# ON FAULT TOLERANCE OF HARDWARE SAMPLERS

BY

BIPLAB DEKA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Associate Professor Rakesh Kumar

# ABSTRACT

In this research, we evaluate the robustness of hardware samplers to hardware faults. This study was motivated by our observation that several applications use sampling as a primitive and that sampling itself is an approximate method for computation. As such, it might be possible to lower energy consumption of hardware implementations of applications by implementing the samplers using more energy efficient, but fault prone, devices. We implemented a sampler in hardware and characterized its output quality in the presence of stuck-at faults and transient faults using an FPGA based gate level fault injection methodology. To understand the application level implications of such errors made by the sampler, we studied its impact on two applications: particle filtering and clustering using a Dirichlet Process Mixture Model (DPMM). Our results indicate that hardware samplers are indeed robust to hardware faults and that their robustness improves in the context of application level metrics. Specifically, we observed that (a) the two applications can tolerate multiple stuck-at faults in the sampler ( $> 5$ faults at the same time), (b) the applications can tolerate gate level transient fault rates as high as $2.4 \times 10^{-4}$, and (c) only faults in a small number of gates ($< 5.2\%$) affect the output quality of the applications. The results show that there may be significant promise to leveraging this robustness to implement sampling based applications with much higher energy efficiency than what was previously thought possible.

# ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Rakesh Kumar, for his guidance and advice throughout this research. I would also like to thank my collaborators Alex Birklykke, Henry Duwe and Vikash K. Mansinghka for their helpful comments, insights, and suggestions. Lastly, I would like to thank Qingkun Li and Zhihao Hong for our collaboration in developing the fault injection framework used in this work.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| DPMM | Dirichlet Process Mixture Model |
| MSE | Mean Squared Error |
| MMSE | Minimum Mean Squared Error |
| UART | Universal Asynchronous Receiver/Transmitter |
| PRNG | Pseudo Random Number Generator |
| PC | Personal Computer |

# CHAPTER 1

# INTRODUCTION

Our research is motivated by two trends in computing. First is the growing importance of sampling based applications. Sampling is widely used for machine learning and inference applications that include applications such as classification and clustering [1], [2]. Sampling is used in these applications to perform approximate inference as exact inference in many of these applications is not computationally tractable [1]. Also, these machine learning and inference applications are now considered some of the most important emerging high performance applications [3], [4]. Recent work has also shown that hardware implementations of sampling based applications could have significant performance and energy efficiency advantages over software implementations [5], [6].

The second trend that motivates our research is the growing unreliability of hardware devices [7]. With increased CMOS technology scaling, devices are becoming more susceptible to manufacturing defects, variability and soft errors [8]. These reliability issues are also present in post-CMOS devices [9]. As such, a lot of recent work has focused on ways to perform computation even in presence of such unreliability in hardware [10], [11]. These works exploit the fact that several classes of important applications can indeed tolerate errors introduced due to hardware faults.

Based on the above trends, our research explores the possibility of implementing sampling based applications on hardware with unreliable devices. As a first step, we evaluate the robustness of a hardware sampler - a primitive used by sampling based applications [5] - to hardware faults.

To evaluate the robustness of a hardware sampler we use an FPGA based fault injection methodology to study the effect of gate level stuck-at and transient faults. We study the effect of these faults on the quality of output of the sampler. In addition, we study two sampling based applications - particle filtering and clustering using a Dirichlet Process Mixture Model (DPMM) - to understand the application level implications of hardware faults in the sampler. Both these applications benefit from being implemented in hardware [12], [13], [14], and [6].

Our results indicate that hardware samplers are indeed robust to hardware faults and that their robustness improves in the context of application level metrics. Specifically, we observed that (a) the two applications can tolerate multiple stuck-at faults in the sampler ( $> 5$ faults at the same time), (b) the applications can tolerate gate level transient fault rates as high as $2.4 \times 10^{-4}$, and (c) only faults in a small number of gates ($< 5.2\%$) affect the output quality of the applications. To the best of our knowledge, this is the first demonstration of the fault tolerance of a hardware sampler, especially in the context of an end to end application. The results show that there may be significant energy benefits from leveraging this robustness to implement sampling based applications.

The remainder of the thesis is organized as follows. Chapter 2 describes the design of our hardware sampler and the metric used to evaluate the quality of its output. Chapter 3 describes the two applications used in our evaluations and the metrics used to evaluate the quality of their outputs. Chapter 4 describes our evaluation methodology and Chapter 5 presents our results. We present related work in Chapter 6 and conclude in Chapter 7.

# CHAPTER 2

# SAMPLING IN HARDWARE

Sampling is the operation of generating a sample from a specified probability distribution. In this research, we focus on sampling from discrete distributions. Sampling then corresponds to randomly choosing from a set of states with specified probabilities for choosing each state (Figure 2.1).

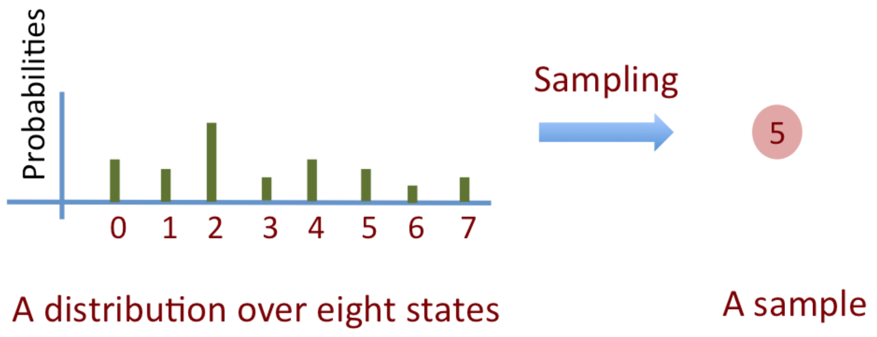


Figure 2.1: Sampling is the operation of randomly choosing from a set of states with specified probabilities for choosing each state.

Our hardware sampler is based on the *inverse CDF method* (also known as inverse transform sampling or inversion sampling) for sampling from a discrete distribution [15]. The inverse CDF method first computes the cumulative distribution function (CDF) of the specified distribution and then returns the largest entry that is smaller than a uniformly generated random number between 0 and 1.

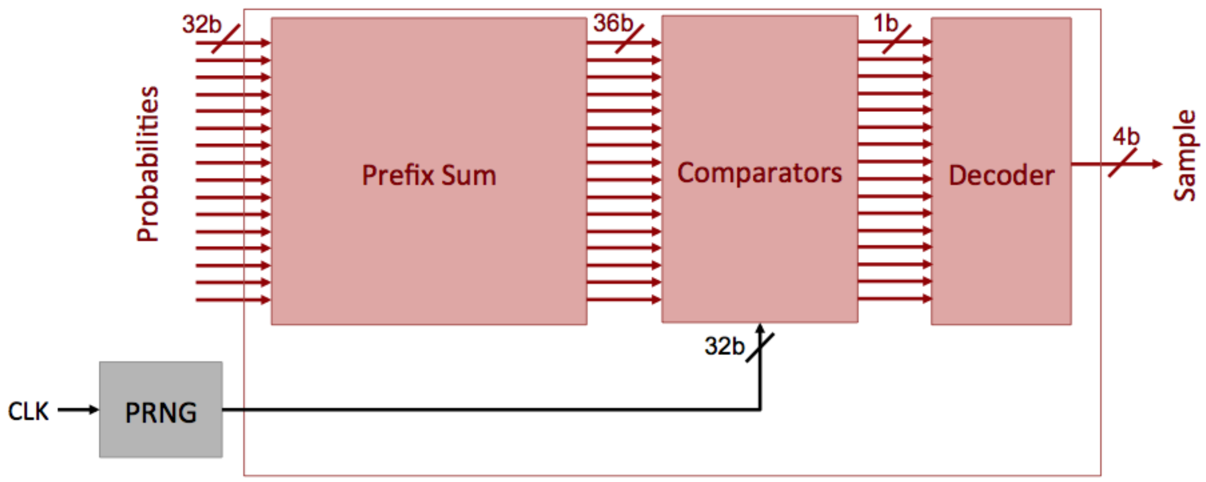


Figure 2.2: Architecture of the hardware sampler.

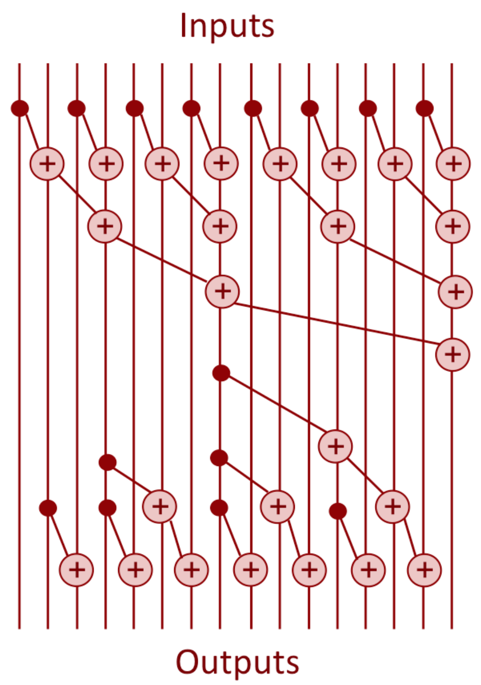


Figure 2.3: Adder tree used to implement the Prefix Sum block of the sampler.

A block diagram of our hardware sampler is shown in Figure 2.2. It consists of three blocks and takes as input a probability distribution over sixteen states. Each of these probabilities is represented as a 32 bit fixed point number. It then uses an adder tree (Figure 2.3) in the Prefix Sum block to generate the CDF. The comparators then compare these sixteen sums to an input random number. These generate sixteen 1-bit signals that the decoder block converts into a 4-bit number between 0 and 15 that is output as the sample. For our experiments, the uniform random numbers were generated in hardware using a 32-bit pseudo random number generator (PRNG) based on the XORShift Algorithm [16]. These generators have area efficient hardware implementations and their outputs pass strong statistical tests [17].

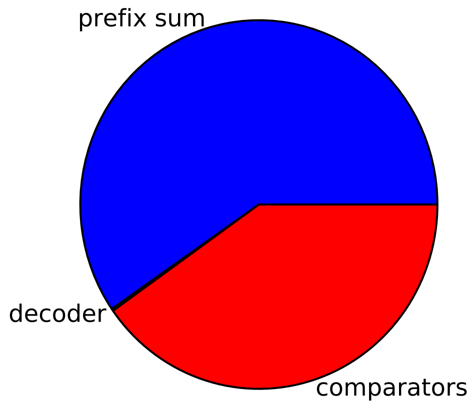
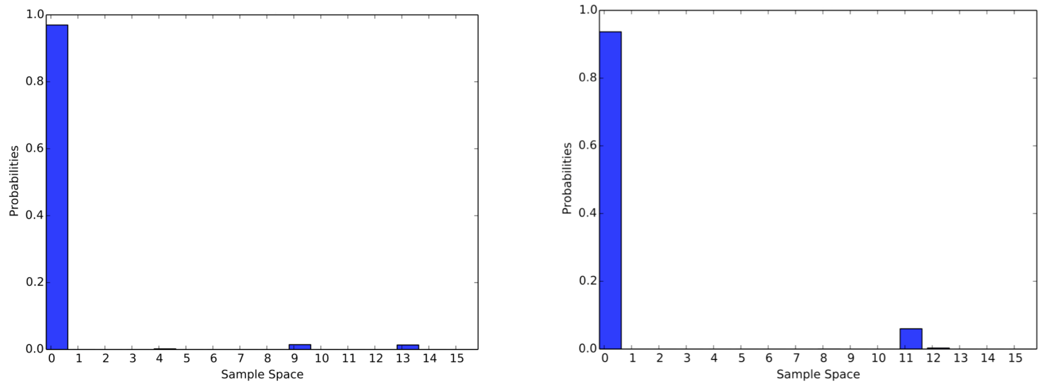


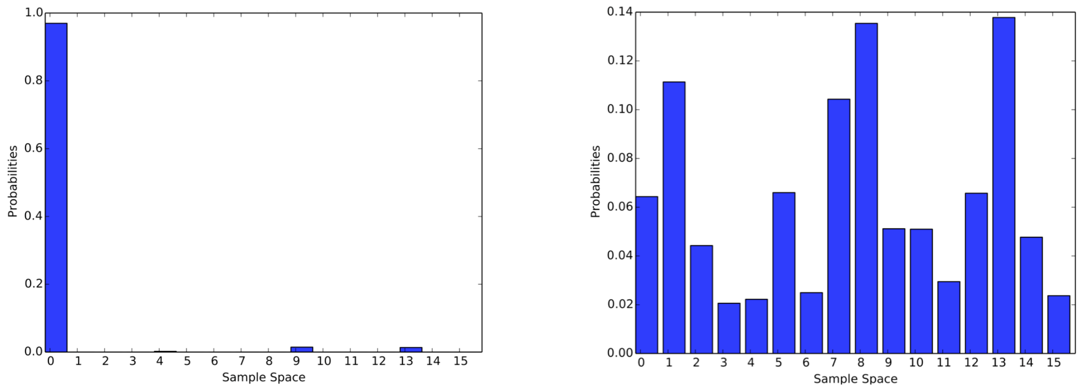
Figure 2.4: Relative number of gates in the different blocks of the sampler. Total number of gates is 3,656.

The hardware sampler was implemented at the RTL level using Verilog and was synthesized using Synopsys Design Compiler and the technology independent GTECH standard cell library. The design consisted of 3,656 gates whose distribution among the three blocks is in Figure 2.4.

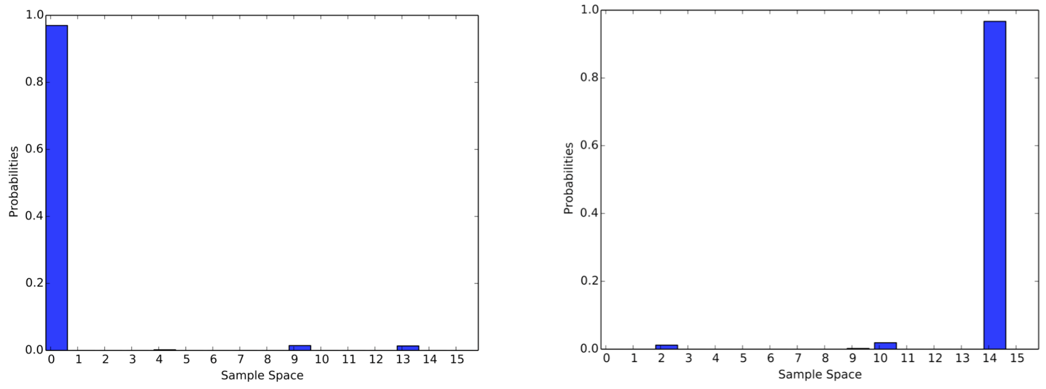
To evaluate the quality of output of the sampler, we compare the applied input distribution to the empirical distribution over 1,000,000 samples generated by the sampler. These comparisons are done in terms of the KL divergence [18] between these two distributions. The value of the KL divergence is always non-negative and is zero for two distributions that are identical. Figure 2.5 presents a few examples of KL divergence between different distributions over 16 states. We note that that at a KL divergence of about 0.3 the distributions look very similar.



(a) Two distributions with low KL divergence ($\sim 0.3$)



(b) Two distributions with high KL divergence ($\sim 4$)



(c) Two distributions with very high KL divergence ($\sim 16$)

Figure 2.5: Examples of KL divergences between different pairs of distributions.

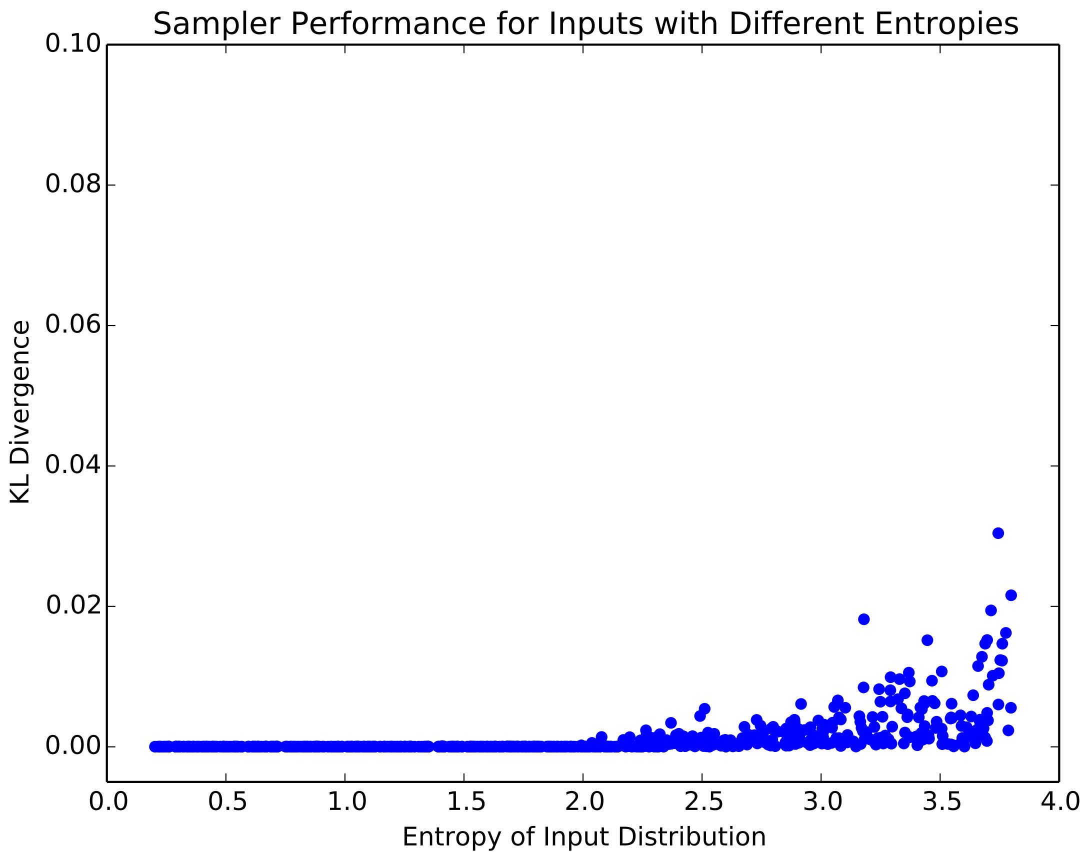


Figure 2.6: Output quality for the sampler for input distributions with different entropies.

We can characterize different inputs to the sampler in terms of their entropy [18], which is a measure of randomness. For a distribution over 16 states, the entropy ranges from 0 to 4. It has the value 0 for a distribution that has all the probability mass associated with one state. It has the value 4 for an uniform distribution. Figure 2.6 presents the output quality of the sampler (in terms of the KL divergence) for inputs with different entropies. We observe that there is a systematic degradation in output quality as the entropy increases. However, the difference is relatively small (compared to the KL divergence value of 0.3 in Figure 2.5a). We present further evaluations of the output quality of the sampler in presence of hardware faults in Chapter 5.

# CHAPTER 3

# APPLICATIONS

In this chapter, we provide an overview of two sampling based applications that we use to evaluate the application level impact of hardware faults in the sampler. Both these applications are iterative and the computation in each iteration follows a model of calculating a probability distribution and sampling from it.

## 3.1 Particle Filtering

Dynamical systems in areas such as navigation, robotics, signal processing, and time series analysis are often modeled using a state-space framework where the phenomenon of interest is viewed as the unknown state of a system [19]. Often the problem of interest is to estimate the state of the system from noisy observations of the state that are available. If the state space model is assumed to be linear, a Kalman filter is often used since it produces the linear minimum mean squared error (MMSE) estimator of the state [20]. When the linearity assumptions are not valid, the Kalman filter and its extensions have clear limitations. In such scenarios, an algorithm called *particle filtering* is used [21]. Particle filtering has found wide applications in areas such as communications, image and video processing, and computer vision (under the name of condensation algorithm [22]).

Dynamical systems to which particle filtering is applied can often be represented as

$$\boldsymbol{x}_t = f_t(\boldsymbol{x}_{t-1}, \boldsymbol{q}_t) \tag{3.1}$$

$$\boldsymbol{y}_t = g_t(\boldsymbol{x}_t, \boldsymbol{v}_t) \tag{3.2}$$

where $\boldsymbol{x}_t$ is the dynamical state of the system, $\boldsymbol{q}_t$ is the process noise, and $\boldsymbol{y}_t$ is the observation vector with measurement noise $\boldsymbol{v}_t$ at time instant $t$. $f_t$ and $g_t$ are possibly non-linear functions describing the model. Given this model and the noisy observation vectors $\boldsymbol{y}_t$ at time instant $t$, the particle filtering algorithm estimates the state of the system $\boldsymbol{x}_t$.

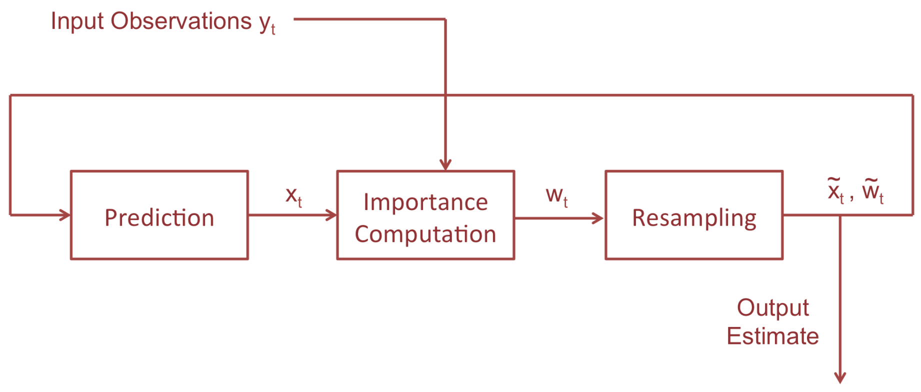


Figure 3.1: Overall structure of the particle filtering algorithm.

The algorithm starts with $N$ samples (also called particles) of the state at time $t = 0$ and then generates new samples to estimate the state at each time instant as it receives the value of the noisy observation at that instant. At each time instant, the algorithm proceeds through the following steps (shown in Figure 3.1):

1. Prediction Step: In this step, samples (particles) for a time step are computed from the samples of the previous time step using the prior probability density function (pdf) $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$. The prior pdf can be derived from Equation 3.1. The sampled set of particles at instant $t$ is denoted by $\{\boldsymbol{x}_t^{(n)}\}_{n=0}^{N-1}$.

2. Importance (Weight) Computation Step: In this step, the weights $w_t^{(n)}$ corresponding to each particle $\boldsymbol{x}_t^{(n)}$ are computed. If the prior pdf is used in the prediction step, then the weights are given by

$$w_t^{(n)} = w_{t-1}^{(n)} \times p(\boldsymbol{y}_t|\boldsymbol{x}_t^{(n)}) \tag{3.3}$$

where $p(\boldsymbol{y}_t|\boldsymbol{x}_t^{(n)})$ can be derived from Equation 3.2. These weights are then renormalized to sum to 1.

3. Resampling Step: This step samples from the set of predicted particles using their importance weights. This eliminates particles with small weights and replicates particles with large weights. The resampled set of particles is denoted by $\{\tilde{\boldsymbol{x}}_t^{(n)}\}_{n=0}^{N-1}$ and the weights of these particles are denoted by $\{\tilde{\boldsymbol{w}}^{(n)_t}\}_{n=0}^{N-1}$ which are typically set to $1/N$. These resampled particles and their weights are used to represent the posterior $p(\boldsymbol{x}_t|\boldsymbol{y}_{1:t})$ and to calculate the estimates of the state $\boldsymbol{x}_t$.

In our evaluations, the hardware sampler is used during the resampling step. We use the linear model presented in [20] for our evaluations. The state space in this case is one dimensional and the model is described as follows:

$$x_t = Ax_{t-1} + q \tag{3.4}$$

$$y_t = x_t + v \tag{3.5}$$

where $A = 1$, $q$ is Gaussian noise with mean 0 and variance 0.001, and $v$ is Gaussian noise with mean 0 and variance 0.1. For our particle filter, we assumed the variance of the initial particles to be 1. Figure 3.2 shows the estimate of the particle filter as well as a Kalman filter for 100 iterations.

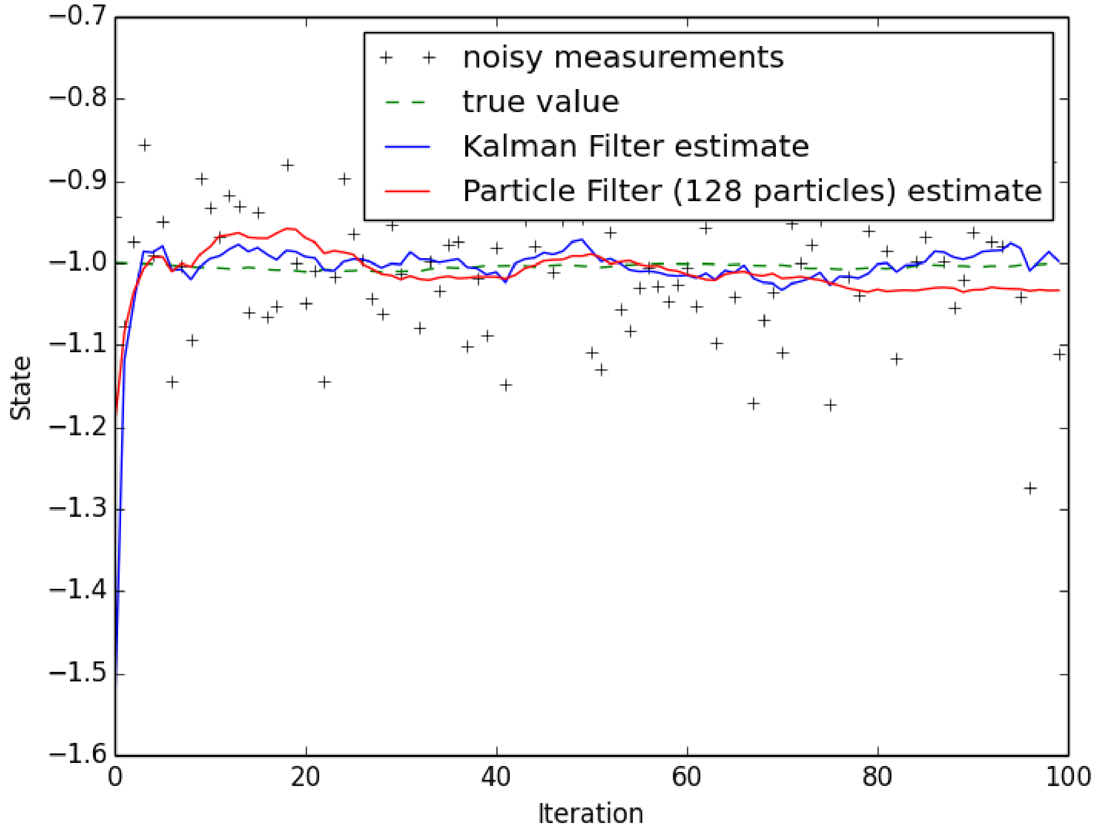


Figure 3.2: Tracking the state of a dynamical system using particle filtering. In every iteration the particle filter takes as input the noisy observation (noisy measurement) and estimates the state (true value) of the system.

The output quality of particle filtering can be characterized by the *mean squared error* (MSE) between the true state (green line in Figure 3.2) and the estimated state (red line). Figure 3.3 shows the MSE of one run of the particle filtering application. For subsequent evaluations, we use the mean of the MSE from iteration 21 to 30 as the output quality metric for particle filtering.

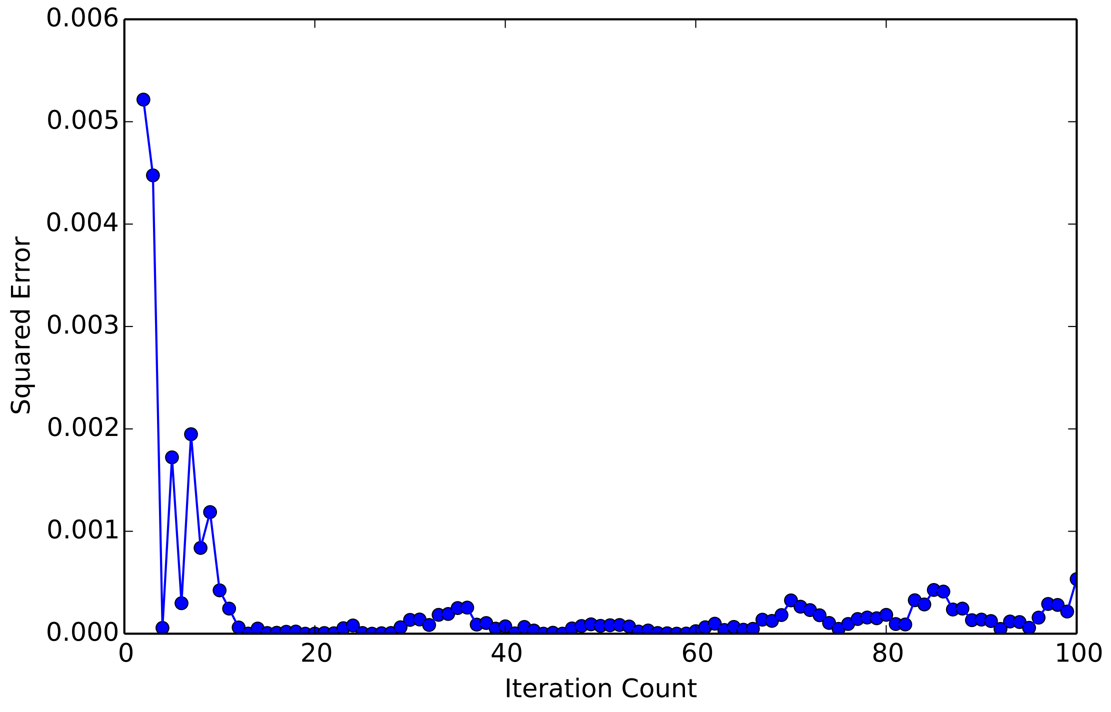


Figure 3.3: The squared error between the true state of the system and the state predicted by a particle filtering with 256 particles. We observe that the particle filter starts tracking the state successfully in less than 15 iterations. For subsequent evaluations, we use the mean of the MSE from iteration 21 to 30 as the output quality metric for particle filtering.

The output quality of particle filtering depends on the number of particles used. Figure 3.4 shows the output quality of particle filtering (in terms of MSE) for different number of particles. For subsequent evaluations, we use a particle filter with 256 particles.

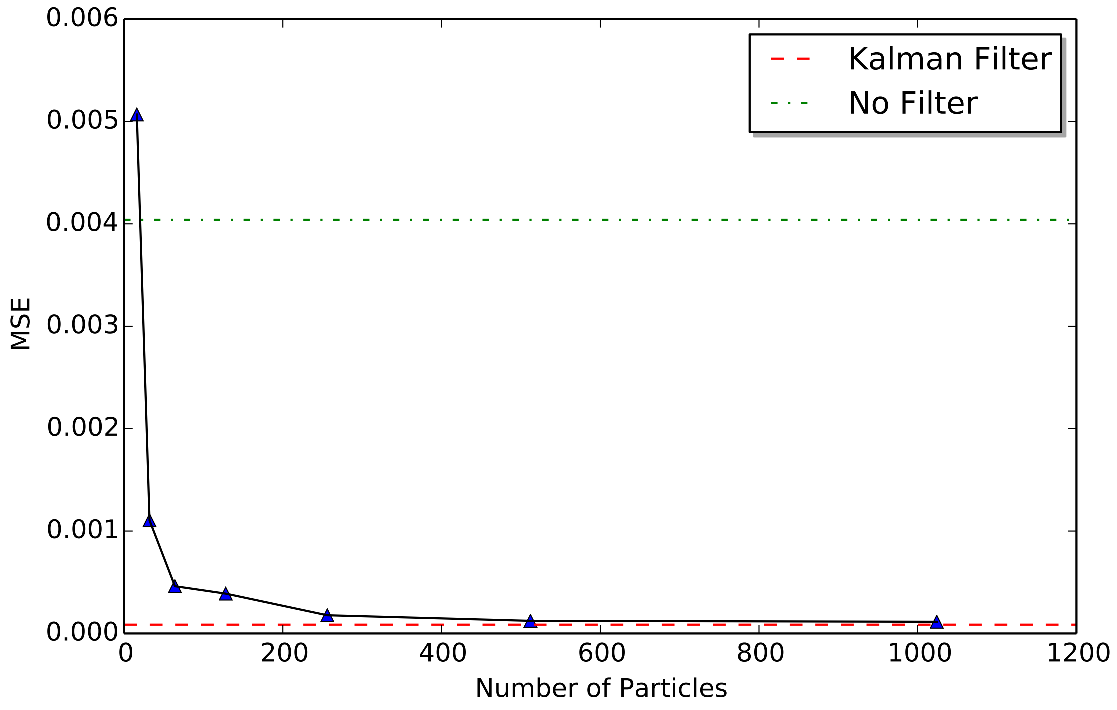


Figure 3.4: The output quality of particle filtering for different number of particles. No Filter represents the MSE of using the noisy observations as the estimate of the state of the system. For subsequent evaluations, we used 256 particles.

## 3.2 Clustering using DPMM

In clustering, the objective is to group $N$ observed data points (say $x_i$'s) into multiple clusters based on some similarity between the data points (Figure 3.5). One method for performing clustering is by using a mixture model [1] 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* $\theta_k$. A particular distribution contributes to the overall distribution based on its weight $\pi_k$ (also called its *mixing proportion*).

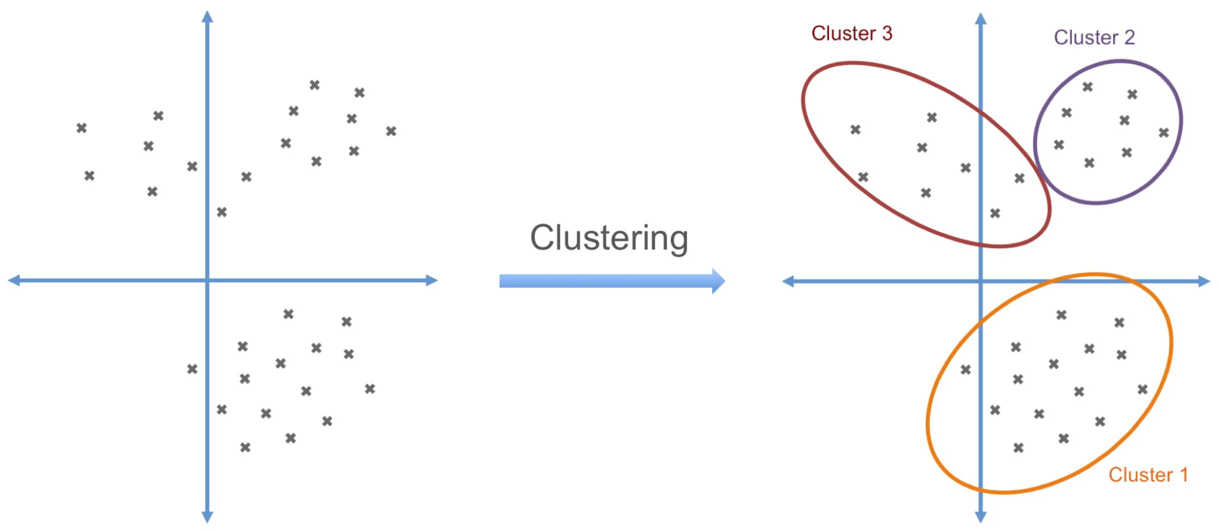


Figure 3.5: An illustration of clustering.

The *Dirichlet Process Mixture Model (DPMM)* (Figure 3.6) is one such mixture model that assumes specific priors over the cluster parameters and the mixing proportions [23]. A detailed discussion of the model can be found in [24]. Here, we briefly present the details and characteristics of this model that are relevant to understanding its use in clustering.

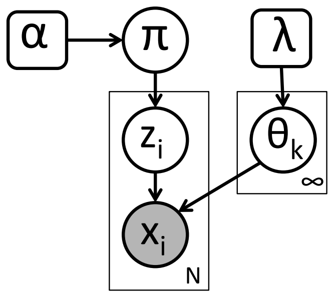


Figure 3.6: The Dirichlet Process Mixture Model (DPMM).

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 $\pi_k$. The cluster parameters $\theta_k$ are given a common prior distribution $G(\lambda)$ with hyperparameter $\lambda$ (Equation 3.8). The distribution $G(\lambda)$ is generally chosen to be the conjugate prior of the distribution $F(\theta_k)$. $\boldsymbol{\pi}$ (a vector of all $\pi_k$s) is given a Griffiths-Engen-McClosky (GEM) prior $\boldsymbol{\pi} \sim GEM(1, \alpha)$ (Equation 3.6)[25]. The conditional distributions in the model are presented below.

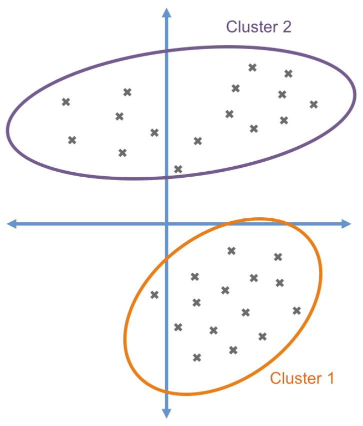
$$\boldsymbol{\pi} \mid \alpha \sim GEM(1, \alpha) \tag{3.6}$$

$$z_i \mid \boldsymbol{\pi} \sim \pi \tag{3.7}$$

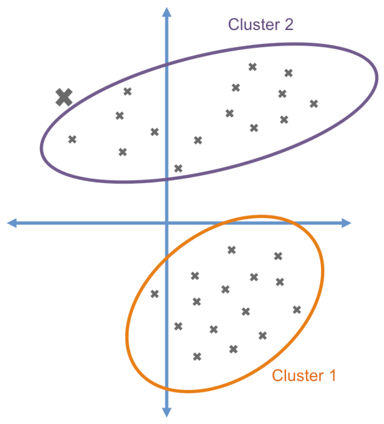
$$\theta_k \mid \lambda \sim G(\lambda) \tag{3.8}$$

$$x_i \mid z_i, \{\theta_k\}_{k=1}^{\infty} \sim F(\theta_{zi}) \tag{3.9}$$

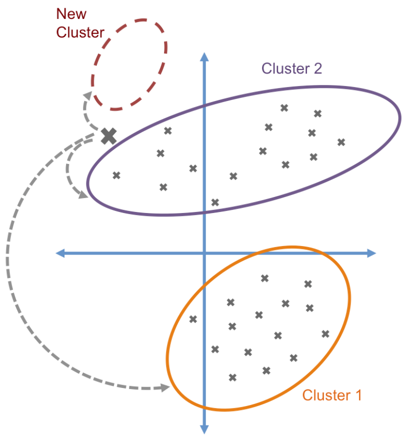
An interesting characteristic of the DPMM is that it allows the model to have an infinite number of clusters *a priori*. However, any finite observed dataset would only contain a finite, but random, number of clusters. Once the data is observed, the number of clusters is inferred from the data using 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, a sampling algorithm in addition to inferring the number of clusters, also has to infer the values of the hidden variables $z_i$ corresponding to each data point $x_i$.



(a)



(b)



(c)

Figure 3.7: An illustration of an intermediate step during the collapsed Gibbs sampling for DPMM. (a) Current clustering at the start of iteration. (b) A data point is removed from its cluster. (c) The probability of this data point belonging to each of three clusters (the two clusters present and a potential new cluster) is calculated. These three probabilities constitute a probability distribution. The algorithm then samples from this distribution and assigns the data point to the cluster corresponding to the sample.

A variety of inference methods based on Gibbs sampling (which is a Markov chain sampling algorithm) have been proposed for inference in DPMM [26]. The collapsed Gibbs sampling algorithm (Algorithm 3 in [26]) 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 ($\theta_k$'s). It is an iterative algorithm that in each iteration updates the values of $z_i$ for each data point one at a time (Figure 3.7). It does that by (a) removing $x_i$ from its present cluster (Figure 3.7a), (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 (Figure 3.7b and 3.7c), and (c) sampling from this distribution to obtain a cluster assignment for $z_i$ (this step is performed using the hardware sampler in our evaluations). Thus, in each iteration, a cluster assignment is recalculated for each data point. In this process new clusters can be created or previously created clusters can be destroyed.

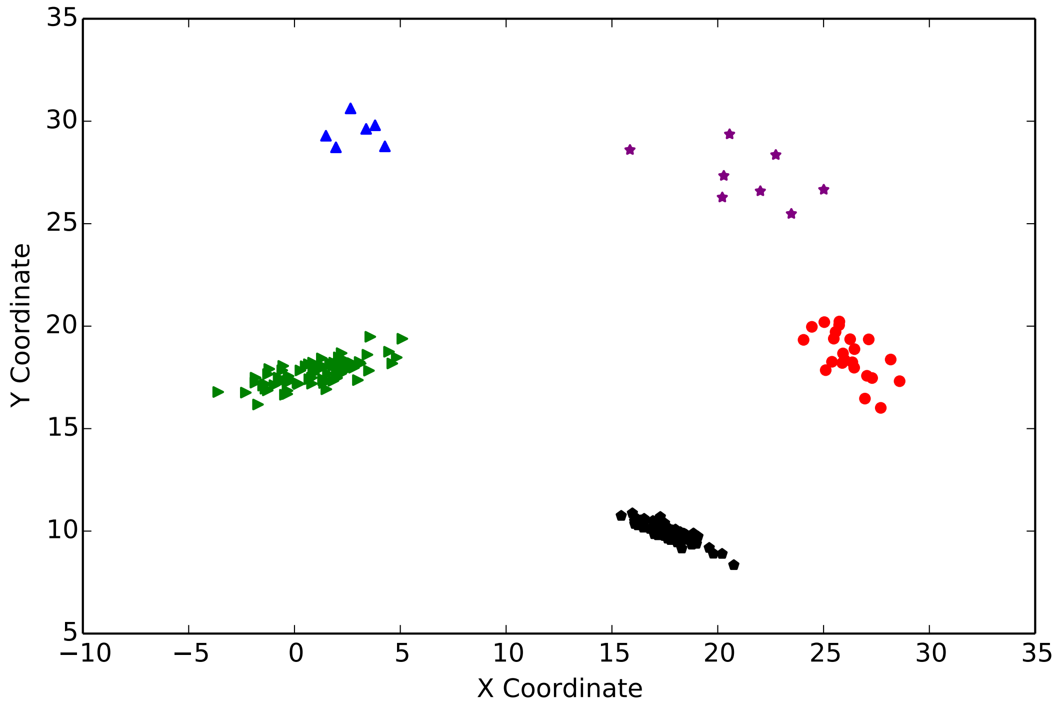


Figure 3.8: The dataset used for our evaluations consists of 200 data points generated from a mixture of five Gaussian distributions.

For our evaluations, we used a dataset of 200 points generated from a mixture of five Gaussian distributions (shown in Figure 3.8). As such, a

Gaussian mixture model with $F(\theta_k)$ being a Gaussian distribution is used. In such a model, the conditional probabilities of $x_i$'s belonging to different clusters are easy to compute (a detailed discussion can be found in [24]).

We begin the clustering algorithm with all data points assigned to a single cluster. As the algorithm iterates, the number of clusters changes and it settles to the correct value of five (Figure 3.9). Figure 3.10 shows the clustering of the datapoints at iteration 3.

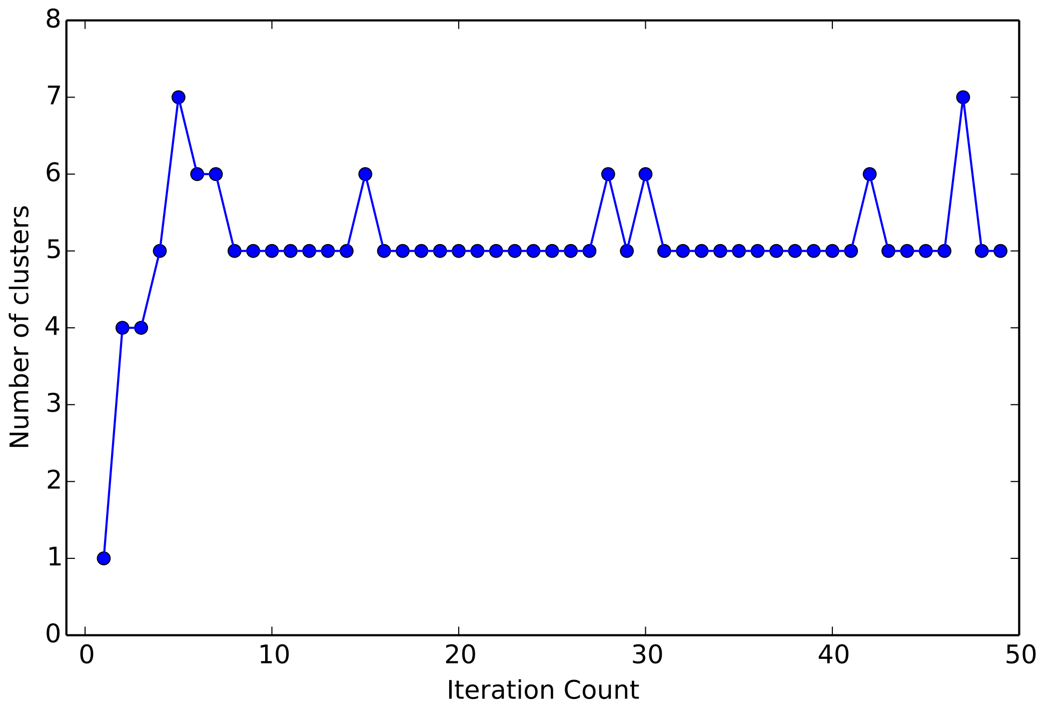


Figure 3.9: The number of clusters corresponding to each iteration for a particular run of the collapsed Gibbs sampling algorithm. The algorithm starts from all data points in a single cluster. We observe that the algorithm finds the right number of cluster (5) in less than 10 iterations.

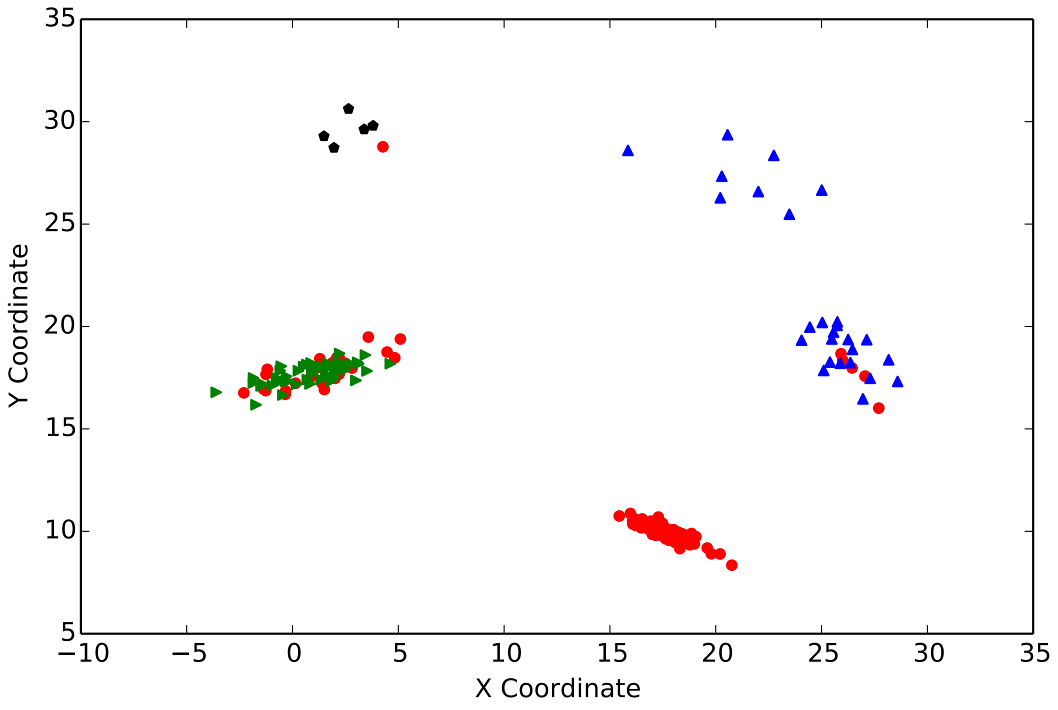


Figure 3.10: An intermediate clustering of the dataset (at iteration 3) for a particular run of the collapsed Gibbs sampling algorithm. The algorithm has clustered the data into four clusters at this point.

The output quality of clustering can be assessed by the mean of the squared error of each data point from the centroid of it cluster. We plot the mean squared error (MSE) of the entire dataset for each iteration in Figure 3.11. For our subsequent evaluations, we use the mean of the MSE from iteration 11 to 15 as an output quality metric for clustering this dataset using DPMM.

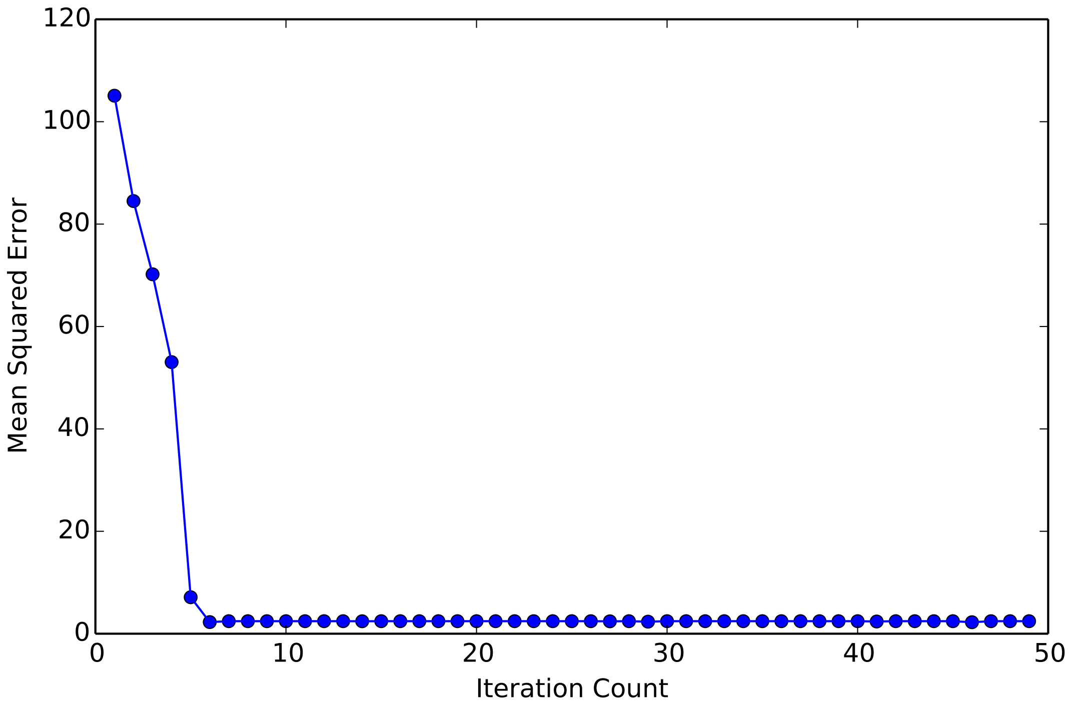


Figure 3.11: The mean squared error (MSE) of the clustered dataset at each iteration of a particular run of the collapsed Gibbs sampling algorithm. We observe that the MSE drops to a low value in less than 10 iterations. For subsequent evaluations, we use the mean of the MSE from iteration 11 to 15 as a output quality metric for clustering this dataset using DPMM.

# CHAPTER 4

# METHODOLOGY

To evaluate the robustness of the sampler to hardware faults, we considered two gate level fault models for our evaluations. The first is the *stuck-at fault* model where we assume that the output of gates can be stuck to a logical 0 or 1 value. The second fault model is the *transient fault* model where we assume that the output of a gate is flipped with a certain probability every clock cycle.

We developed an FPGA based fault injection framework[1] to operate a circuit under stuck-at and transient faults and to study the impact faults have on the final output of our applications. Our framework is targeted at the Xilinx Zedboard hardware platform [27] that has a system on chip (SOC) that combines programmable logic (FPGA) with an ARM processor. We used the Xilinx Vivado toolchain [28] for FPGA development and Xilinx SDK [29] for developing software for the ARM processor.

An overview of our fault injection framework is shown in Figure 4.1. Below we describe the individual steps in our toolflow:

- We begin with the RTL description of the circuit and synthesize to a netlist using Synopsys Design Compiler with the technology independent GTECH standard cell library [30] (Figure 4.2a).

- Our tool[2] then lets users specify the number of fault gates, randomly selects that many gates in the netlist and modifies the netlist to add extra logic and ports to enable faults at the output of each faulty gate (Figure 4.2b).

- Our tool generates a hardware wrapper that has the logic and the registers that are used to enable the faults in the netlist during operation. For stuck-at faults, the wrapper consists of registers to which the fault

[1]This was developed in collaboration with Qingkun Li and Zhihao Hong

[2]For this part, we use a modified version of the publicly available CrashTest tool [31]

enable signals are connected. For transient faults, each fault location needs a fault enable signal that is enabled with a specified fault rate. We achieve this by having hardware pseudo random number generators (PRNGs) and comparing the random number generated in each clock cycle to a value written in a register (Figure 4.2c).

- Our tool also generates a software driver that runs on the ARM processor on the Zedboard. It communicates with the personal computer (PC) using the Universal Asynchronous Receiver/Transmitter (UART) port and accepts commands to write the programmable registers in the hardware wrapper.

- Finally, our framework compiles the software drivers and the hardware components (modified netlist and the hardware wrapper) and downloads them to the Zedboard.

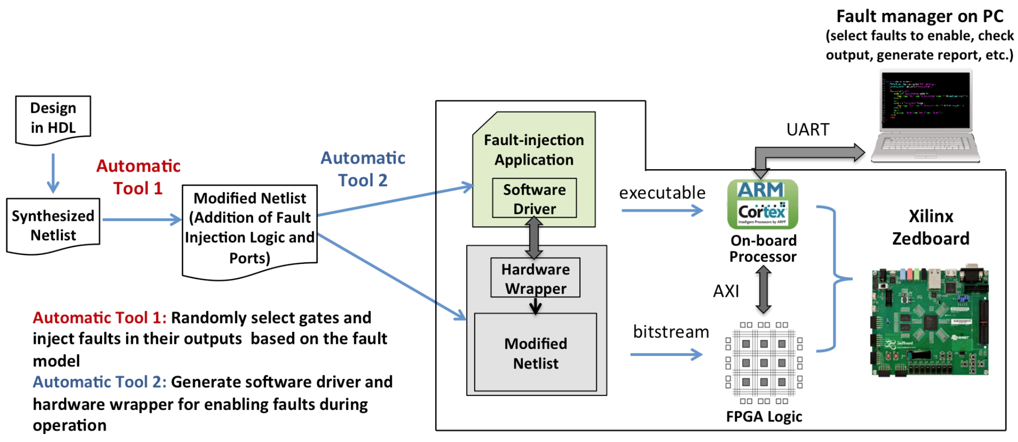
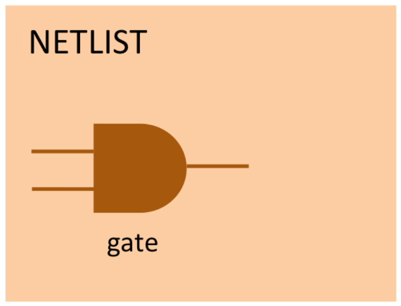
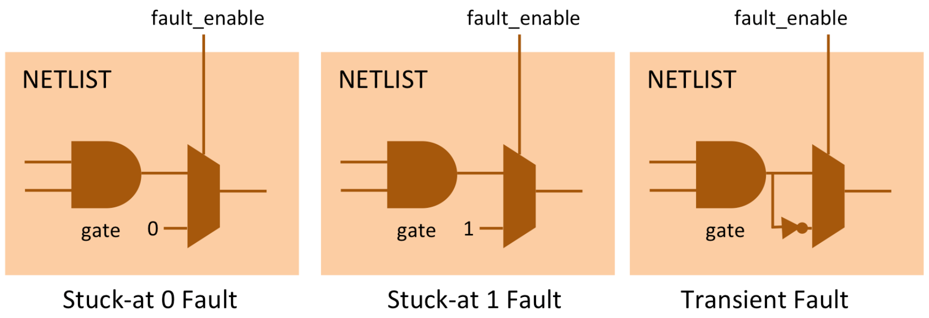

Fault manager on PC
(select faults to enable, check output, generate report, etc.)
Design in HDL
Automatic Tool 1
Synthesized Netlist
Modified Netlist (Addition of Fault Injection Logic and Ports)
Automatic Tool 2
Fault-injection Application
Software Driver
Hardware Wrapper
Modified Netlist
executable
bitstream
UART
ARM Cortex
On-board Processor
AXI
FPGA Logic
Xilinx Zedboard
Automatic Tool 1: Randomly select gates and inject faults in their outputs based on the fault model
Automatic Tool 2: Generate software driver and hardware wrapper for enabling faults during operation

Figure 4.1: An overview of the FPGA based fault injection framework.

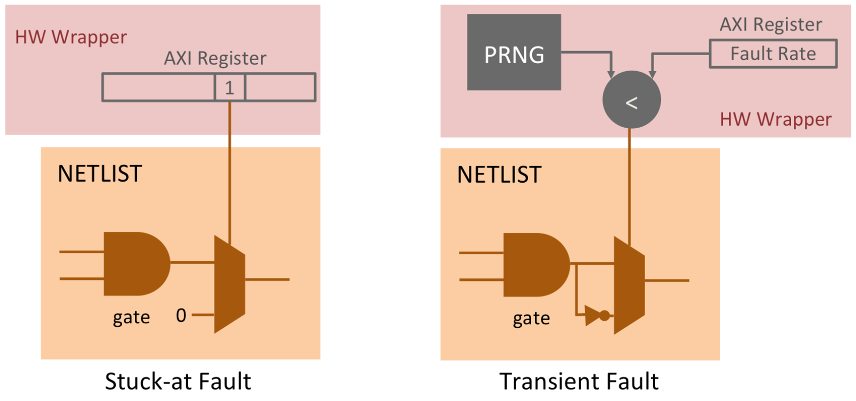
Our framework also included utilities written in the high level language Python that a user can run on the PC to perform various fault injection campaigns. We executed software implementations of our applications on the PC and replaced the sampling operations with calls to our Python utilities. These utilities return a sample by performing the sampling operation on the sampler circuit implemented on the FPGA under different fault conditions.



(a) Original Netlist



(b) Our tool adds a multiplexer at each fault location with the original input connected to one input and the other input connected to a 0, 1 or a negation of the original input. The select signal of the multiplexer is connected to a new input port.



(c) The hardware wrappers contain additional logic and registers to enable faults in the netlist.

Figure 4.2: Circuit netlist during different stages of our toolflow.

We developed the particle filtering application in-house and used it for the model presented in [20] with 256 particles. This results in the application generating samples from a distribution with 256 states every iteration. Since our hardware sampler can only sample from a distribution over 16 states, we perform two sampling operations to generate each sample. For the first sampling operation, we compose 16 bins each comprised of 16 states. We compute the probability of each bin by adding the probabilities of the states in that bin. We then sample from this probability distribution to select a bin. In the second sampling operation, we sample from the probabilities of the 16 states in the selected bin to select a state. We used the particle filter on the linear model with Gaussian noise as described in Section 3.1. The output quality metric is the mean squared error (MSE) from iteration 21 to 30 for 100 independent runs.

For clustering using DPMM, we used a publicly available implementation.[3] We used a dataset that consists of 200 data points generated from a mixture of five Gaussian distributions as described in Section 3.2. The output quality metric is the mean squared error (MSE) from iteration 10 to 15 for 30 independent runs.

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

# CHAPTER 5

# RESULTS

In this section, we present the results of our fault injection experiments as described in Chapter 4. Through our results, we aim to answer the following questions:

- How defect tolerant is the hardware sampler? Can it provide acceptable output even in the presence of multiple defects (stuck-at faults)?
- Is the hardware sampler tolerant to transient faults? Under what transient fault rates does the sampler still provide acceptable outputs?
- What fraction of the gates are critical in the sampler? How are these critical gates distributed across different sub-blocks?

First, we answer each of the above questions at the sampler level using the KL divergence as a metric to compare the quality of output of the sampler (as described in Chapter 2). Then, we answer each of the above questions at the application level in context of the two applications - particle filtering (Section 3.1) and clustering using DPMM (Section 3.2).

## 5.1 Robustness to Stuck-at Faults

To evaluate the robustness of the sampler to stuck-at faults (defect tolerance), we performed sampling with different number of stuck-at faults. For each stuck-at fault number we randomly select the fault injection locations. Figure 5.1 presents the KL divergence observed for one particular input (with entropy around 2) at different number of stuck-at faults. We report the mean of 100 independent runs for each experiment. We observe that the KL divergence remains low ($< 0.1$) even at more than 10 stuck-at faults. Also, there is gradual degradation in the quality of output with increasing number of faults. However, the acceptable value of KL divergence depends on the particular application a sampler is used for.

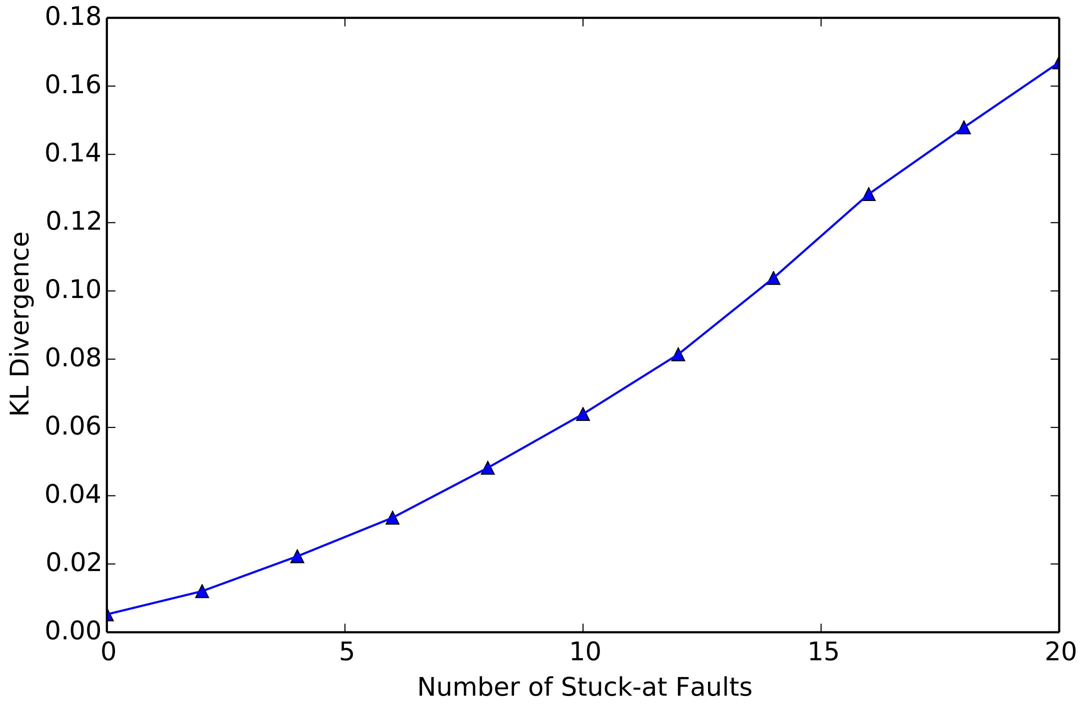


Figure 5.1: The output quality of the sampler for different number of stuck-at faults. We observe that the KL divergence remains low ($< 0.1$) even at 5-10 stuck-at faults.

Figure 5.2 presents the output quality of particle filtering at different number of stuck-at faults in the sampler. We observe that the application can tolerate multiple stuck-at faults in the sampler and performs meaningful filtering even in the presence of more than 10 stuck-at faults.

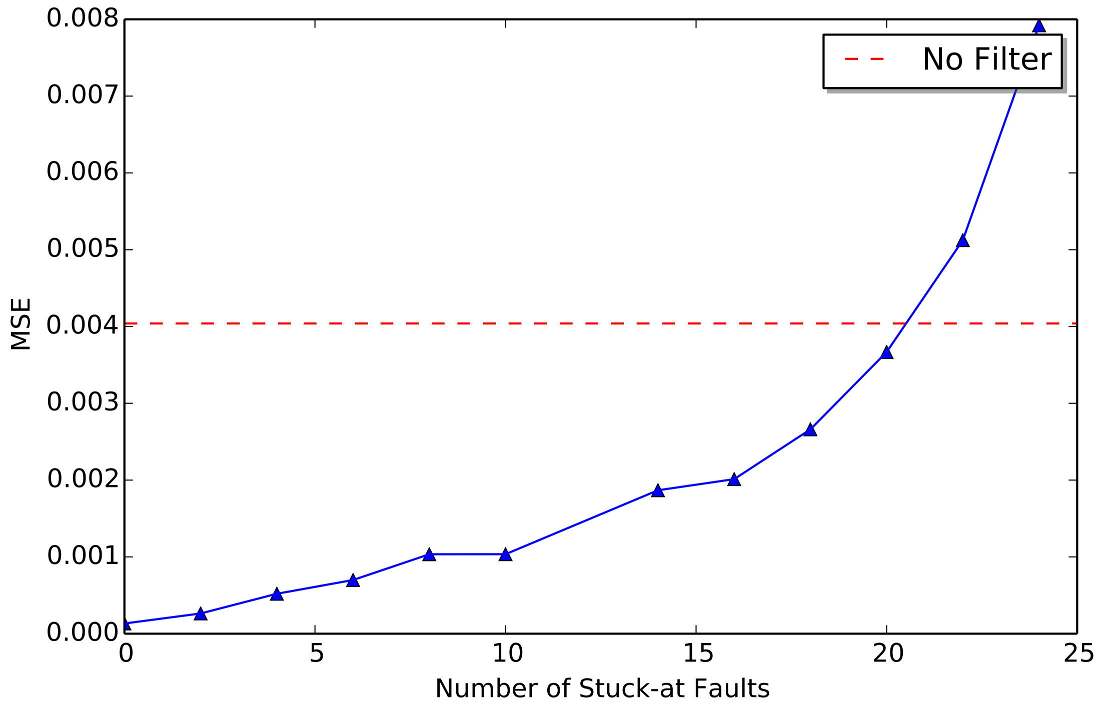


Figure 5.2: The output quality of particle filtering in the presence of different numbers of stuck-at faults. *No Filter* represents the MSE of using the noisy observations as the estimate of the state of the system. We observe that the application can tolerate multiple stuck-at faults in the sampler and performs filtering even in the presence of more than 5 stuck-at faults.

Figure 5.3 presents the output quality of clustering using DPMM at different number of stuck-at faults in the sampler. We observe that the application can tolerate multiple stuck-at faults in the sampler and performs clustering even in the presence of more than 5 stuck-at faults.

The above results indicate that the hardware samplers indeed exhibit defect tolerance and applications using the sampler can still produce meaningful outputs in the presence of multiple stuck-at faults.

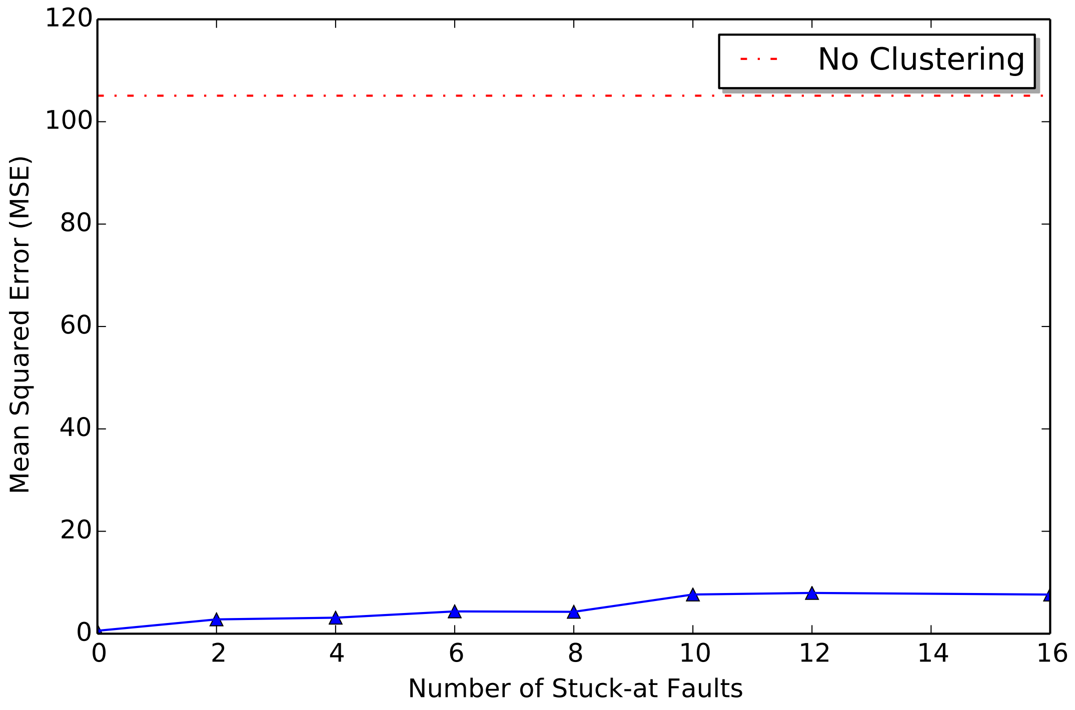


Figure 5.3: The output quality of clustering using DPMM in the presence of different numbers of stuck-at faults. *No Clustering* represents the MSE at the initial iteration of the algorithm when all input data points are considered to be in one cluster. We observe that the application can tolerate multiple stuck-at faults in the sampler and performs clustering even in the presence of more than 5 stuck-at faults.

## 5.2 Robustness to Transient Faults

To evaluate the robustness of the sampler to transient faults, we performed sampling under different gate level transient fault rates. For each transient fault rate, we inject faults at that rate in the output of all gates in the design. Figure 5.4 presents the KL divergence observed for one particular input (with entropy around 2) at different transient fault rates. We report the mean of 100 independent experiments for each fault rate. We observe that the KL divergence remains low ($< 0.1$) even at fault rates as high as $1.2 \times 10^{-2}$. Also, there is gradual degradation in the quality of output with increasing fault rate.

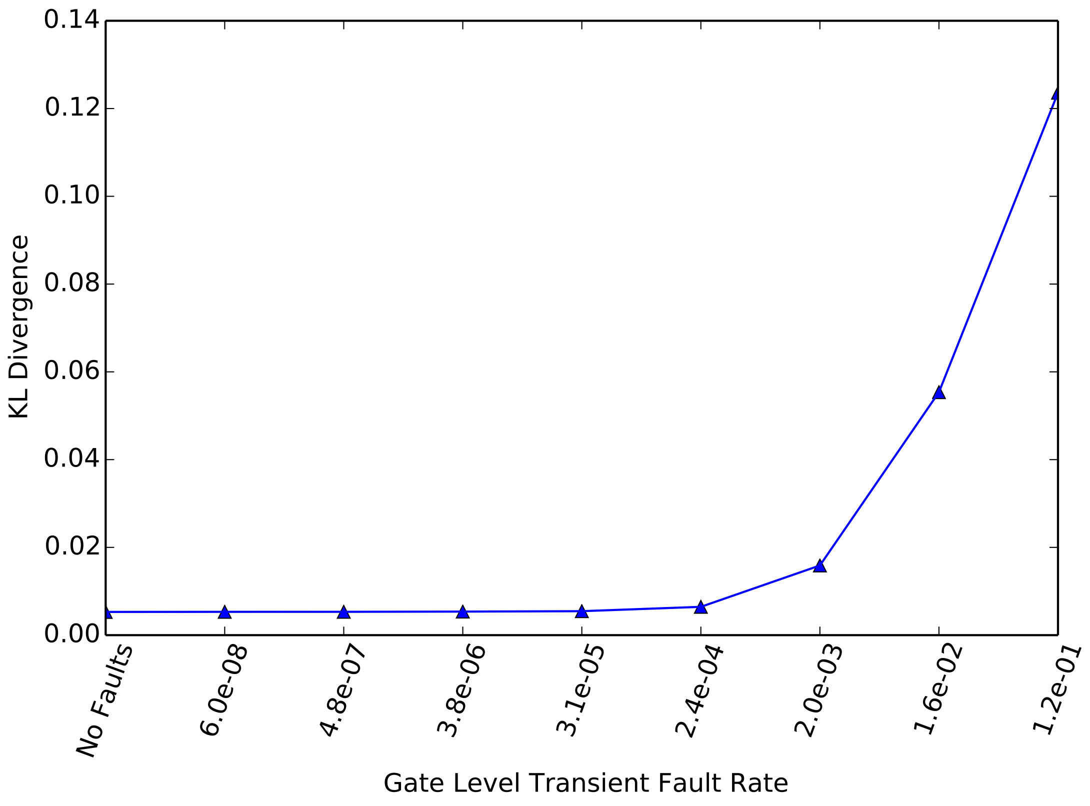


Figure 5.4: The output quality of the sampler for different gate level transient fault rates. We observe that the KL divergence remains low ($< 0.1$) even at fault rates as high as $1.2 \times 10^{-2}$.

Figure 5.5 presents the output quality of particle filtering at different gate level transient fault rates. We observe that the application can tolerate transient faults with rates as high as $2 \times 10^{-3}$ and performs meaningful filtering even at that fault rate.

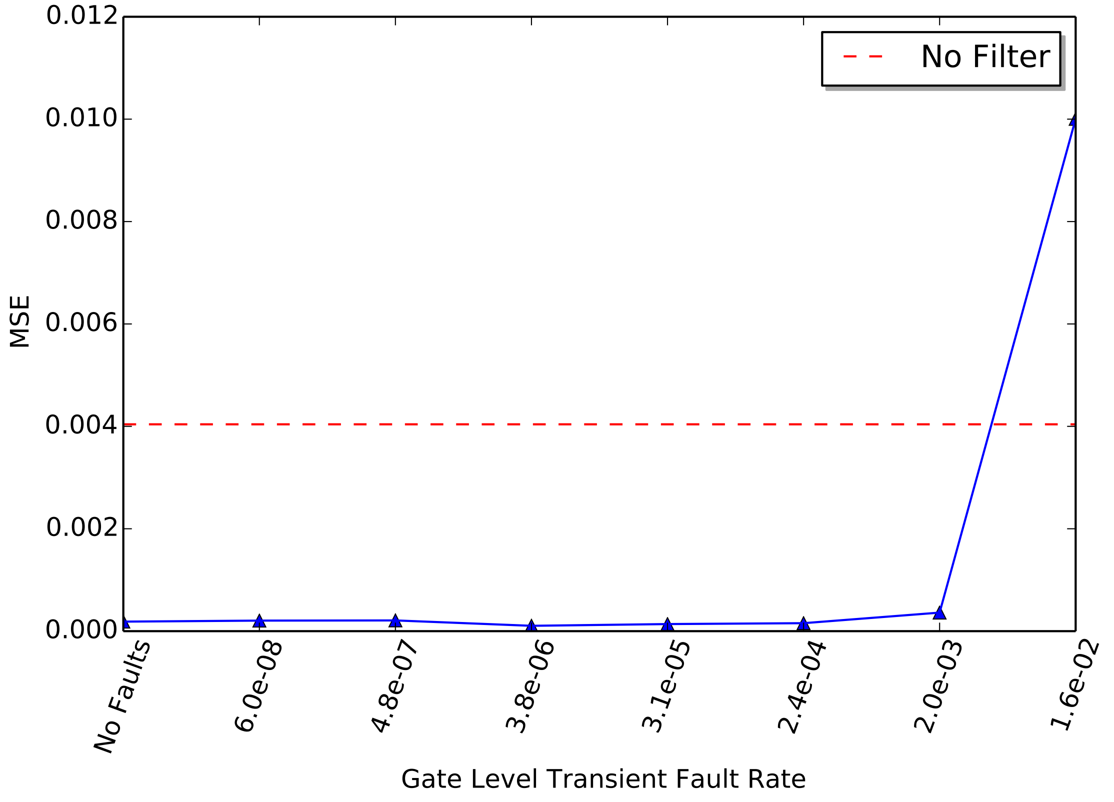


Figure 5.5: The output quality of particle filtering for different gate level transient fault rates. *No Filter* represents the MSE of using the noisy observations as the estimate of the state of the system. We observe that the KL divergence remains low ($< 0.1$) even at fault rates as high as $1.2 \times 10^{-2}$.

Figure 5.6 presents the output quality of clustering using DPMM at different gate level transient fault rates. The dotted line represents the output quality if all data points were assumed to be in the same cluster (as in the starting point of the algorithm). We observe that the application can tolerate transient faults with rates as high as $2.4 \times 10^{-4}$ and performs meaningful clustering even at that fault rate.

These results indicate that the hardware samplers indeed exhibit tolerance to transient faults at high error rates and applications using the sampler can still produce meaningful outputs. The fault rates we experimented with are high compared to transient fault rates in present systems. For example, the 104K node BlueGene/L system at Lawrence Livermore National Laboratory experiences a soft error in the cache once every five hours [32]. Our fault rates are of the order of 1 every 1,000-10,000 clock cycles and in every gate.

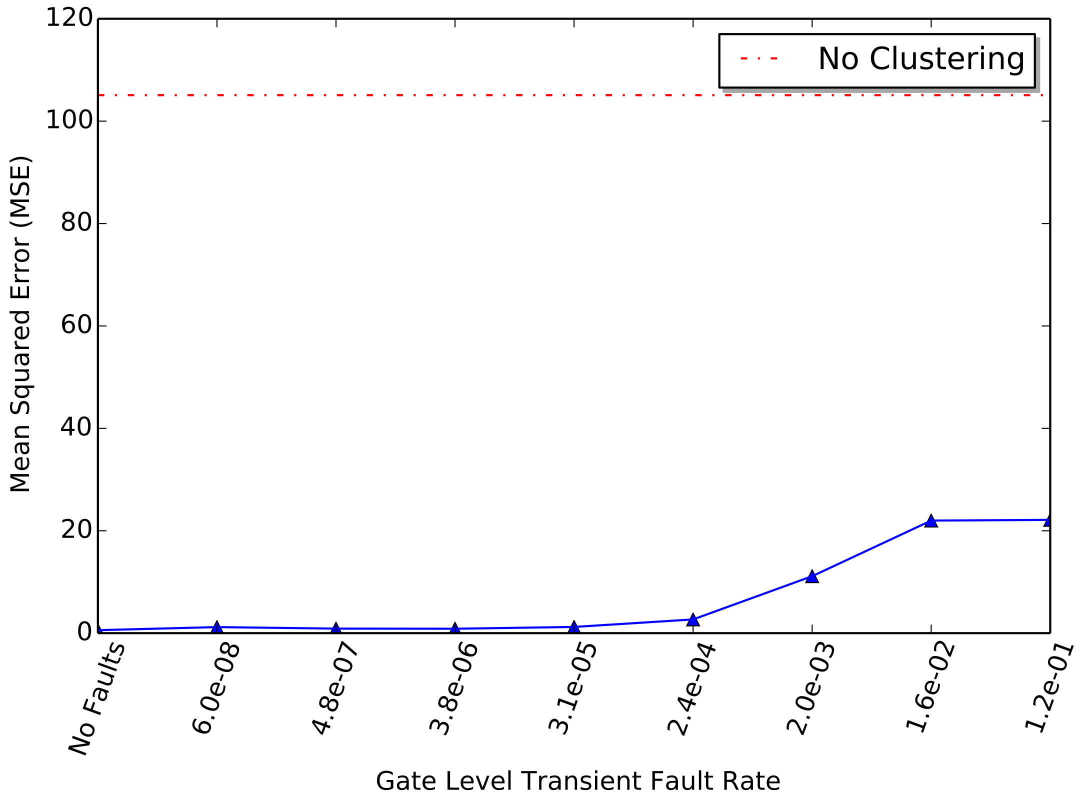


Figure 5.6: The output quality of clustering using DPMM for different gate level transient fault rates. *No Clustering* represents the MSE at the initial iteration of the algorithm when all input data points are considered to be in one cluster. We observe that the application can tolerate transient faults with rates as high as $2.4 \times 10^{-4}$ and performs meaningful clustering even at that fault rate.

## 5.3 Distribution of Critical Gates

We are interested in knowing what fraction of gates in the design cannot tolerate faults (we call them *critical gates*). To determine the fraction of critical gates, we systemically injected stuck-at faults at all 3,656 possible locations one location at a time. We then looked at the resulting quality of the samplers output (in terms of KL divergence) for faults in each gate. Figure 5.7 shows, for different KL divergence levels, the percentage of gates that would be deemed critical for that level. We observe that less than 20% of the gates would be critical for an acceptable KL divergence level of 0.1.

Faults in the other 80% of the gates do not increase the KL divergence to more than 0.1 for this particular input distribution.

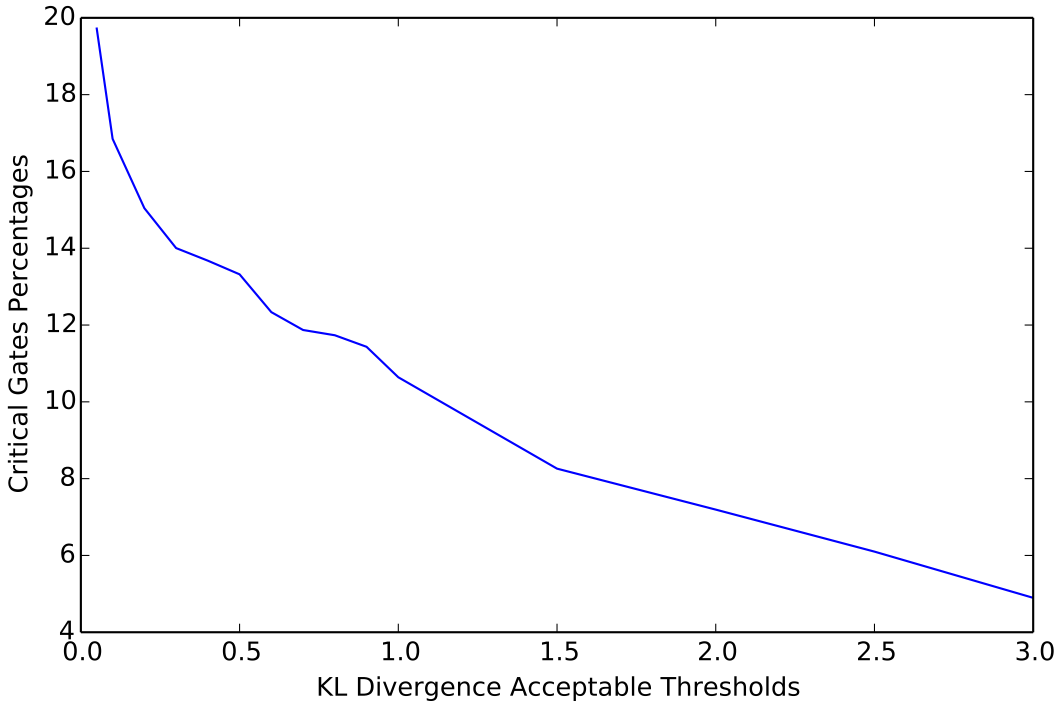
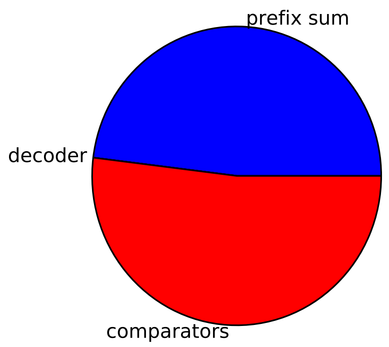


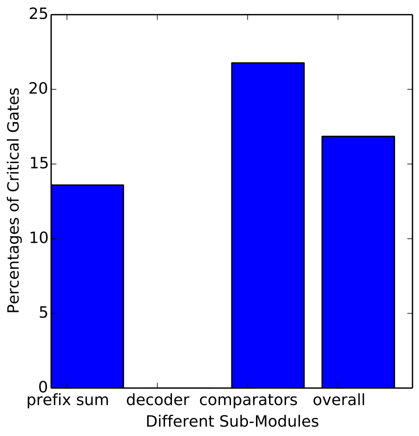
Figure 5.7: The percentages of gates that would be deemed critical for different KL divergence levels that can be tolerated by an application using the sampler. We observe that less than 20% of the gates would be critical for an acceptable KL divergence level of 0.1.

The distribution of critical gates among different sub-blocks is shown in Figure 5.8 for an acceptable KL divergence threshold of 0.1. We observe that critical gates are distributed almost equally between the prefix-sum and comparator blocks (Figure 5.8a). Also, 13%, 21%, and 17% of gates in the prefix-sum, comparators and the entire sampler, respectively, are critical (Figure 5.8b).

The preceding results are for one input distribution. We expect the number of critical gates at the application level to be lower because a particular gate might be critical for one input but not for another. When the application uses the sampler with a faulty gate, it uses the sampler with several different inputs throughout the application. Only for some of those inputs will that particular gate be critical and as such, it might end up not showing any significant impact on the final application output quality. Of course, we



(a)



(b)

Figure 5.8: Distribution of critical gates for an acceptable KL divergence of 0.1. (a) The number of critical gates in different sub-blocks. We observe that the critical gates are distributed almost equally between the prefix sum block and the comparators. (b) The percentage of critical gates in different sub-blocks. We observe that the entire sampler has around 4% critical gates.

expect that there would be some gates that would be critical for a large number of inputs and would indeed affect the final application output quality.

To find the number of such critical gates at the application level, we repeated the same experiment for particle filtering and clustering using DPMM. For particle filtering, we injected stuck-at faults into half of the 3,656 gates (randomly chosen) one at a time and observed the impact on the output quality (in terms of MSE) of the particle filter. To determine an acceptable output quality threshold, we ran the application 100 times without any faults and observed the MSE. The highest MSE observed was $1.86 \times 10^{-3}$ or a $log_{10}$MSE of $-2.73$. We used this MSE as our threshold. We consider a gate to be critical if a fault in that gate results in a mean MSE (over 5 experiments) higher than this threshold. Figure 5.9 shows the results of our experiment. We observed that less than 4% of the gates were critical. This is significantly lower than the 17% estimate we had from Figure 5.8b.

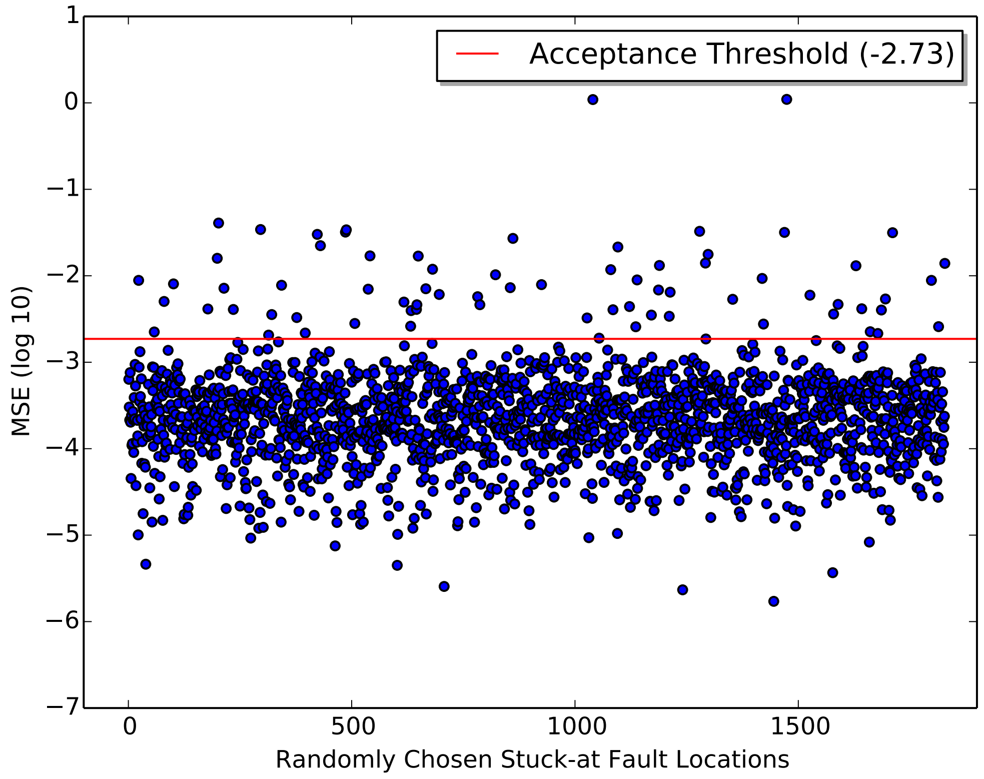
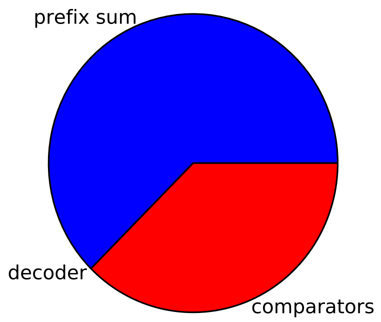


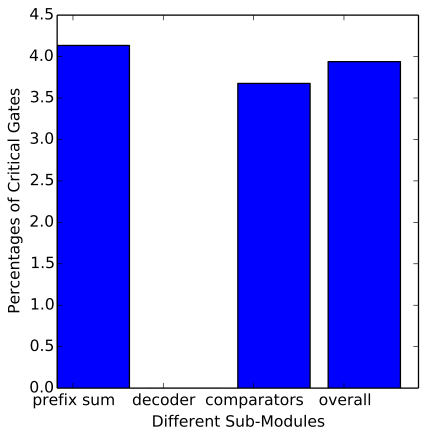
Figure 5.9: The output quality of particle filtering under stuck-at faults in 1,828 (50%) randomly chosen gates injected one at a time. The acceptability threshold was chosen by running the application without any faults 100 times and taking the worst MSE that was observed. We observed that less than 4% of the gates were critical.

The distribution of critical gates among different sub-blocks is shown in Figure 5.10 in the context of particle filtering. We observe that the prefix sum block contains 62.5% of the critical gates (Figure 5.10a). Also, the percentages of critical gates in the prefix-sum, comparators, and the entire sampler are all less than 4.2% (Figure 5.10b).

For clustering using DPMM, we injected stuck-at faults into 15% of the 3,656 gates (randomly chosen) one at a time and observed the impact on the output quality (in terms of MSE) of clustering. To determine an acceptable output quality threshold, we ran the application 100 times without any faults and observed the MSE. The highest MSE observed was 3.90. We used this MSE as our threshold. We consider a gate to be critical if a fault in that gate results in a mean MSE (over 5 experiments) higher than this threshold.



(a)



(b)

Figure 5.10: Distribution of critical gates at the application level for particle filtering. (a) The number of critical gates in different sub-blocks. We observe that the prefix sum block contains 62.5% of the critical gates. (b) The percentage of critical gates in different sub-blocks. We observe that the entire sampler has less than 4% critical gates.

Figure 5.11 shows the results of our experiment. We observed that 5.1% of the gates were critical. This is significantly lower than the 17% estimate we had from Figure 5.8b.

The distribution of critical gates among different sub-blocks is shown in Figure 5.12 in the context of clustering using DPMM. We observe that the prefix sum block contains 60.7% of the critical gates (Figure 5.12a). Also, the percentages of critical gates in the prefix-sum, comparators, and the entire sampler are all less than 5.2% (Figure 5.12b).

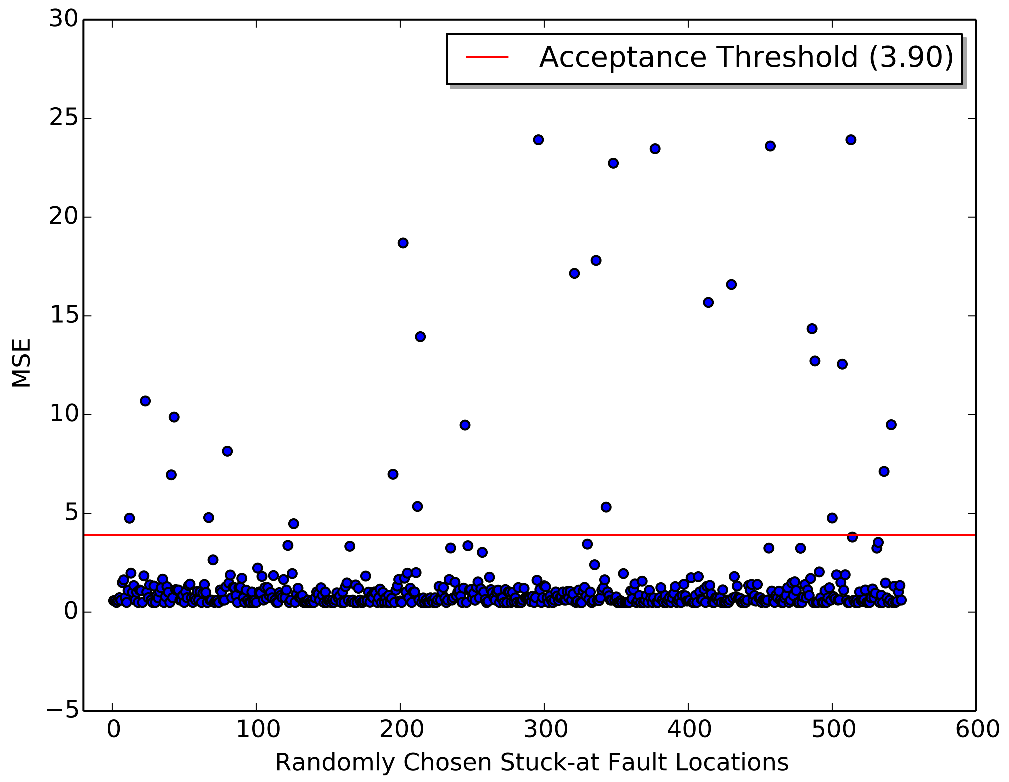
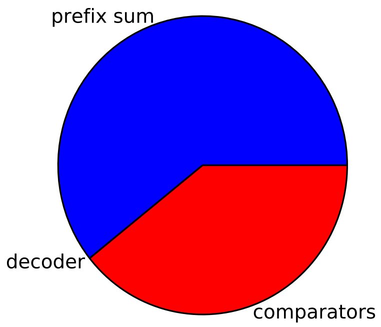
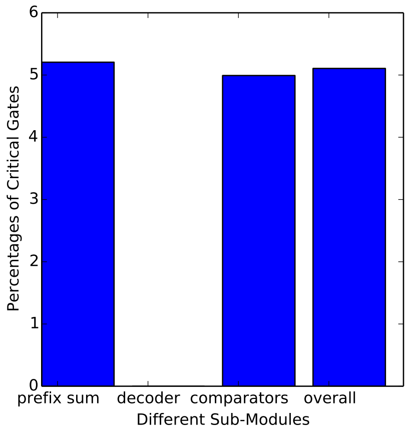


Figure 5.11: The output quality of clustering using DPMM under stuck-at faults in 548 (15%) randomly chosen gates injected one at a time. The acceptability threshold was chosen by running the application without any faults 100 times and taking the worst MSE that was observed. We observed that 5.1% of the gates were critical.

The above results suggest that since the percentage of critical gates in the design is low, it could be possible to pay the extra cost (in terms of area or power) to make these gates more fault tolerant while implementing the rest of the gates with more energy efficient, but fault prone, devices. These critical gates could be made more fault tolerant in either of two ways: by implementing them with reliable devices or by adding redundancy for these gates.



(a)



(b)

Figure 5.12: Distribution of critical gates at the application level for clustering using DPMM. (a) The number of critical gates in different sub-blocks. We observe that the prefix sum block contains 60.7% of the critical gates. (b) The percentage of critical gates in different sub-blocks. We observe that the entire sampler has 5.1% critical gates.

# CHAPTER 6

# RELATED WORK

Sampling in hardware has been studied before. Prior work has proposed using hardware samplers as a hardware primitive towards building probabilistic computing systems for Bayesian inference [5]. Prior work also has demonstrated using multiple Bayesian inference applications that sampling based hardware implementations can have significant performance advantages over software implementations of such applications [6]. These works, however, do not study the effect of hardware faults and do not consider the possibility of implementing the samplers using more energy efficient, but fault prone, devices.

The closest prior work to our work was by Deka et al. [33]. That work showed using several applications that sampling based applications could be tolerant to errors at the output of the sampler. However, it does not take into account the relationship between hardware faults and the errors at the output of the sampler. In our work, we perform fault injections at the hardware level and study the effect of hardware faults at the output of the sampler as well as the impact these errors have on the output of the application.

While our work is focused on evaluating the fault tolerance of a hardware sampler, there has been prior work studying the fault tolerance of other circuits and accelerators. For example, prior work has demonstrated the stuck-at fault tolerance of a hardware accelerator based on artificial neural networks [4].

# CHAPTER 7

# CONCLUSION

In this research, we evaluated the robustness of hardware samplers to hardware faults. Our work was motivated by the facts that (a) several important applications use sampling, (b) sampling in hardware could have energy-efficiency and performance benefits, and (c) hardware devices are increasingly becoming more error prone. We carried out evaluations of the robustness of a hardware sampler using stuck-at and transient fault models at the gate level. To understand the application level implications of errors made by the sampler due to hardware faults, we studied its impact on two applications: particle filtering and clustering using a Dirichlet Process Mixture Model (DPMM). Our results demonstrate that the hardware sampler is indeed robust to hardware faults and its robustness improves in the context of end to end applications. Specifically, we observed that (a) the applications can tolerate multiple stuck-at faults in the sampler ( $> 5$ faults at the same time), (b) the applications can tolerate gate level transient fault rates as high as $2.4 \times 10^{-4}$, and (c) only faults in a small number of gates ($< 5.2\%$) were found to affect the output quality of these applications. The results show that there may be significant energy benefits from leveraging this robustness to implement sampling based applications.

# REFERENCES

[1] C. M. Bishop et al., *Pattern recognition and machine learning.* Springer New York, 2006, vol. 1.

[2] 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.

[3] P. Dubey, "Recognition, mining and synthesis moves computers to the era of tera," *Technology@ Intel Magazine*, pp. 1–10, February 2005.

[4] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, 2012, pp. 356–367.

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

[6] V. Mansinghka and E. Jonas, "Building fast Bayesian computing machines out of intentionally stochastic, digital parts," *arXiv preprint arXiv:1402.4914*.

[7] S. Borkar, "Design perspectives on 22nm CMOS and beyond," in *Proceedings of the 46th Annual Design Automation Conference.* ACM, 2009, pp. 93–94.

[8] M. Alam, "Reliability-and process-variation aware design of integrated circuits," *Microelectronics Reliability*, vol. 48, no. 8, pp. 1114–1122, 2008.

[9] H. Wei, M. Shulaker, G. Hills, H.-Y. Chen, C.-S. Lee, L. Liyanage, J. Zhang, H.-S. Wong, and S. Mitra, "Carbon nanotube circuits: Opportunities and challenges," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 619–624.

[10] N. R. Shanbhag, R. A. Abdallah, R. Kumar, and D. L. Jones, "Stochastic computation," in *Proceedings of the 47th Design Automation Conference.* ACM, 2010, pp. 859–864.

[11] M. de Kruijf, S. Nomura, and K. Sankaralingam, “Relax: An architectural framework for software recovery of hardware faults,” in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 497–508.

[12] A. Athalye, M. Bolić, S. Hong, and P. M. Djurić, “Generic hardware architectures for sampling and resampling in particle filters,” *EURASIP Journal on Advances in Signal Processing*, vol. 2005, no. 17, pp. 2888–2902, 1900.

[13] A. Sankaranarayanan, R. Chellappa, and A. Srivastava, “Algorithmic and architectural design methodology for particle filters in hardware,” in *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, Oct 2005, pp. 275–280.

[14] M. Bolic, “Architectures for efficient implementation of particle filters,” Ph.D. dissertation, State University of New York at Stony Brook, 2004.

[15] L. Devroye, “Sample-based non-uniform random variate generation,” in *Proceedings of the 18th conference on Winter simulation.* ACM, 1986, pp. 260–265.

[16] G. Marsaglia, “Xorshift RNGs,” *Journal of Statistical Software*, vol. 8, no. i14.

[17] S. Vigna, “Further scramblings of Marsaglia’s xorshift generators,” *arXiv preprint arXiv:1404.0390*, 2014.

[18] S. Kullback, *Information theory and statistics.* Courier Dover Publications, 2012.

[19] H. E. Rauch, C. Striebel, and F. Tung, “Maximum likelihood estimates of linear dynamic systems,” *AIAA journal*, vol. 3, no. 8, pp. 1445–1450, 1965.

[20] G. Welch and G. Bishop, “An introduction to the Kalman filter,” Chapel Hill, NC, USA, Tech. Rep., 1995.

[21] A. Doucet and A. M. Johansen, “A tutorial on particle filtering and smoothing: Fifteen years later.”

[22] M. Isard and A. Blake, “Condensationconditional density propagation for visual tracking,” *International journal of computer vision*, vol. 29, no. 1, pp. 5–28, 1998.

[23] C. E. Rasmussen, “The infinite Gaussian mixture model,” in *In Advances in Neural Information Processing Systems 12.* MIT Press, 2000, pp. 554–560.

[24] E. B. Sudderth, "Graphical models for visual object recognition and tracking," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

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

[26] 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.

[27] "Zedboard." [Online]. Available: http://www.zedboard.org/

[28] "Xilinx Vivado Design Suite." [Online]. Available: http://www.xilinx.com/products/design-tools/vivado/

[29] "Xilinx Software Development Kit." [Online]. Available: http://www.xilinx.com/tools/sdk.htm

[30] P. Kurup, *Logic synthesis using Synopsys®*. Springer, 1997.

[31] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework," in *Computer Design, 2008. ICCD 2008. IEEE International Conference on.* IEEE, 2008, pp. 363–370.

[32] J. N. Glosli, D. F. Richards, K. Caspersen, R. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing.* ACM, 2007, p. 58.

[33] B. Deka, A. A. Birklykke, H. Duwe, V. K. Mansinghka, and R. Kumar, "Markov chain algorithms: A template for building future robust low power systems," in *Asilomar Conference on Signals, Systems, and Computers*, 2013.