremum exists because the set in question is bounded above by `target`, and the set is nonempty because it must contain 0 by our assumption that  $\mu (0, \text{target}] > \epsilon$ . To see that  $\mu [\theta, \text{target}] \geq \epsilon$ , we can construct an increasing sequence of points  $x_n \leq \theta$  such that  $\lim_{n \rightarrow \infty} x_n = \theta$ , where for each  $n$ ,  $\mu [x_n, \text{target}] \geq \epsilon$ . We have then that:

$$\bigcap_i [x_i, \text{target}] = [\theta, \text{target}]$$

We use this in conjunction with the fact that measures are continuous from above, meaning that if  $A_1, A_2, \dots$  is a sequence of measurable sets such that  $A_{i+1} \subseteq A_i$  for all  $i$ , then

$$\mu \left( \bigcap_{i=1}^{\infty} A_i \right) = \lim_{i \rightarrow \infty} \mu A_i$$

Hence we have

$$\begin{aligned} \mu [\theta, \text{target}] &= \mu \left( \bigcap_{i=1}^{\infty} [x_i, \text{target}] \right) \\ &= \lim_{n \rightarrow \infty} \mu [x_n, \text{target}] \\ &\geq \epsilon \end{aligned}$$

The proof that  $\mu(\theta, \text{target}] \leq \epsilon$  is the dual argument, using continuity from below.  $\square$

The conclusion of this theorem states two inequalities involving  $\theta$ . On the one hand, we need  $\theta$  to be small enough that we can ensure at least one *training example* will lie between  $\theta$  and target. On the other hand, we want  $\theta$  to be large enough that if we *only* misclassify *test examples* that lie between  $\theta$  and target, the error will nevertheless be at most  $\epsilon$ .

The next two lemmas formalize these properties. Recall that choose maps all negative training examples to 0, leaves positive examples unchanged, and then takes the maximum of the resulting vector. The next lemma says that given a point  $\theta$  such that  $\mu[\theta, \text{target}] \geq \epsilon$ , the measure of the event that an example gets mapped to something less than  $\theta$  is at most  $1 - \epsilon$ .

```
lemma miss_prob:
  ∀ ε, ∀ θ: nnreal, θ > 0 →
  μ(Icc θ target) ≥ ε →
  μ{x : ℙ | ∀ a b,
    (a,b) = label target x →
    (if b then a else 0) < θ} ≤ 1 - ε
```

The next lemma shows why the property  $\mu(\theta, \text{target}] \leq \epsilon$  is useful. In particular, it says that for such a  $\theta$ , in order to have an error  $> \epsilon$  on the hypothesis selected by choose, all training examples must get mapped to something less than  $\theta$ . Formally, we say that the set of training samples which would lead to an error greater than  $\epsilon$ , is a subset of those in which all the examples get mapped to a value less than  $\theta$ .

```
lemma all_missed:
  ∀ ε: nnreal,
  ∀ θ: nnreal,
  μ(Ioc θ target) ≤ ε →
  {S | error μ target
    (choose n (label_sample target n S))
    > ε} ⊆
  {S | ∀ i,
    ∀ p = label target (kth_projn S i),
    (if p.snd then p.fst else 0) < θ}
```

Finally, we prove a bound related to the complexity function, which computes the number of training examples needed:

```
lemma complexity_enough:
  ∀ ε: nnreal, ∀ δ: nnreal, ∀ n: ℕ,
  ε > (0: nnreal) → ε < (1: nnreal) →
  δ > (0: nnreal) → δ < (1: nnreal) →
  (n: ℝ) > (complexity ε δ) → ((1 - ε)^(n+1)) ≤ δ
```

Combining these lemmas together, we can finish the proof:

*Proof of choose\_PAC.* We have seen that always\_succeed handles the case  $\mu(0, \text{target}] \leq \epsilon$ . For the other case, where  $\mu(0, \text{target}] > \epsilon$ , we can apply extend\_to\_epsilon\_1 to get a  $\theta$  with the specified properties. By all\_missed we

know that the event that the hypothesis selected has error  $> \epsilon$  is a subset of the event where all the training examples get mapped to  $< \theta$ . Then, by miss\_prob we know the probability that a given example gets mapped to  $< \theta$  is  $\leq 1 - \epsilon$ . Because the training examples are selected independently, the probability that all  $n + 1$  examples get mapped to a value  $< \theta$  is at most  $(1 - \epsilon)^{n+1}$ . Applying complexity\_enough, we have that  $(1 - \epsilon)^{n+1} \leq \delta$ , so we are done.  $\square$

## 10 Related Work

Classic results about the average case behavior of quicksort and binary search trees have been formalized by a number of authors using different proof assistants [17, 37, 40]. In each case, the authors write down the algorithm to be analyzed using a variant of the monadic style we discuss in Section 6. Gopinathan and Sergey [19] verify the error rate of Bloom Filters and variants. Affeldt et al. [2] formalize results from information theory about lossy encoding. For the most part, these formalizations only use discrete probability theory, with the exception of Eberl et al.'s analysis of treaps [17], which requires general measure-theoretic probability. They report that dealing with measurability issues adds some overhead compared to pencil-and-paper reasoning, though they are able to automate many of these proofs.

Several projects have formalized results from cryptography, which also involves probabilistic reasoning [8, 9, 12, 30]. A challenge in formalizing such proofs lies in the need to establish a relation between the behavior of two different randomized algorithms, as part of the game-playing approach to cryptographic security proofs. Because cryptographic proofs generally only use discrete probability theory, these libraries do not formalize measure-theoretic results. There are many connections between cryptography and learning theory [32], which would be interesting to formalize.

There have been formalizations of measure-theoretic probability theory in a few proof assistants. Hurd [23] formalized basic measure theory in the HOL proof assistant, including a proof of Caratheodory's extension theorem. Hözl and Heller [21] developed a more substantial library in the Isabelle theorem prover, which has since been extended further. Avigad et al. [5] used this library to formalize a proof of the Central Limit Theorem. Several measure theory libraries have also been developed in Coq [1, 27, 36]. The ALEA library [3] instead uses a synthetic approach to discrete probability in Coq, a technique that has subsequently been extended to continuous probabilities by Bidlingmaier et al. [11].

More recent work has formalized theoretical machine learning results. Selsam et al. [34] use Lean to prove the correctness of an optimization procedure for stochastic computation graphs. They prove that the random gradients used in their stochastic backpropagation implementation are unbiased. In their proof, they add axioms to the system for various basic mathematical facts. They argue that even if

there are errors in these axioms that could potentially lead to inconsistency, the process of constructing formal proofs for the rest of the algorithm still helps eliminate mistakes.

Bagnall and Stewart [6] use Coq to give machine-checked proofs of bounds on generalization errors. They use Hoeffding's inequality to obtain bounds on error when the hypothesis space is finite or there is a separate test-set on which to evaluate a classifier after training. They apply this result to bound the generalization error of ReLU neural networks with quantized weights. Their proof is restricted to discrete distributions and adds some results from probability theory as axioms (Pinsker's inequality and Gibbs' inequality).

Bentkamp et al. [10] use Isabelle/HOL to formalize a result by Cohen et al. [14], which shows that deep convolutional arithmetic circuits are more expressive than shallow ones, in the sense that shallow networks must be exponentially larger in order to express the same function. Although convolutional arithmetic circuits are not widely used in practice compared to other artificial neural networks, this result is part of an effort to understand theoretically the success of deep learning. Bentkamp et al. report that they proved a stronger version of the original result, and doing so allowed them to structure the formal proof in a more modular way. The formalization was completed only 14 months after the original arXiv posting by Cohen et al., suggesting that once the right libraries are available for a theorem prover, it is feasible to mechanize state of the art results in some areas of theoretical machine learning in a relatively brief period of time.

After the development described by our paper was publicly released, Zinkevich [42] published a Lean library for probability theory and theoretical machine learning. Among other results, this library contains theorems about PAC learnability when the class of hypotheses is finite. Because the decision stump hypothesis class is the set of all nonnegative real numbers, our result is not covered by these theorems.

A related but distinct line of work applies machine learning techniques to automatically construct formal proofs of theorems. Traditional approaches to automated theorem proving rely on a mixture of heuristics and specialized algorithms for decidable sub-problems. By using a pre-existing corpus of formal proofs, supervised learning algorithms can be trained to select hypotheses and construct proofs in a formal system [7, 22, 24–26, 33].

## 11 Conclusion

We have presented a machine-checked, formal proof of PAC learnability of the concept class of decision stumps. The proof is formalized using the Lean theorem prover. We used the Giry monad to keep the formalization simple and close to a pencil-and-paper proof. To formalize this proof, we specialized the measure theory formalization of the `mathlib` library to the necessary basic probability theory. As expected,

the formalization is at times subtle when we must consider topological or measurability results, mostly to prove that the learning algorithm and `ERROR` are measurable functions. The most technical part of the proof has to do with proving the existence of an interval with the appropriate measure, a detail that standard textbook proofs either omit or get wrong.

Our work shows that the Lean prover and the `mathlib` library are mature enough to tackle a simple but classic result in statistical learning theory. A next step would be to formally prove more general results from VC-dimension theory. In addition, there exist a number of generalizations of PAC learnability, such as *agnostic* PAC learnability, which removes the assumption that some hypothesis in the class perfectly classifies the examples. Other generalizations allow for more than two classification labels and different kinds of `ERROR` functions. It would be interesting to formalize these various extensions and some related applications.

## Acknowledgments

We thank Gordon Stewart for comments on a previous draft of the paper. We thank the anonymous reviewers from the CPP'21 PC for their feedback. Some of the work described in this paper was performed while Koundinya Vajjha was an intern at Oracle Labs. Vajjha was additionally supported by the Alfred P. Sloan Foundation under grant number G-2018-10067. Banerjee's research was based on work supported by the US National Science Foundation (NSF), while working at the Foundation. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the authors and do not necessarily reflect the views of the NSF.

## References

- [1] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Kazuhiko Sakaguchi, and Pierre-Yves Strub. 2020. `math-comp` Analysis Library. <https://github.com/math-comp/analysis>.
- [2] Reynald Affeldt, Manabu Hagiwara, and Jonas Sénizergues. 2014. Formalization of Shannon's Theorems. *J. Autom. Reason.* 53, 1 (2014), 63–103.
- [3] Philippe Audebaud and Christine Paulin-Mohring. 2009. Proofs of randomized algorithms in Coq. *Sci. Comput. Program.* 74, 8 (2009), 568–589.
- [4] Robert J. Aumann. 1961. Borel structures for function spaces. *Illinois J. Math.* 5, 4 (12 1961), 614–630.
- [5] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. 2017. A Formally Verified Proof of the Central Limit Theorem. *J. Autom. Reason.* 59, 4 (2017), 389–423.
- [6] Alexander Bagnall and Gordon Stewart. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *AAAI'19: The Thirty-Third AAAI Conference on Artificial Intelligence*. 2662–2669.
- [7] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An Environment for Machine Learning of Higher Order Logic Theorem Proving. In *Thirty-sixth International Conference on Machine Learning (ICML)*. 454–463.
- [8] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 146–166.

- [9] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *POPL*. 90–101.
- [10] Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. 2019. A formal proof of the expressiveness of deep learning. *Journal of Automated Reasoning* 63, 2 (2019), 347–368.
- [11] Martin E. Bidlingmaier, Florian Faissole, and Bas Spitters. 2019. Synthetic topology in Homotopy Type Theory for probabilistic programming. *CoRR* abs/1912.07339 (2019). arXiv:1912.07339 <http://arxiv.org/abs/1912.07339>
- [12] Bruno Blanchet. 2006. A Computationally Sound Mechanized Prover for Security Protocols. In *2006 IEEE Symposium on Security and Privacy*. 140–154.
- [13] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)* 36, 4 (1989), 929–965.
- [14] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the Expressive Power of Deep Learning: A Tensor Analysis. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016*. 698–728.
- [15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *CADE-25 - 25th International Conference on Automated Deduction*. 378–388.
- [16] R. M. Dudley. 2014. *Uniform Central Limit Theorems* (2nd ed.). Cambridge University Press.
- [17] Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. 2018. Verified Analysis of Random Trees. In *ITP*. 196–214.
- [18] Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis (Lecture Notes in Mathematics, Vol. 915)*, B. Banaschewski (Ed.), 68–85.
- [19] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In *CAV*, Shuvendu K. Lahiri and Chao Wang (Eds.). 279–303.
- [20] Johannes Hözl. 2013. *Construction and stochastic applications of measure spaces in higher-order logic*. Ph.D. Dissertation. Technical University Munich.
- [21] Johannes Hözl and Armin Heller. 2011. Three Chapters of Measure Theory in Isabelle/HOL. In *ITP*. 135–151.
- [22] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A Learning Environment for Theorem Proving. In *7th International Conference on Learning Representations, ICLR 2019*.
- [23] Joe Hurd. 2003. *Formal Verification of Probabilistic Algorithms*. Ph.D. Dissertation. Cambridge University.
- [24] Jan Jakubuv and Josef Urban. 2019. Hammering Mizar by Learning Clause Guidance (Short Paper). In *ITP*. 34:1–34:8.
- [25] Cezary Kaliszyk, François Chollet, and Christian Szegedy. 2017. Hol-Step: A Machine Learning Dataset for Higher-order Logic Theorem Proving. In *5th International Conference on Learning Representations, ICLR 2017*.
- [26] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšík. 2018. Reinforcement Learning of Theorem Proving. In *NeurIPS*. 8836–8847.
- [27] Robert Kam. 2008. coq-markov Library. <https://github.com/coq-contribs/markov>.
- [28] Michael J Kearns and Umesh Virkumar Vazirani. 1994. *An Introduction to Computational Learning Theory*. MIT press.
- [29] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of Machine Learning*. MIT press.
- [30] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *POST*. 53–72.
- [31] Simon Peyton Jones, Mark Jones, and Erik Meijer. 1997. Type classes: an exploration of the design space. In *Haskell Workshop*.
- [32] Ronald L. Rivest. 1991. Cryptography and Machine Learning. In *Advances in Cryptology - ASIACRYPT '91*. 427–439.
- [33] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019*. 336–353.
- [34] Daniel Selsam, Percy Liang, and David Dill. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In *International Conference on Machine Learning (ICML)*. 3047–3056.
- [35] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- [36] Joseph Tassarotti. 2020. coq-proba Probability Library. <https://github.com/jtassarotti/coq-proba>.
- [37] Joseph Tassarotti and Robert Harper. 2018. Verified Tail Bounds for Randomized Programs. In *ITP*. 560–578.
- [38] The mathlib Community. 2020. The Lean Mathematical Library. In *CPP*. 367–381.
- [39] Leslie G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.
- [40] Eelis van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *TYPES*. 256–271.
- [41] Vladimir Naumovich Vapnik. 2000. *The Nature of Statistical Learning Theory, Second Edition*. Springer.
- [42] Martin Zinkevich. 2020. <https://github.com/google/formal-ml>