Markov Chain Algorithms: A Template for Building Future Robust Low Power Systems

Biplab Deka*, Alex A. Birklykke†*, Henry Duwe*, Vikash K. Mansinghka‡ and Rakesh Kumar*
*Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign, USA
Email: {deka2, duweiii2, rakeshk}@illinois.edu
†The Technology Platforms Section, Aalborg University, Denmark, Email: alb@es.aau.dk
‡Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, USA, Email: vkm@mit.edu

***Abstract*—Although computational systems are looking towards post CMOS devices in the pursuit of lower power, the inherent unreliability of such devices makes it difficult to design robust systems without additional power overheads for guaranteeing robustness. As such, algorithmic structures with inherent ability to tolerate computational errors are of significant interest. We propose to cast applications as stochastic algorithms based on Markov chains as such algorithms are both sufficiently general and tolerant to transition errors. We show with four example applications - boolean satisfiability (SAT), sorting, LDPC decoding and clustering - how applications can be cast as Markov Chain algorithms. Using algorithmic fault injection techniques, we demonstrate the robustness of these implementations to transition errors with high error rates. Based on these results, we make a case for using Markov Chains as an algorithmic template for future robust low power systems.**

I. Introduction

It is becoming increasingly clear that disruptive low power solutions are needed to meet future power-performance goals. Such disruptive solutions are already being explored in the context of microprocessors. Examples include approaches based on trading robustness for lower power such as better-than-worst-case (BTWC) design [1]. However, it is not clear if such approaches scale well to post CMOS technologies where hardware fault rates are going to be significantly higher [2]. As such, a radical rethinking of system design might be needed to build robust systems in the fault prone post CMOS era.

In this paper, we explore one such disruptive solution – robustifying applications by casting these as Markov chain (MC) algorithms. This approach is based on the insight that Markov chain algorithms can produce acceptable results even in the face of transition errors (Section III-B). Given the above insight, we propose to replace the rigid and sensitive control flow used in many conventional algorithms for solving deterministic problems with a Markov process that guides the execution towards the solution through a sequence of random steps. To leverage the robustness of such algorithms, we propose building systems that let faults in devices manifest themselves only as transition errors at the algorithmic level. Such an approach could form the basis of building systems that save power by using low power, possibly unreliable, devices and allow only controlled errors that the application can tolerate.

Clearly, the underlying assumption is that for a wide range of applications it is possible to find a solution using a Markov chain. In Section III-A, we present a general approach that lets a wide range of applications use Markov chains to find their solutions. In Section IV, we show how such Markov chains can be constructed for four example applications: boolean satisfiability (SAT), low density parity check (LDPC) code decoding, sorting, and clustering. In addition, we note that Markov chain implementations already exist for a wide range of problems such as the ones solved using stochastic local search strategies.

This paper makes the following contributions:

- We propose a novel algorithmic approach to building robust applications — by transforming applications into Markov chain algorithms. We propose executing such algorithms on a system that limits the manifestation of hardware faults to transition errors in the Markov chains.
- We present a methodology for constructing Markov chain algorithms for four problems — Boolean satisfiability, LDPC decoding, sorting and clustering. We argue that the methodology can be applied to a large class of applications.
- We quantify the robustness benefits of casting applications into Markov chain algorithms using algorithmic error injections. We show that Markov chain implementations are indeed significantly more robust than the traditional implementations of these applications. The Markov chain applications were able to tolerate transition errors with rates as high as 10-20%. Even at high error rates, Markov chain algorithms produced acceptable results in terms of runtime and output quality. Also, these algorithms showed gradual degradation in runtime or output quality with increasing error rates.

Our results on the robustness of Markov chains show that Markov chains may indeed be an attractive template for building future low power systems. Such systems could trade off robustness for power benefits using one of several strategies, such as (a) voltage overscaling (b) use of low power, possibly unreliable, post CMOS devices [2] (c) eliminating design gaurdbands [1], or (d) functional underdesign. Energy benefits may be further enhanced by exploiting the sampling-level parallelism that may exist in many Markov chain-based implementations.

II. Related work

The proposed approach to build robust applications by casting them into Markov chain algorithms can be thought of as a novel approach for *application robustification*. Sloan *et al* [3] propose an approach for application robustification where an application with unknown correct output x^* is cast into a minimization function $f(x)$ whose minimum lies at x^*. The minimization function is then solved using an error-tolerant solver such as gradient descent or conjugate gradient. Our approach, on the other hand, casts an application into a Markov chain where peak(s) in the limiting distribution correspond to the application output(s).

This work was supported in part by Systems on Nanoscale Information fabriCs (SONIC), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

More generally, the proposed approach falls into the category of algorithm-based fault tolerance (ABFT) [4]. Most prior ABFT work is focused on error detection. Correction still relies on checkpointing and restart. Checkpoint-and-restart approaches may have prohibitive overhead at high fault rates. Some works do exist on algorithmic correction [4, 5]. However, these approaches are specific to algebra applications.

A related body of work exists on probabilistic computing [6]. However, prior approaches have focused on techniques to program and execute probabilistic applications. This is the first work to the best of our knowledge that attempts to build robust general applications by casting them into probabilistic implementations.

III. Background and motivation

In this section, we review some salient properties of Markov chains and discuss how applications can use Markov chains to find their solutions and why such Markov chain algorithms would be robust to certain errors.

A. Applications and Markov chains

Consider an application that has a set of possible solutions or states. One of those solutions is actually the correct solution or a goal state. For example, consider sorting. Any permutation of the inputs is a state and one such permutation is the goal state. Assuming that there exists an efficient mechanism to check if a particular solution is the correct solution, one approach to find the correct solution of the application is to generate samples from the state space and check to see if any of them is indeed the correct solution. First consider the case when these samples are generated completely randomly (Figure 1(a)). This scenario corresponds to generating samples from a uniform distribution over the states. For practical applications, the state space is generally large. So, if samples are generated completely randomly, it might take a large number of samples before we find the correct solution. Clearly, this scheme is not efficient.

A Markov chain algorithm performs the above sampling more intelligently. A Markov chain algorithm is an iterative algorithm which, in every iteration, produces a sample from the sample space of the application (Figure 1(c)). These chains are constructed such that, for a given application, the distribution over states has a significant peak at the correct solution (or the goal state) (Figure 1(b)) i.e. the probability of generating the goal state as a sample is significantly higher than the other states. This suggests that if we use these samples to check for a solution, we will find the solution much faster than when using completely random sampling. Note that such a strategy can also be adopted for the case when the application can have more than one correct solution. In that case, the *steady state distribution* over states will have multiple peaks, one at each of the solution states.

How does a Markov chain algorithm ensure such a steady state distribution over states? This is accomplished by guaranteeing specific *transition probabilities* between states. These transition probabilities dictate what state the Markov chain will produce next given the state that it has just produced. When these transition probabilities are such that the Markov chain has certain properties (*irreducibility and aperiodicity*), there is a unique steady state distribution over states [7]. This provides applications a mechanism to ensure that the overall steady state distribution over states has a specific structure (one with a peak at the goal state) simply by ensuring specific transition probabilities in each iteration of the algorithm.

The specific method to calculate the transition probabilities in each iteration depends on the application.However, for all applications, each iteration of the Markov chain algorithm consist of two essential computations (Figure 1(d)). First is the calculation of the transition probabilities. These probabilities depend on the current state and the application inputs. Second is drawing a sample for the next state from the transition probability distribution. This becomes the state that the Markov chain produces in the next iteration. These computations in each iteration, when performed appropriately for each application, results in a Markov chain that has a steady state distribution over states with peaks at the correct solutions of the application. We present several examples of this process in Section IV.

B. Robustness of Markov chain algorithms

In this section, we discuss why Markov chains are expected to be robust to certain kinds of errors. As described in Section III-A, a Markov chain algorithm performs two operations in every iteration: calculating the transition probability distribution and sampling from this distribution (Figure 1(d)). Given this understanding, we can think of the robustness of such algorithms at two different levels. First, *within an iteration*, if there are errors in calculating the transition probability distribution, it does not necessarily mean that the sample generated in that iteration would be different from the case in which there were no errors. This is the first level of error tolerance.

Second, even if errors did result in a different sample, when we look *across iterations* at the overall algorithm, the effect is that the steady state distribution over states will be different. However, it is not essential for efficient execution to have an exact distribution as long as the altered distribution has a peak at the correct solution. For such an altered distribution, it is still possible to arrive at a solution in a reasonable amount of time. Due to the above two levels of error tolerance, we expect Markov chains to produce acceptable outputs even in the presence of errors. These errors can be in transition probability calculations, in sampling from the transition probability distribution or other errors that lead to a transition error.

Errors may have an effect on runtime, however. The more the steady state distribution is altered due to errors, the higher is the potential runtime. Errors could also degrade the output quality for certain applications. We discuss these effects for example applications in Section VI. Ultimately, what transition error rates are tolerable to an application would depend on what runtime and output quality is acceptable to that particular application.

C. Generality of Markov chain algorithms

Markov chains have found wide use in the domain of statistical inference [8] and machine learning [9]. In addition, they are used to solve many other computational problems — especially hard problems — using stochastic local search methods. A few examples include the traveling salesman problem [10], the knapsack problem [11], VLSI placement, estimation of the matrix permanent [12], and constraint satisfaction problems [13]. As such, many applications already have Markov chain implementations.

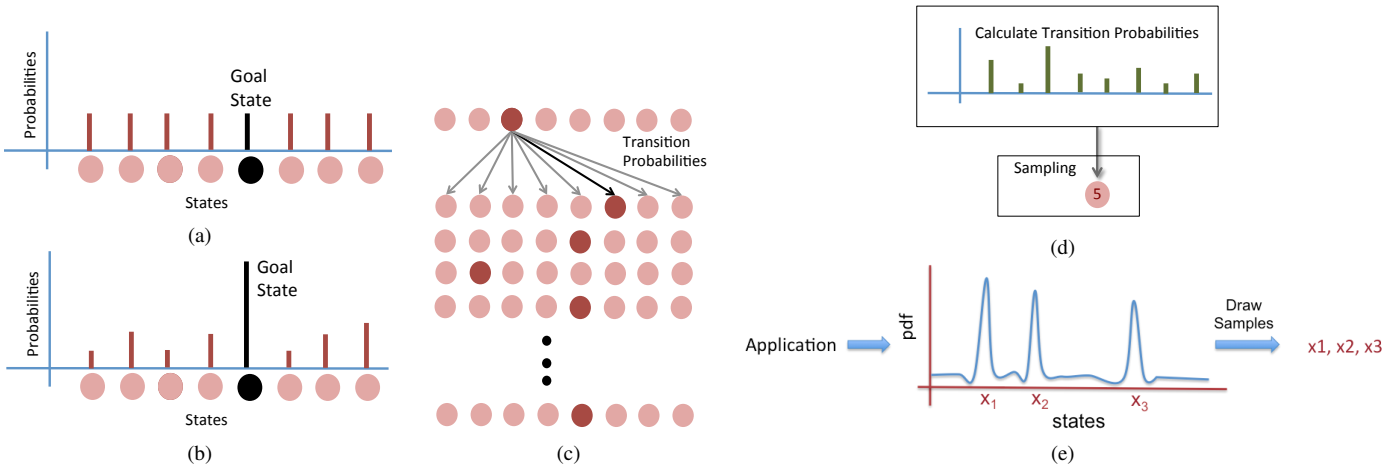


Fig. 1. (a) Random sampling in the state space results in a uniform steady state distribution over states (b) Markov chain sampling results in a steady state distribution over states that has a peak at the goal state (c) A Markov chain produces a sample from the state space in each iteration (d) In each iteration, the Markov chain algorithm performs two computations: calculating the transition probability distribution and generating a sample from the distribution.(e)An Approach for Converting Applications to Markov chain Algorithms. A Markov chain is constructed such that the steady state distribution over states has peaks at the correct solutions. Samples produced by this Markov chain are checked to see if they are the correct solutions.

In addition, applications normally *not* considered as sampling may be implemented using Markov chain algorithms by using the general methodology described in Section III-A. This involves coming up with a strategy to calculate transition probabilities in each iteration of the Markov chain such that the steady state transition probability over states has peaks at the correct solutions (Figure 1(e)). In Section IV, we describe how Markov chain implementations for four example applications are made possible by employing this methodology.

IV. Casting Applications as Markov Chain Algorithms

In this section, we present a case study of four applications that are representative of a wider range of applications to demonstrate the general methodology of casting applications as Markov chain algorithms. After discussing the details of each of these applications, we describe how Markov chain algorithms are used to find solutions for these applications and also the output quality metrics for each application.

We chose two well understood combinatorial applications: boolean satisfiability (SAT) and sorting. These applications are representative of very wide classes of algorithmic problems. Satisfiability is the canonical $\mathcal{NP}$-complete problem [14]. Therefore, if we can implement SAT as a Markov chain algorithm, we can obtain Markov chain implementation for any problem in $\mathcal{NP}$ by a reduction to SAT. Sorting is a canonical $\mathcal{P}$-complete problem [14] and so if we can cast sorting as a Markov chain algorithm, all other problems in $\mathcal{P}$ can be cast to a Markov chain via sorting. The third application, LDPC decoding, is a representative application from the area of communication systems. The fourth application, clustering without an a priori knowledge of the number of clusters, is a representative application from the domain of unsupervised machine learning.

A. Boolean satisfiability (SAT)

Boolean Satisfiability is the problem of determining if there exists a satisfying assignment to a set of boolean variables given a set of constraints. If the set of boolean variables is $\{x_1,x_2,...,x_n\}$, one example clause could be

$$x_1 \vee \neg x_3 \vee x_4$$

For a problem to be satisfiable, there must exist an assignment of the variables $\{x_1,x_2,...,x_n\}$ such that all clauses evaluate to true.

In the Markov chain sampling framework, we can think of any assignment of the boolean variables as a state and the particular assignments that result in all clauses being satisfied as the goal states. Our objective then is to construct a Markov chain that has a steady state distribution over states with peaks at the goal states. As discussed in Section III-A, this can be done by using a suitable transition probability function.

We construct a transition probability function that is based on the stochastic local search mechanism used in WalkSAT [15] — a well known and effective solver for SAT problems. We place a restriction in terms of what state transitions are allowed — from any particular state, the algorithm can only transition to another state that differs only in the assignment of a single variable. Thus, transition probabilities to other states become equivalent to probabilities of different variables in the present state being flipped. We calculate these probabilities over variables and sample from it using the following procedure in each iteration:

1) Pick a clause i by sampling from a distribution over clauses where the probability assigned to a clause $\mathbf{x}$ is given by
$$p(i \,|\, \mathbf{x}) = \frac{1}{Z} \exp\left\{-\frac{1}{\beta} g_i(\mathbf{x})\right\} \quad (1)$$
where $g_i(\mathbf{x})$ is either 0 or 1 based on whether the clause is satisfied or not and Z is a normalizing factor ensuring that the probabilities sum up to 1 over all clauses. In this distribution, unsatisfied clauses are assigned a high probability and satisfied clauses a low probability.
2) We assign a probability of zero to any variable that is not present in clause i. For variable x_j present in position j in clause i, let $N(x_j)$ be a function that returns the break count (the number of satisfied clauses that become unsatisfied if that variable is flipped) for the variable. Let G_i be the set of variable indexes associated with clause g_i. Using these quantities, we can define the conditional probability of the random variable $j \in G_i$ given clause i:
$$p(j \,|\, i, \mathbf{x}) = \frac{1}{Z} \exp\left\{-\frac{1}{\alpha} U(x_j)\right\} \quad (2)$$

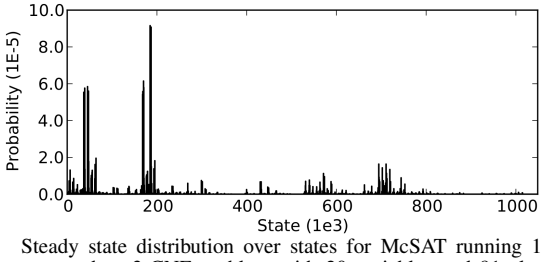


Fig. 2. Steady state distribution over states for McSAT running 1 million iterations on random 3-CNF problem with 20 variables and 91 clauses. We verified that the distribution indeed has peaks at the satisfied states.

with Z is a normalizing factor and $U(x_j)$ is an energy function defined by

$$U(x_j) = N(x_j)\left(1 + k \cdot \delta\left[N(x_j) > 0\right]\right).$$

Here, α is our temperature and $k > 0$. The energy function ensures that variables with low break counts are assigned high probabilities. Pick a variable j by sampling from this distribution.

3) Flip the state of variable j.

Figure 3 presents McSAT, the Markov chain algorithm based on this procedure. Every iteration of this algorithm computes a probability distribution over variables, samples from it, and flips that variable. This is equivalent to sampling the next state from a transition probability distribution over states. The procedure used to calculate these probabilities results in the steady state distribution over states that has significant peaks at the goal states.. We present the empirical probability distribution over states (the binary assignment $\mathbf{x}$ converted to base-10) when the algorithm is allowed to run for 1 million iterations without the termination condition for a random 3-CNF problem in Figure 2. We observe that the steady state distribution over states indeed has significant peaks at the states that satisfy this problem.

Note that, in case of SAT, any solution that satisfies all clauses is acceptable. Thus as long as the Markov chain algorithm produces such a solution, there is no effect on the application's output quality.

B. LDPC decoding

Low density parity check (LDPC) codes are linear block codes characterized by sparse parity check matrices. A (N, M) linear block code has a codeword of length N with M parity check bits. It is specified by its parity check matrix $\boldsymbol{H}$ and a generator matrix $\boldsymbol{G}$. $\boldsymbol{H}$ has the property that $\boldsymbol{Hc} = \boldsymbol{0}$ where $\boldsymbol{c}$ is a valid codeword. $\boldsymbol{G}$ is used to encode a message $\boldsymbol{u}$ into a codeword $\boldsymbol{c} = \boldsymbol{Gu^T}$. This process is called source encoding. Once the message has been encoded, the codeword is sent from the transmitter to the receiver over a noisy channel.

Source decoding is performed at the receiver by computing the syndrome s given by $\boldsymbol{s} = \boldsymbol{Hz}$ where H is the parity check matrix of the code that was used. If $\boldsymbol{z}$ differs from $\boldsymbol{c}$ due to channel noise, the result will be a non-zero vector. Thus, when $\boldsymbol{s} \neq 0$, an error has been detected and error correction will have to be performed. If, on the other hand, $\boldsymbol{s} = 0$ the received codeword is believed to be valid, and the message can be extracted from it by looking at the appropriate K bits.

In a Markov chain setting, the assignment of the N bits in the received vector can be considered a state. An assignment for which $\boldsymbol{s} = 0$ is a goal state. Our objective is to construct a Markov chain that has a steady state distribution over states with peaks at the goal states. To do that, we construct a transition probability function that is based on the weighted bit flip algorithm (WBF) [16] — one of the several algorithms for decoding LDPC codes. We place the restriction that the algorithm can only transition to a state that differs in the assignment of a single bit. Thus, transition probabilities to other states become equivalent to probabilities of different bits in the present state being flipped.

Before we introduce our procedure for calculating the probabilities of different bits being flipped and sampling from their distribution, let us define some notation. Vectors $\boldsymbol{c}$, $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ are indexed by n and vector $\boldsymbol{s}$ by m. The element of matrix $\boldsymbol{H}$ at index (m, n) is referred to as $h_{m,n}$. Also, it is important to keep in mind that row m of $\boldsymbol{H}$ represents the m-th check and all the non-zero bits in that row correspond to the bits in the encoded message that participate in that check. We denote this set of code word bits participating in the m-th check as the neighborhood system $\mathcal{N}(m) := \{n : h_{m,n} = 1\}$. Similarly, all parity checks in which bit n participates is denoted by the neighborhood system $\mathcal{M}(n) := \{m : h_{m,n} = 1\}$.

Using this notation, a Markov chain algorithm is presented in Figure 4. We call this algorithm McWBF. In each iteration, the algorithm flips one bit in the received vector $\boldsymbol{z}$. To decide which bit to flip, it first computes probabilities for each bit based on an energy function E_n that depends on how many unsatisfied parity checks that particular bit appears in. It then computes a probability of flipping each bit based on the energy function and then samples from the probability distribution to choose a bit and flips it. The procedure used to calculate these probabilities results in a steady state distribution over states that has significant peaks at the goal states. This, in turn, results in the recovery of a valid codeword.

C. Sorting

Sorting is the problem of finding a permutation of n input numbers $x_1, ..., x_n$ such that the numbers are arranged in ascending (or descending) order based on these indices $(x_i < x_{i+1} \forall i \in (1, ..., n-1))$.

In the Markov chain setting, we can think of any permutation of the n numbers as a state and the permutations that results in all the numbers being sorted as the goal states. One method of performing sorting is to iterate over the numbers n times and performing in-place swaps of two adjacent numbers if they are out of order. We construct a transition probability function that is based on this iterative in-place swapping scheme. We place a restriction in terms of what state transitions are allowed — from any particular state, the algorithm can only transition to another state that can be obtained by swapping two adjacent numbers. Thus, transition probabilities to other states become equivalent to probabilities of different adjacent number pairs being swapped. We calculate the probabilities over such pairs and sample from it using the following procedure in each iteration:

1) For each pair of adjacent numbers x_i and x_{i+1}, calculate the pair's probability of being swapped as

$$p(i \,|\, \mathbf{x}) = \frac{1}{Z} \exp\left\{-\frac{1}{\beta}\delta[x_i < x_i + 1]\right\} \tag{3}$$

where Z is a normalizing factor, β is the temperature of the distribution and $\delta[a]$ is one if proposition a is true and zero otherwise.

procedure McSAT(C)
  Randomly initialize $\mathbf{x}^{(0)}$
  for $t = 1, 2, \ldots$ **do**
    $i \sim p(i \mid \mathbf{x}^{(t-1)})$
    $j \sim p(j \mid i, \mathbf{x}^{(t-1)})$
    $\mathbf{x}^{(t)} \leftarrow \mathbf{x}^{(t-1)}$
    $x_j^{(t)} \leftarrow flip(x_j^{(t-1)})$
    if No unsatisfied clause **then**
      break
    end if
  end for
end procedure

Fig. 3. McSAT: A Markov chain algorithm for SAT

procedure McWBF($\boldsymbol{H}, \boldsymbol{y}$)
  for $t = 1, 2, \ldots$ **do**
    $|y|_{min,m} = \min_{n \in \mathcal{N}(m)} |y_n|$
    $E_n = \sum_{m \in \mathcal{M}(n)} (2s_m - 1)|y|_{min,m}$
    $p(N = n|\boldsymbol{z}) = \frac{1}{Z} \exp\left\{\frac{1}{\beta} E_n\right\}$
    $n \sim p(n \mid \boldsymbol{z})$
    $z_n = 1 - z_n$
    if $\boldsymbol{Hz} = 0$ **then**
      break
    end if
  end for
end procedure

Fig. 4. McWBF: A Markov chain decoding algorithm for LDPC codes.

procedure McSort($\mathbf{x}$)
  initialize $\mathbf{c}^{(0)}$
  for $t = 1, 2, \ldots$ **do**
    $i \sim p(i \mid \mathbf{x})$
    swap($x_i^{(t)}, c_{i+1}^{(t)}$)
    if Numbers are sorted **then**
      break
    end if
  end for
end procedure

Fig. 5. McSort: A Markov chain algorithm for sorting.

2) Obtain a specific pair by sampling from the probability distribution over adjacent pairs of numbers.
3) Swap that pair of numbers.

Figure 5 presents McSort, the Markov chain algorithm based on this procedure. Each iteration of this algorithm comprises of computing a probability distribution over pairs of adjacent numbers, sampling from that distribution and swaping the elements in a pair. The procedure used to calculate these probabilities results in the steady sate distribution over states to have peaks at the goal states.

D. Clustering

In clustering, the objective is to group some observed data (say x_i's) into multiple clusters based on some similarity between the data points. One method for performing clustering is by using a mixture model which assumes that the overall data was generated from a mixture of several distributions with each cluster representing the subset of the data that originated from the same distribution. Each such distribution generally has the same form $F(\theta_k)$ (say Gaussian) with different *cluster parameters* θ_k. A particular distribution contributes to the overall distribution based on its weight π_k (also called its *mixing proportion*).

The *Dirichlet Process Mixture Model (DPMM)* (Figure 6) is one such mixture model that assumes specific priors over the cluster parameters and the mixing proportions. A detailed discussion of the model can be found in [17]. Here, we briefly present the details and characteristics of this model that are relevant to understanding its use in clustering.

DPMM assumes that each data point x_i has a corresponding hidden variable z_i that represents the cluster that generated x_i. Hence, z_i takes a value k (that corresponds to a cluster number) with probability π_k. The cluster parameters θ_k are given a common prior distribution $G(\lambda)$ with hyperparameter λ (Equation 6). The distribution $G(\lambda)$ is generally chosen to be the conjugate prior of the distribution $F(\theta_k)$. $\boldsymbol{\pi}$ (a vector of all π_ks) is given a a Griffiths-Engen-McClosky (GEM) prior $\boldsymbol{\pi} \sim GEM(1, \alpha)$ (Equation 4)[18]. The conditional distributions in the model are presented below.

$$\boldsymbol{\pi} \mid \alpha \sim GEM(1, \alpha) \tag{4}$$

$$z_i \mid \boldsymbol{\pi} \sim \pi \tag{5}$$

$$\theta_k \mid \lambda \sim G(\lambda) \tag{6}$$

$$x_i \mid z_i, \{\theta_k\}_{k=1}^{\infty} \sim F(\theta_{zi}) \tag{7}$$

An interesting characteristic of the DPMM is that it allows the model to have an infinite number of clusters *a priori*. However, any finite observed dataset would only contain a finite, but random, number of clusters. Once the data is

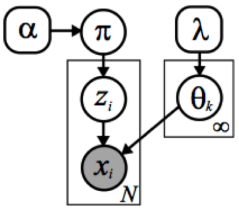


Fig. 6. Dirichlet process mixture model (DPMM).

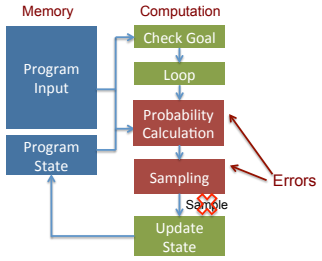


Fig. 7. Fault Model

observed, the number of clusters is inferred from the data using the Bayesian posterior inference framework. This allows the complexity of the model to grow as new data is observed allowing future data to map to previously unseen clusters. The expected number of clusters grows logarithmically with the size of the dataset.

For clustering, our objective is to construct a Markov chain algorithm that, in addition to inferring the number of clusters, also infers the values of the hidden variables z_i corresponding to each data point x_i. We can think of the assignments of the z_is as a state. The goal state is an assignment that results in an acceptable clustering based on a clustering criterion such as mean squared error.

A variety of inference methods based on Gibbs sampling (which is a Markov chain sampling algorithm) have been proposed for inference in DPMM [19]. The collapsed Gibbs sampling algorithm (Algorithm 3 in [19]) is suitable for our use of DPMM for clustering as we are only interested in knowing the cluster assignments (z_i's) and not the actual cluster parameters (θ_k's). It is an iterative algorithm that in each iteration updates the values of z_i for each data point one at a time. It does that by (a) removing x_i from its present cluster (b) computing the conditional probability of x_i belonging to each of the clusters present in that iteration and also to a potential new cluster, and (c) samples from this distribution to obtain a cluster assignment for z_i. Thus, the computation in each iteration fits into our model of calculating a probability distribution and sampling from it (Section III-A).

Since, we used a dataset generated from Gaussian distributions in our evaluations, we assumed a Gaussian mixture model with $F(\theta_k)$ being a Gaussian distribution. In such a model, the conditional probabilities of x_i's belonging to different clusters are easy to compute (details in [17]).

V. Methodology

In this section, we present the details of our methodology for evaluating the robustness of the Markov chain implementations

of the four applications discussed in Section IV. We first describe the exact implementations that were used for our evaluations. Then we describe our fault model and the error injection methodology.

A. Applications

For satisfiability, we compare two different implementation: McSAT as presented above, and an iterative version of DPLL which is a deterministic back-tracking algorithm for SAT [20]. DPLL, unlike McSAT, is a complete solver as it is able to tell the user if a given problem is unsatisfiable. Both SAT implementations were evaluated with randomly generated 3-CNF problem with 100 variables and 400 clauses. In addtion, we further evaluated the robustness of McSAT using SAT-encoded problems from the following domains: graph coloring, planning, and all interval series.

For sorting, we compare McSort as presented above with a baseline implementation of QuickSort. These were evaluated using a fully randomized sequence of integers in the range $(1, 1000)$. For LDPC decoding, we evaluated the robustness of the McWBF decoding algorithm. We used randomly generated messages that were encoded and decoded using a Gallager (273,82) LDPC code. We used DPMM for clustering and used a collapsed Gibbs sampler for inference. We used code for the collapsed Gibbs sampler that is available online [1]. The dataset used for clustering consisted of 200 two dimensional data points generated from a mixture of five Gaussian distributions. Accordingly, the DPMM model assumed the data to be a mixture of Gaussian distributions. The Gibbs sampling algorithm was run for 25 iterations and the mean squared error (MSE) of the clustering assignment was used as the quality metric.

B. Fault model and error injection methodology

The fault model assumed in our evaluations is shown in Figure 7. We consider faults that manifest themselves as errors in the transition probability calculation and sampling in each iteration. We assume that memory is fault-free and so are the other parts of the computation (loop iteration, termination checking and state update). This is a reasonable model to work with as the transition probability calculation and sampling constitute the bulk of the computation in each iteration for our applications. As such, other parts of the computation can be made robust using redundancy-based techniques.

We perform error injections at the algorithmic level based on the above model. The worst effect of a fault in the transition probability calculation or in sampling in an iteration is to produce a different sample. Hence, in our injection experiments, we modify the source code to corrupt the sample produced in each iteration with a given error rate. The corrupted sample is assigned a random sample in the state space.

In McSAT, the objective of each iteration is to find a suitable variable to flip. In our error injections, we randomly assign a variable to be flipped (instead of the one selected by the algorithm) at the end of an iteration with a particular error rate. To make as fair a comparison as possible, we inject the same type of error in DPLL. Similar to McSAT, in McWBF, the algorithm selects a bit to flip in the received signal vector in each iteration. We corrupt the result of a particular iteration by selecting a random bit to be flipped . The number of such iterations is again governed by the fault rate. For sorting, we inject errors by randomly choosing elements to swap at the end of an iteration with a particular error rate, instead of the choices made by the algorithms. For clustering, in each iteration, when new cluster assignments are being computed for each of the data points, we assign the data points to one of the existing clusters or to a new cluster randomly.

[1] https://github.com/jacobeisenstein/DPMM

VI. Results

The results of the algorithmic error injections for SAT and sorting are shown in Figure 8. McSAT is a randomized algorithm and as such, it has a variable runtime even without errors. We show the distribution of runtimes (in terms of the number of iteration) at different error rates. We observe that the algorithm produces acceptable runtimes even at fault rates as high as 10% or 20% (the median runtime is still within the 75-th percentile mark for the error-free case). In addition, the results also exhibit a gradual degradation in runtime as error rates increase.

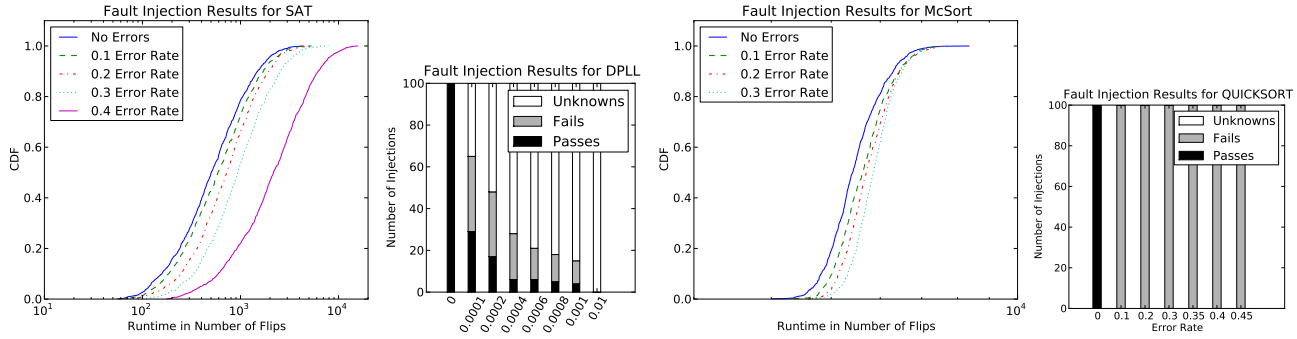
In comparison, the deterministic algorithm, DPLL, does not show similar robustness to errors. Figure 8 also present the fraction of runs where DPLL produced correct outputs, incorrect outputs or simply crashed. While DPLL manages to produce correct results in some cases, it either returns an incorrect result (fails) or crashes (unknowns) in the majority of situations.

In order to verify that the observed robustness is not an artifact of a particular input, we repeated our experiments for McSAT with other inputs. We specifically consider SAT-encoded problems from the following domains: graph coloring, planning, all interval series and logistics. All input files were taken from the SATLIB benchmark suite [21]. Results presented in Figure 9 show that McSAT displays significant robustness across all inputs.

McSort also shows acceptable runtimes in presence of errors with high error rates and shows gradual degradation of results as error rates increase. In contrast, the deterministic algorithm, Quicksort, does not always produce correct outputs in the presence of errors.

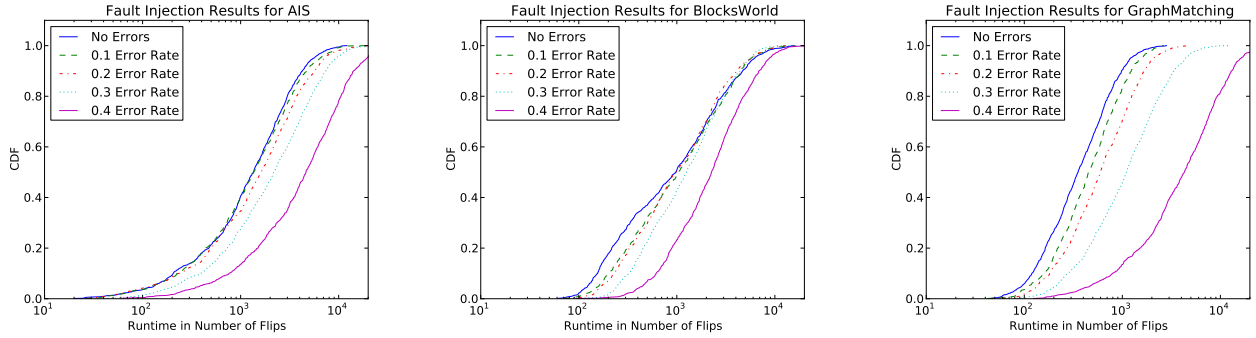
The results of the algorithmic error injections in LDPC decoding using McWBF are shown in Figure 10. Similar to McSAT and McSort, LDPC decoding shows acceptable runtimes even in the presence of errors with high error rates and shows gradual degradation in runtime as error rates are increased. Figure 10(b) shows the BER performance of the algorithm at different error rates. We observe that at error rates of 0.1% or 1%, the BER performance remains unchanged. At higher error rates, there is a gradual degradation in BER performance.

Figure 11 shows the effect of errors on clustering using a DPMM model and Gibbs sampling. At 0.1% error rate, the clustering is still performed correctly. However, at 1% error rate, we observe a few samples that are clustered incorrectly. Thus, errors have an effect on the output quality in this case. Figure 11(d) shows the mean squared error (MSE) of the clusters as number of iterations increase for different error rates. We observe that errors do not effect the runtime. In each case, the algorithm produces a cluster with the minimum MSE possible in about the same number of iterations. However, in terms of the minimum MSE achieved (a measure of output



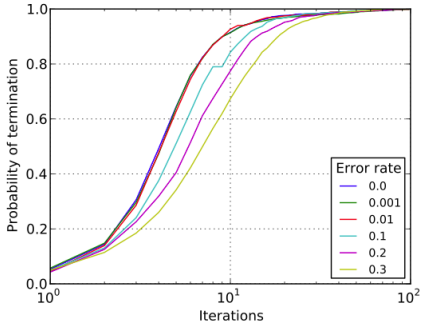
(a) McSAT (b) DPLL (c) McSort (d) Quicksort

Fig. 8. Algorithmic error injection results for SAT and sorting. The plots for McSAT and McSort show the runtime distribution for 1000 runs of the algorithm under different error rates as they always produce the correct output at these fault rates. DPLL and Quicksort on the other hand, frequently fail in producing result

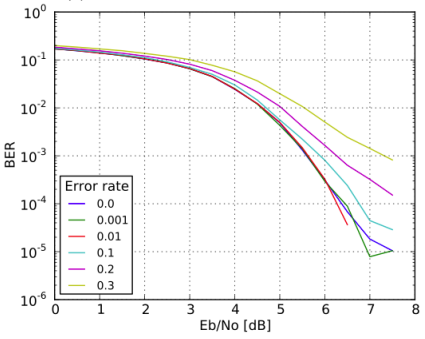


(a) AIS (b) Blocks World (c) Graph Coloring

Fig. 9. Robustness of McSAT Across Different Inputs from SATLIB.



(a) McWBF Runtime Distribution



(b) McWBF Bit Error Rate (BER) Performance

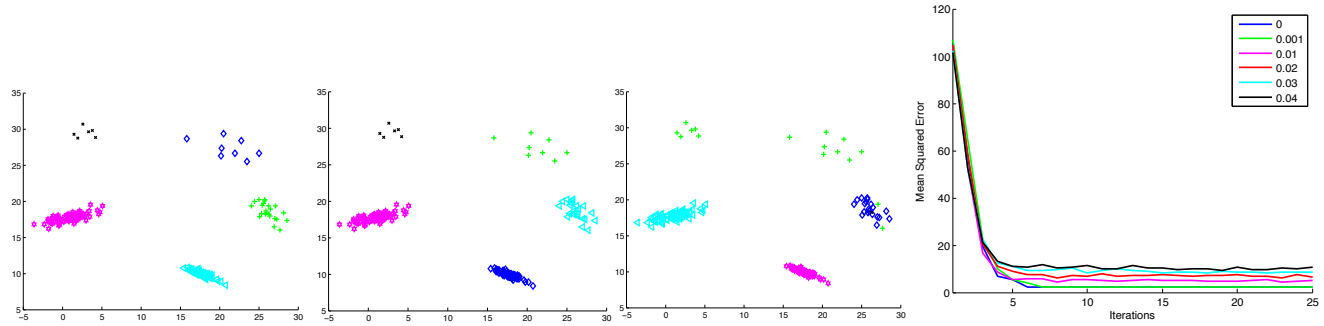
Fig. 10. Algorithmic error injection results for LDPC decoding using McWBF algorithm. The runtime distribution is at a particular signal to noise ratio (at $E_b/N_0 = 5$dB). The BER performance shows the effect of errors on the output quality of the application (lower is better).

quality), there is a gradual degradation with increasing error rates.

VII. Discussion and Future Work

Our results presented in Section VI show that Markov chain algorithms can produce acceptable results even in the presence of errors at high error rates. In order to demonstrate the feasibility of using such algorithms as a template for building low power systems using unreliable post CMOS devices, future work will focus on hardware implementations of Markov chain algorithms for specific applications. These would include applications such as, *neural spike sorting* [22] and *stereo matching* [23] for which Markov chain sampling algorithms exist, but have not been implemented in hardware. Using fine grained fault injection experiments, the robustness of these implementations to errors created by different hardware faults would be evaluated. In addition to robustness, such hardware implementations would also enable us to determine the potential for exploiting different *sampling level parallelism* strategies ([24]) for increased performance.

Hardware implementations would also enable us to determine the potential energy savings that can be achieved by different techniques that trade off robustness for energy benefits. This includes techniques such as function under-design and voltage overscaling. In order to appreciate how much energy savings can be achieved from such an approach, let us assume that we operate at a 14% error rate at the hardware level. In the worst case, where none of these faults get masked, this would lead to a 14% error rate at the algorithm level which we showed is acceptable for applications such as SAT and LDPC decoding. From the voltage-error model presented in [3], this translates to operation at 70% of the nominal voltage leading to operation at $0.7^2 = 0.49 \approx 50\%$ of the nominal energy. Energy savings may be possible in non-voltage-overscaling scenarios as well. For example, the high robustness of certain hardware



(a) No errors (b) 0.01% error rate (c) 0.1% error rate (d) Mean squared error

Fig. 11. (a,b,c) Algorithmic error injection results for clustering using a collapsed Gibbs sampler in a DPMM. (d) Mean squared error (MSE) for clustering at different error rates.

block may mean that they could be implemented using low-power, inherently unreliable technologies. Similarly, it may be possible to implement several datapath blocks at reduced precision with minimal effect on the quality of output.

In the longer term, our research goals are to (a) evaluate the robustness of Markov chain algorithms not just at the algorithmic level but also the robustness of their hardware and software implementations, (b) cast applications that are generally not implemented as Markov chain algorithms as Markov Chain-based implementations to achieve better robustness, and (c) development of a general execution model and programmable platform that would allow the robust execution of a variety of Markov chain-based implementations on the same low power system.

VIII. Conclusion

Based on the insight that Markov chains can be tolerant to errors in transition probabilities, we investigated building robust applications by implementing them as Markov chain algorithms. Several applications already have Markov chain implementations, whereas others can be converted to Markov chains. To demonstrate the process, we implemented four applications — SAT, LDPC decoding, sorting and clustering as Markov chains and evaluated their robustness using algorithmic fault injections. We demonstrated that these applications, when cast as Markov chains, are significantly more robust than their deterministic implementation. In fact, these algorithms were evaluated at transition error rates as high as $10 - 20\%$. Even at such high error rates, Markov chain algorithms produced acceptable results in terms of runtime and output quality. Also, these algorithms showed gradual degradation in runtime or output quality with increasing error rates. Based on these results and based on the generality of Markov chain algorithms, we make a case for using Markov chain algorithms as a template for implementing applications on future robust, low power hardware.

References

[1] D. Blaauw, S. Kalaiselvan, K. Lai, W.-H. Ma, S. Pant, C. Tokunaga, S. Das, and D. Bull, "Razor ii: In situ error detection and correction for pvt and ser tolerance," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, 2008, pp. 400–622.

[2] N. Shanbhag, S. Mitra, G. De Veciana, M. Orshansky, R. Marculescu, J. Roychowdhury, D. Jones, and J. Rabaey, "The search for alternative computational paradigms," *Design Test of Computers, IEEE*, vol. 25, no. 4, pp. 334–343, 2008.

[3] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *Dependable Systems and Networks (DSN), 2010*, June 2010.

[4] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. C-33, no. 6, pp. 518 –528, 1984.

[5] J. Sloan, G. Bronevetsky, and R. Kumar., "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance on parallel systems." in *Dependable Systems and Networks (DSN), 2013*, June 2013.

[6] V. Mansinghka, "Natively probabilistic computation," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[7] D. W. Stroock, *An introduction to Markov processes*. Springer, 2005, vol. 230.

[8] J. Besag, "Markov chain monte carlo for statistical inference," University of Washington, Center for, Tech. Rep., 2000.

[9] C. Andrieu, N. De Freitas, A. Doucet, and M. I. Jordan, "An introduction to mcmc for machine learning," *Machine learning*, vol. 50, no. 1-2, pp. 5–43, 2003.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[11] B. Morris and A. Sinclair, "Random walks on truncated cubes and sampling 0-1 knapsack solutions," in *in Proc. 40th IEEE Symp. on Foundations of Computer Science*. IEEE Computer Society Press, 2002, pp. 230–240.

[12] M. Jerrum, A. Sinclair, and E. Vigoda, "A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries," *Journal of the ACM*, pp. 671–697, 2004.

[13] H. H. Hoos, "Stochastic local search – methods, models, applications," Ph.D. dissertation, Darmstadt Technische Universität, 1998.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[15] B. Selman, H. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," in *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, 1995, pp. 521–532.

[16] J. Zhang and M. Fossorier, "A modified weighted bit-flipping decoding of low-density parity-check codes," *Communications Letters, IEEE*, vol. 8, no. 3, pp. 165–167, 2004.

[17] E. B. Sudderth, "Graphical models for visual object recognition and tracking," Ph.D. dissertation, Cambridge, MA, USA, 2006, aAI0809973.

[18] J. Pitman, "Combinatorial stochastic processes," Department of Statistics, University of California at Berkeley, Technical Report Technical Report 621, 2002.

[19] R. M. Neal, "Markov chain sampling methods for dirichlet process mixture models," *Journal of computational and graphical statistics*, vol. 9, no. 2, pp. 249–265, 2000.

[20] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962. [Online]. Available: http://doi.acm.org/10.1145/368273.368557

[21] H. H. Hoos and T. Stutzle. (2000, Aug.) Satlib: An online resource for research on sat. [Online]. Available: www.satlib.org

[22] F. Wood and M. J. Black, "A nonparametric bayesian alternative to spike sorting," *Journal of Neuroscience Methods*, vol. 173, no. 1, pp. 1–12, 2008.

[23] A. Barbu and S.-C. Zhu, "Generalizing swendsen-wang to sampling arbitrary posterior probabilities," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 8, pp. 1239–1253, 2005.

[24] S. Williamson, A. Dubey, and E. P. Xing, "Parallel markov chain monte carlo for nonparametric mixture models," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 98–106.