A Numerical Optimization-based Methodology for Application Robustification: Transforming Applications for Error Tolerance

Joseph Sloan, David Kesler, Rakesh Kumar
University of Illinois
Urbana-Champaign
jsloan,dkesler2,rakeshk@illinois.edu

Ali Rahimi
Intel Labs
Berkeley
ali.rahimi@intel.com

ABSTRACT

There have been several attempts at correcting process variation induced errors by identifying and masking these errors at the circuit and architecture level [10, 27]. These approaches take up valuable die area and power on the chip. As an alternative, we explore the feasibility of an approach that allows these errors to occur freely, and handle them in software, at the algorithmic level. In this paper, we present a general approach to converting applications into an error tolerant form by recasting these applications as numerical optimization problems, which can then be solved reliably via stochastic optimization. We evaluate the potential robustness and energy benefits of the proposed approach using an FPGA-based framework that emulates timing errors in the floating point unit (FPU) of a Leon3 processor [11]. We show that stochastic versions of applications have the potential to produce good quality outputs in the face of timing errors under certain assumptions. We also show that good quality results are possible for both intrinsically robust algorithms as well as fragile applications under these assumptions.

1. INTRODUCTION

Power has been, for some time now, a first order design constraint for microprocessors [1]. In fact, performance, yield, and functionality are routinely sacrificed for power considerations today [17, 24].

An important reason why modern microprocessors consume a significant amount of power is that they are often designed conservatively (i.e., are *guardbanded*) to allow correct operation under the worst-case manufacturing and environmental conditions [7]. The power cost of conservative design is high and is only increasing with increasing process variation in the current CMOS and post-CMOS technologies. Applying a power reduction technique such as voltage scaling reduces power, but the benefits continue to be limited by the inherently conservative nature of the baseline worst-case design [29].

Some recent proposals [10] have advocated reducing processor power by eliminating design-level guardbands against worst-case conditions. Processors without design-level guardbands consume lower power than their counterparts designed for the worst-case. However, such processors may be unreliable once the voltage is reduced below a certain threshold. Unreliability is due to the possibility of timing errors induced by process variation and environmental fluctuations.

Previous proposals largely employ hardware-based mechanisms to detect and correct variation-induced errors in processors with reduced guardbands. These mechanisms often rely on temporal or spatial redundancy to detect and correct the errors. Hardware-based mechanisms to detect and correct variation-induced timing errors have associated area and power costs. Costs may be especially prohibitive in the face of drastic reduction in the supply voltage [10, 25].

In this paper, we explore the feasibility of an approach that allows these errors to occur freely, and handle them in software, at the algorithmic level. (Figure 1) An algorithmic approach for error correction would allow us to eliminate or minimize the area and power cost of lower-level hardware-based mechanisms to detect and correct errors by replacing the original computation with one that may take slightly longer to complete. The approach presented in this paper consists of reformulating applications as stochastic optimization problems. In the last thirty years, the machine learning and numerical optimization community has produced and analyzed many successful stochastic optimization procedures and online learning algorithms for solving large-scale learning problems (see [21, 8, 26] for surveys). We propose an entirely different application for stochastic optimization: a generic engine for building robust applications on processors that produce variation-induced errors. Unlike the traditional setting for stochastic gradient descent, where stochasticity arises because the gradient direction is computed from a random subset of a dataset, here the processor itself is the source of stochasticity. We call our approach *application robustification*.

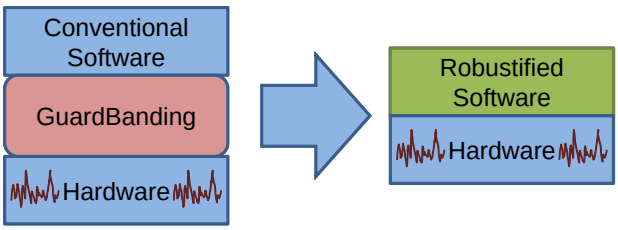

Figure 1: The traditional approach to dealing with hardware uncertainties is through guradbanding. We allow hardware errors to be exposed to software which is robustified to tolerate these errors.

As a specific instance of the proposed approach, we show that it is possible, under certain optimistic assumptions, to robustify a large class of important, common applications against timing errors that occur in the numerical units of voltage overscaled processors. For example, solving Least Squares problems or finding eigenvalues of a matrix can be readily cast in a variational form. Similarly many combinatorial problems such as sorting an array of numbers, finding a minimum cut, a maximum flow, shortest distances, or a matching in a graph can also be cast into variational form. To solve such problems on a stochastically correct processor (*stochastic processor*) [19, 3, 20, 4], we express them as constrained optimization problems, mechanically convert these to an unconstrained exact penalty form, and then solve them using stochastic gradient descent and conjugate gradient algorithms. When the source of unreliability in the processor is stochastic, existing theoretical results on the convergence rate and robustness of stochastic gradient optimization carry over directly to this setting. This approach is quite generic, since linear programming, which is P-complete, can be implemented this way. In fact, we present examples of robustification under optimistic assumptions for both applications for which precise outputs are typically required (fragile applications), e.g., *sorting*, etc., as well as the ones for which small errors in the output are typically acceptable (intrinsically robust applications), e.g., *IIR filters*, etc.

Note that this paper explores only the potential upside of the proposed approach. Several simplifying assumptions have been made (as discussed in Section 6 and throughout the paper). Future work will continue to evaluate and mitigate the costs.

Our contributions in this paper are as follows:

- We present a general approach for converting applications into a form that may be robust to variation-induced errors. Our approach is applicable, under certain optimistic assumptions, to all applications that can be mapped into a stochastic optimization problem. To the best of our knowledge, this is the first work on a generic methodology to transform application code for error tolerance that may work for both fragile and intrinsically robust applications.
- We develop an FPGA-based framework for evaluating the potential robustness benefits of the proposed approach. Our FPGA-based framework emulates timing errors in the floating point unit (FPU) of a Leon3 [11] processor. We show through our experiments that stochastic versions of applications can produce good quality outputs in the face of errors under certain assumptions. We also show that good quality results are possible for both intrinsically robust algorithms as well as fragile applications.
- We demonstrate that there is a real need to develop optimization-based code transformation methodologies that address the processor as a new source of stochasticity. Writing stochastic versions of applications may become a necessity for future CMOS and post-CMOS computing due to increasing variation. Our future work will evaluate and mitigate the costs of the proposed approach.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 summarizes known properties of stochastic gradient solvers and illustrates a generic framework for implementing robust applications using modified gradient descent. Section 4 presents four examples of converting an application into its robust, stochastic form. Section 5 presents our methodology and results. Section 6 discusses the limitations of the proposed methodology and future work. Section 7 concludes.

2. RELATED WORK

Recently proposed stochastic processor designs [19, 3, 20, 4] aim to fundamentally re-think the software / hardware interface by allowing hardware to produce errors even during nominal operation. The numerical algorithmic techniques presented in this paper represent one method of addressing errors in stochastic processors by handling them at the application level.

Some past works have also addressed error tolerance at the algorithmic level. Algorithmic noise tolerance [14] is a technique for DSPs in which voltage overscaling is employed to reduce power consumption and knowledge of the DSP's transfer function and input/output characteristics are used to tolerate errors that occur. Error resilient system architectures [5] target probabilistic algorithms and use a large pool of unreliable, power-efficient computing resources as the main workhorse, while a smaller set of reliable resources is used to deal with errors and ensure that computations are completed. Algorithm-based fault tolerance [16] addresses errors at the algorithmic level by encoding input data with supplemental checksums, modifying algorithms to produce the encoding for the output data, and using the encoded data to detect and correct errors when possible. In

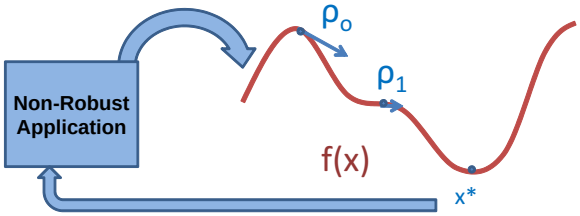

Figure 2: Application Robustification involves converting an application to an unconstrained optimization problem, where the minimum corresponds to the output of the original non-robust application.

contrast to the above approaches that are limited by their application-specific nature, the methodology that we present in this paper for application-level error tolerance is generic and can potentially drive a large class of important applications on stochastic processors.

3. PROPOSED APPROACH

Our goal is to recast a given problem into an equivalent numerical problem that can tolerate noise in the FPU, and whose solution encodes the solution to the equivalent problem. Let the vector x^* denote the (unknown) solution to our problem. To devise a robust algorithm, we construct a cost function f whose minimum is attained at x^*. Solving the problem then amounts to minimizing f. The main challenges, as illustrated in Figure 2:

- How to construct f without knowing the actual value of x^* a priori?
- How to choose an optimization engine that converges quickly and tolerates CPU noise?

Since the selection of the minimization function can often depend on the optimization engine, we first detail the choice of our optimization engine.

3.1 Stochastic Solvers for Constrained Optimization

Under mild conditions, as long as step sizes are chosen carefully, gradient descent converges to a local optimum of the cost function even when the gradient is known only approximately. For this reason, we rely on gradient descent as the primary optimization engine to construct algorithms that tolerate noise in the CPU's numerical units. To minimize a cost function $f : \mathbf{R}^d \to \mathbf{R}$, gradient descent generates a sequence of steps $x^1 \ldots x^i \in \mathbf{R}^d$ via the iteration

$$x^i \leftarrow x^{i-1} + \lambda^i \nabla f(x^{i-1}), \tag{1}$$

starting with a given initial iterate $x0 \in \mathbf{R}^d$. The vector $\nabla f(x^{i-1})$ is a subgradient of f at x^{i-1}, and the positive scalar λ^i is a step size that may vary from iteration to iteration. The goal is for the sequence of iterates to converge to a local optimizer, x^*, of f.

The bulk of the computation in gradient descent is in computing the gradient ∇f. There may be variation-induced errors while computing ∇f. We denote the resulting noisy gradient by $\nabla f(x^{i-1}; \xi^i)$, with ξ^i denoting a random variable independent of x^{i-1}. The remaining operations, including computing the step size, updating x^i with the step, and testing for convergence, are assumed to be carried out reliably as they are critical for convergence. Thankfully, these steps require relatively little computation and can be robustified at a small cost (e.g., increasing the voltage during these steps, software-level redundancy, etc.).

The suitability of gradient descent for processors with reduced guardbands is due to the fact that under various assumptions of local convexity on f, x^i is known to approach the true optimum as iterations progress. The following theorem is distilled from [21], but variants of these results have appeared throughout the literature (for example, [8, 26, 31] and references therein)

THEOREM 1. *Let x^* be a minimizer of f. Suppose that the noisy subgradient, $\nabla f(x; \xi)$ is unbiased ($\mathbb{E}_\xi \nabla f(x; \xi) = \nabla f(x)$), and has bounded variance ($\mathbb{E}_\xi \|\nabla f(x, \xi)\|^2 < M^2$ for some scalar $M > 0$).*

If f is convex, lower-semicontinuous, and the step sizes obey $\lambda^i = O(1/\sqrt{i})$, then the iterates of (1) satisfy

$$\mathbb{E}\, f(x^i) - f(x^*) = O\left(i^{-1/2}\right). \tag{2}$$

If f is c-strongly convex and L-Lipschitz, and the step sizes obey $\lambda^i = O(1/i)$, then the iterates of (1) satisfy

$$\mathbb{E}\, f(x^i) - f(x^*) = O\left(\tfrac{LM^2}{c^2} \cdot i^{-1}\right). \tag{3}$$

The expectation in both cases is over the sequence of $\xi_1, \ldots, \xi_i$.

Thus, not only is the correct answer recovered almost surely, but each additional iteration improves its accuracy *beyond* the precision of the subgradient. That is, even if the CPU approximates ∇f to only a few bits of precision, as long as the approximation is unbiased, gradient descent can eventually extract a solution with arbitrarily high accuracy. Therefore we get for free the benefit of additional iterative refinement techniques [15] that are typically used to improve the accuracy of numerical algorithms on today's processors. The robustness of gradient descent makes it an attractive choice as the computational back-end for solving the optimization problems.

For some applications, the natural conversion is to a constrained variational form

$$\underset{x \in \mathbf{R}^d}{\text{minimize}} \quad f(x) \tag{4}$$

$$\text{s.t. } g(x) \leq 0, \tag{5}$$

$$h(x) = 0 \tag{6}$$

for some functions f, g, and h. Constrained versions of gradient descent in the stochastic setting have been previously analyzed [21]. These methods typically involve projecting the gradient or the iterate on the feasible set after each iteration. This step can be quite expensive, as it typically involves solving at least a Least Squares problem. Interior point methods based on log-barrier/Newton steps [30] are ostensibly promising, but in practice, they require computing an Newton-step, which wipes out any potential power benefits. Instead, we rely on an exact penalty method to convert constrained problems into unconstrained problems that can be solved by gradient descent. The following result is distilled from [6] (mainly Proposition 5.5.2, and folding in Linear Independence Constraint Qualification (LICQ) conditions on g and h):

THEOREM 2. *Let x^* be a unique optimizer of (4), with g and h both affine linearly independent functions of x. Then there exists $\mu_0 > 0$ so that for every $\mu > \mu_0$, x^* also minimizes*

$$f(x) + \mu \sum_i |h_i(x)| + \mu \sum_j [g_j(x)]_+ . \qquad (7)$$

The operator $[\cdot]_+ = \max(0, \cdot)$ returns its argument if it is positive, and zero otherwise. A similar result for quadratic exact penalty functions of the form $f(x) + \mu \sum_i h_i 2(x) + \mu \sum_j [g_j(x)]_+^2$ also hold [23]. This theorem states that a constrained optimization problem of the form (4) can be converted into an unconstrained form (7) by penalizing constraint violations in the objective function.

3.2 Variants on Gradient Descent

As Theorems 1 and 2 show, the actual rate of convergence depends on several factors including the modulus of convexity c of the minimization function f, and the size of each step taken. For example, if the objective function has low modulus of convexity (a property called ill-conditioning), the gradient search direction can converge arbitrarily slowly, instead bouncing around in directions perpendicular toward that of the minimum. To alleviate some of the artifacts, we can add *momentum* to the search direction using the update rule:

$$x^i \leftarrow x^{i-1} + \lambda^i d^i \qquad (8)$$

$$d^i \leftarrow \alpha \nabla f(x^{i-1}) + (1 - \alpha) d^{i-1} \qquad (9)$$

This modified direction essentially becomes a smoothed running average of the recent directions/gradients, and the scalar α controls the amount of smoothing in the search direction. Adding momentum provides two benefits. If the gradient is pointing in a similar direction for multiple consecutive iterations then it is likely to continue in that direction in the next few iterations. In that case, the momentum is built up causing the descent to move faster along this direction. On the other hand, if the gradient is oscillating between two different directions from iteration to iteration, then the momentum helps to dampen the oscillations, and points the search direction towards the direction of progress.

Similarly, different step sizes may work better for different applications when performing gradient search. Scaling the step size as $\frac{1}{i}$, where i is the number of iterations, may make the step size too small in later iterations, making it difficult for the search to converge. Scaling it as $\frac{1}{\sqrt{i}}$ allows the step size to remain larger while still causing it to continuously decrease. We also examined using a fixed number of iterations, followed by a period of variable stepsizing. We refer to this technique as *aggressive stepping*. In the phase of variable step sizing, the step size is increased by a factor $\beta_{success}$ every time the step causes the cost function to decrease. On the other hand, the step size is decreased by a factor β_{fail} every time the last move caused the cost function to increase. The phase continues until the percent change between two consecutive steps drops below a threshold.

Finally, while gradient descent on a convex function is guaranteed to make progress, it is possible to construct a function where this progress is arbitrarily slow. Consider, for example, an elongated quadratic valley. The gradient descent direction is generally a poor direction for this type of function and other ill-conditioned problems, because it doesn't point toward the minimum. *Preconditioning* fixes this problem with gradient descent by reshaping the cost function. Given the cost function $f(x)$, we minimize instead a new function $g(y) = f(Ay)$. We chose the matrix A so that $g(y)$ is better conditioned, i.e. looks more like a bowl than a valley. Once we have the optimum y^*, we can then recover x^* via the relation $x = Ay$.

3.3 Conjugate Gradient

While we use gradient descent as a search strategy for most of our kernels, some kernels may warrant other search strategies. For example, with a Least Squares problem, discussed in the following section, the structure of the problem can be exploited to construct better search directions and step sizes. One approach, typically reserved for very large problems, is the conjugate gradient (CG) method [13]. The method examines the gradients of the cost function to construct a sequence of search directions that are mutually conjugate to each other (i.e. where two search direction p_i and p_j satisfy $p_i^T A p_j = 0, \quad \forall i \neq j$ for a particular matrix A). On a reliable processor, when CG is applied to a Least Squares problem, it is guaranteed to converge in at most n iterations (where n is the number of variables to solve for in the Least Squares problem). The convergence of CG when the gradient directions are noisy is also well-understood [28]. To reduce the effect of noisy gradients,

our implementation of CG resets the search direction after every few iterations.

4. APPLICATION TRANSFORMATION FOR ROBUSTNESS

How to transform a given problem into its variational form (4) is often immediate from the definition of the problem. For example, the least-squares problem is already defined as an optimization problem. Otherwise, the post-condition of the problem can often be converted into a cost function whose optimum solves the problem illustrated by the IIR example below. Once converted into a variational form, any optimization technique that is robust to numerical noise, such as the ones described above, can be used to find a solution to the problem. We provide several illustrative examples below.

Least Squares.

Given a matrix A and a column vector b of the same height, a fundamental problem in numerical linear algebra is to find a column vector x that minimizes the norm of the residual $Ax - b$. This problem is typically implemented on current CPUs via the SVD or the QR decomposition of A. In Section 5 we show that these algorithms are disastrously unstable under numerical noise, but that minimizing $f(x) = \|Ax - b\|^2 = x^\top A^\top Ax - 2b^\top x + b^\top b$ by gradient descent tolerates numerical noise well. The gradient in this case is $\nabla f(x) = A^\top(Ax - b)$.

IIR filters.

Filtering a signal with an Infinite Impulse Response (IIR) filter is a basic operation in signal processing. The problem is naturally defined as passing an input signal $u[t]$ through a rational transfer function $H(z) = \frac{\sum_{i=0}^{n} a_i z^{-i}}{\sum_{i=0}^{m} b_i z^{-i}}$ to obtain the desired output $x[t]$. It is typically implemented on current CPUs by the feed-forward recursion:

$$x[t] = \frac{1}{b_0}\left(\sum_{i=0}^{n} a_i u[t-i] - \sum_{i=1}^{m} b_i x[t-i]\right)$$

On a stochastic processor, this recursive implementation accrues noise in x as t grows. To recast this variationally, observe that the output signal x must satisfy the post-condition $\sum_{i=0}^{m} b_i x[t-i] = \sum_{i=0}^{n} a_i u[t-i]$ for all t, or in matrix form, $Bx = Au$, where the matrices A and B are banded diagonal,

$$A = \begin{bmatrix} a_0\ 0 & \dots & \\ a_n \dots a_0\ 0 & \dots & \\ & \ddots & \\ & \dots\ a_n \dots a_0 \end{bmatrix} \tag{10}$$

$$B = \begin{bmatrix} b_0\ 0 & \dots & \\ b_m \dots b_0\ 0 & \dots & \\ & \ddots & \\ & \dots\ b_m \dots b_0 \end{bmatrix} \tag{11}$$

and u and x are t-dimensional column vectors that represent the given input and desired output signals respectively. The desired output therefore minimizes $f(x) = \|Bx - Au\|^2$, and can be found by Least Squares as described above. In experiments, we use the standard noisy feed-forward technique to generate the initial iterate for the stochastic Least Squares solver.

Sorting.

To sort an array of numbers on current CPUs, one often employs recursive algorithms like QUICKSORT or MERGESORT. Sorting can be recast as an optimization over the set of permutations. Among all permutations of the entries of an array $u \in \mathbf{R}^n$, the one that sorts it in ascending order also maximizes the dot product between the permuted u and the array $v = [1 \dots n]^\top$ [9]. In matrix notation, for an $n \times n$ permutation matrix X, Xu is the sorted array u if X maximizes the linear cost $v^\top Xu$. Since permutation matrices are the extreme points of the set of doubly stochastic matrices, which is polyhedral, such an X can be found by solving the linear program

$$\max_{X \in \mathbf{R}^{n \times n}} v^\top Xu \quad \text{s.t. } X_{ij} \geq 0,\ \sum_i X_{ij} \leq 1,\ \sum_j X_{ij} \leq 1. \tag{12}$$

The corresponding unconstrained exact quadratic penalty function is

$$f(X) = -v^\top Xu + \lambda_1 \sum_{ij} [X_{ij}]_+^2 + \lambda_2 \sum_i \left[\sum_j X_{ij} - 1\right]_+^2 + \lambda_2 \sum_j \left[\sum_i X_{ij} - 1\right]_+^2 \tag{13}$$

where λ_1 and λ_2 are suitably large constants, and the ijth coordinate of the subgradient of f is

$$[\nabla f(X)]_{ij} = -u_i v_j + 2\lambda_1 [X_{ij}]_+ + 2\lambda_2 \left[\sum_j X_{ij} - 1\right]_+ + 2\lambda_2 \left[\sum_i X_{ij} - 1\right]_+ . \tag{14}$$

Note that sorting is traditionally not thought of as an application that is error tolerant. Our methodology produces a potentially error tolerant implementation of sorting.

Bipartite Graph Matching.

Given a bipartite graph $G = (U, V, E)$ with edges E connecting left-vertices U and right-vertices V, and weight function $w(e)$, $e \in E$, a classical problem is to find a subset $S \subseteq E$ of edges with maximum total weight $\sum_{e \in S} w(e)$ so that every $u \in U$ and every $v \in V$ is adjacent to at most one edge in S. This is the maximum weight bipartite graph matching problem and is typically solved using the Hungarian algorithm or by reducing to a MAXFLOW problem and applying the Push-Relabel algorithm [12]. Like other linear assignment problems, it can also be solved by linear programming: let W be the $|U| \times |V|$ matrix of edge weights and let X be a $|U| \times |V|$ indicator matrix over edges, with X_{ij} binary, and only one element in each row and each column of X set. The weight of a matching given by X is then $\sum_{ij} X_{ij} W_{ij}$, which is linear in X, so it suffices to search over doubly stochastic matrices, as in the previous example.

Typical implementations of Bipartite Graph Matching are again not considered error tolerant. Our methodology produces a potentially error tolerant implementation of Bipartite Graph Matching.

Other combinatorial problems.

A host of other combinatorial problems can be solved exactly on stochastic processors by reduction to linear programming. These include MAXFLOW, MINCUT, and SHORTESTPATH [22]. In addition, the best approximation algorithms for many NP-hard problems involve rounding the solution to linear programs [22].

Other numerical problems.

The Courant-Fisher Minmax Theorem [13][Theorem 8.1.2] expresses the kth largest eigenvalue and eigenvector of a matrix in variational form. Alternatively, one can find the top eigenvalue/eigenvector pair by maximizing a Rayleigh quotient, subtracting the resulting rank-1 matrix from the target matrix, and repeating k times. Many data fitting problems, like fitting Support Vector Machines, are defined as variational problems, and efficient stochastic gradient algorithms for them already exist [26].

To summarize, the above numerical optimization-based methodology can be used to make a large class of applications robust - the ones that require precisely correct outputs (fragile applications), e.g., *sorting*, etc., as well as the ones that that do not (intrinsically robust applications), e.g., *IIR filters*, etc. To the best of our knowledge, this is the first work on a generic methodology to transform application code for timing error tolerance that may work for both fragile and intrinsically robust applications.

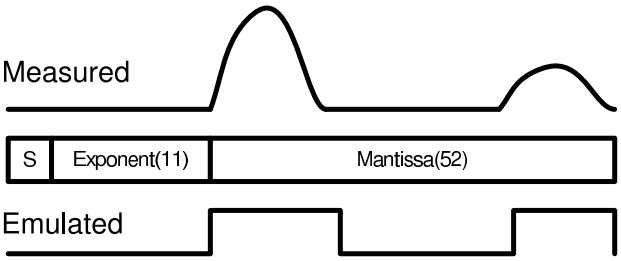

Figure 3: Measured distribution of error magnitudes for floating point data versus the distribution used for emulating the behavior.

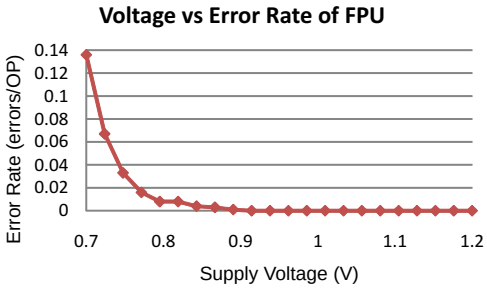

Figure 4: Error Rate of an FPU as the voltage is scaled.

5. EXPERIMENTS

5.1 Methodology

To evaluate the robust versions of the above algorithms, we built an FPGA-based framework with support for controlled fault injection. Our framework consists of an Altera Stratix II EP2S180 FPGA that hosts a Leon3 [11] soft core processor. The FPGA-based framework allows us to run the stochastic and baseline implementations of our applications on the Leon3 core.

The framework is designed to provide us fine-grained control over the stochasticity of the processor. To introduce stochasticity, we chose to inject errors in the floating point unit (FPU) of the Leon3 core. Error injection was done using a software-controlled fault injector module that we mapped onto the FPGA. At random times, the fault injector perturbs one randomly chosen bit in the output of the FPU before it is committed to a register. The distribution of bit faults was modeled from circuit level simulations of functional units [18], where many of the errors predominantly occur in the most significant bits. The rest of the faults primarily occur in the low order bits, resulting in low magnitude errors. Figure 3 illustrates the measured distribution of faults across floating point bits, and the distribution used to emulate this behavior.

The time between corruptions was drawn using a uniform distribution generated by a Linear Feedback Shift Register. While the fault model is simplistic, it is appropriate considering the goal of the paper. Also, the fault model is a surprisingly reasonable approximation of voltage overscaling-induced errors in the FPU.

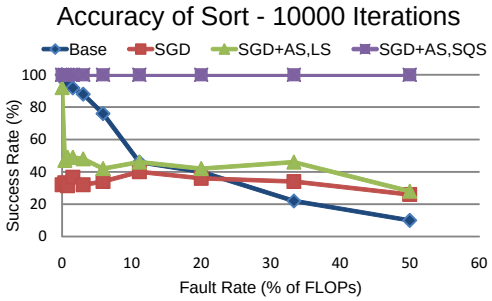

Figure 5: Success rate for different implementations of Sorting as a function of fault rate

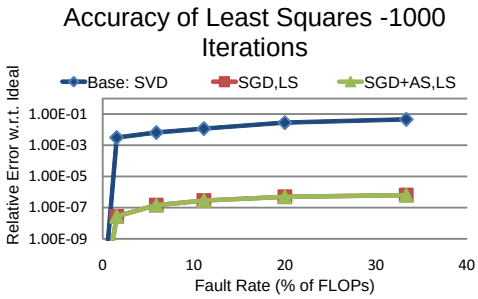

Figure 6: Relative error for different implementations of Least Squares as a function of fault rate (Lower is better). SQS results in errors larger than 1.0.

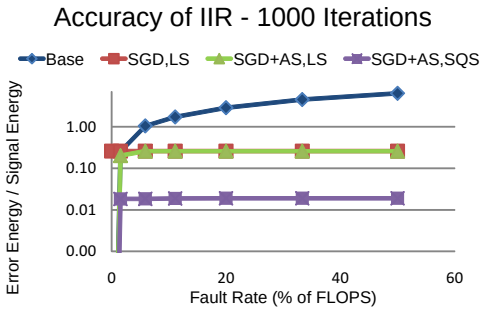

Figure 7: Error-to-Signal ratio for different implementations of IIR as a function of fault rate (Lower is better)

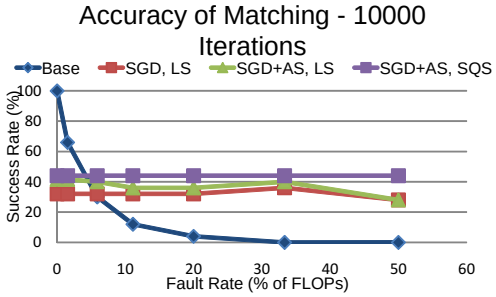

Figure 8: Success rate for different implementations of Bipartite Graph Matching as a function of fault rate

To calculate the energy benefits from application robustifcation, a model for voltage versus error rate of the FPU is needed. Figure 4 represents the relationship between voltage and error rate for the FPU that was used for our energy calculations. The results were generated using circuit-level simulations.

5.2 Gradient Descent

To explore the feasibility of the proposed approach to provide robustness and energy benefits, we evaluated stochastic gradient descent (SGD) on four problems, Least Squares, IIR filters, Bipartite Graph Matching, and Sorting across a wide range of fault rates. We evaluated both *linear scaling (LS)* of the step size, $\frac{1}{t}$, and *sqrt scaling (SQS)* of the step size, $\frac{1}{\sqrt{t}}$, where t is the number of iterations. We also examined *aggressive stepping (AS)* (see Section 3.2). In our graphs, SGD refers to a fixed number of iterations, while SGD+AS refers to the fixed number of iterations with a period of aggressive stepping at the end.

The metric used to describe the quality of output is different for each benchmark. For Sorting, the y-axis represents the percentage of outputs where the entire array is sorted correctly (any undetermined entries (NaNs), wrongly sorted number, etc., is considered a failure). For Bipartite Graph Matching, the y-axis represents the percentage of outputs where all the edges are accurately chosen. For Least Squares, the quality of output is measured as the relative difference between the ideal output and actual output. ($\|Ax - b\|^2$) For the IIR filter, the quality of output was measured using the mean square error (MSE) metric, and the ratio of the error energy and output signal energy. ($\|Y - Y_{actual}\|/\|Y\|$)

We chose small problem sizes for our evaluations due to low FPGA-based simulation speeds and the need to manually orchestrate each experiment (e.g., identify coefficients, parameters, etc.). For sorting, array size is 5 elements. For the LSQ problem, A is 100×10 and B is 100×1. Bipartite Graph Matching is performed for a graph with 11 nodes and 30 edges. IIR filter uses a 10-tap filter for 500 input samples. State of the art deterministic applications are used for each of the application baselines. Sorting was implemented using the C++ Standard Template Library (STL). Least Squares was implemented using SVD, QR, or Cholesky decompositions. IIR was implemented using a simple procedural routine (Section 4). Bipartite Graph Matching was implemented using the OpenCV library [2].

Our evaluations were performed for different fault rates. We define fault rate to be the inverse of the average number of floating point operations between two faults. Note that the baseline kernels will not see any errors at very low fault rates (≤ 0.1), due to the small problem sizes (i.e. not, enough floating point operations).

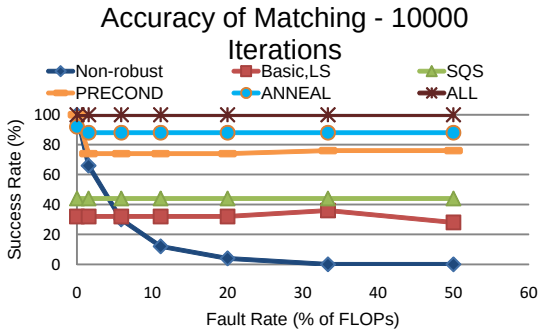

Figure 9: The effect of enhancements to gradient descent on the success rate of Bipartite Graph Matching

Examining the results, we see that we are able to achieve high quality results for both the fragile and the intrinsically robust applications. Sorting (Figure 5) performs poorly with linear step size scaling, but with sqrt step size scaling is able to achieve 100% accuracy even with large fault rates. Least Squares (Figure 6), on the other hand, performs better with linear step size scaling. It is also able to get highly accurate results, within 10^{-6}% of the exact value computed offline with an SVD-based baseline. Similarly, IIR (Figure 7) using SGD produces several orders of magnitude less error compared to the baseline procedural IIR implementation. IIR error reduces further with sqrt step scaling. The benefits of Aggressive Stepping for the applications are most pronounced for low fault rates ($< 1\%$).

Bipartite Graph Matching (Figure 8) using 10000 iterations of SGD showed little performance degradation with increasing fault rates. However, the maximum success rate obtained, even using aggressive stepping and step scaling, was limited to below 50%.

5.3 Gradient Descent Variants

Gradient descent fares well at low error rates, but the performance can fall off very rapidly for some applications and with certain inputs that result in poorly conditioned objective functions. Here, we examine several techniques which allow gradient descent to perform consistently better even at higher error rates. In these tests, 0-50% of floating point operations are erroneous. In order to reduce the number of variables, we examine only Bipartite Graph Matching. We also compare the results of gradient descent to that of the baseline Bipartite Graph Matching routine from the OpenCV library [2]

5.3.1 *Preconditioning*

The basic version of gradient descent involves minimizing the cost function $-c^T x + \lambda[Ax - b]_+$. Preconditioning allows us to rewrite the cost function so that gradient descent is solving an easier problem. We perform preconditioning by taking the QR decomposition to get an orthogonal matrix Q and a right triangular matrix R such that $A = QR$. The cost function can then be rewritten as $-c^T x + \lambda[QRx - b]_+$. Defining the new y as $y = Rx$, allows us to then rewrite our cost function as $-c^T x + \lambda[Qy - b]_+$. We also need to find a c_{new} such that $c_{new}^T y = c_{new}^T Rx = c^T x$. This gives us $c_{new}^T R = c^T$ which can be rewritten as $R^T c_{new} = c$. This allows us to solve for c_{new}. Gradient descent can then be used to minimize $-c_{new}^T y + \lambda[Qy - b]_+$. After finding the y that minimizes the cost function, solving $Rx = y$ for x, gives us the answer to the original problem.

Figure 9, shows that the basic gradient descent performs worse than the non-robust Bipartite Graph Matching algorithm at low error rates ($< 5\%$). Once preconditioning is performed, gradient descent is able to achieve an accuracy comparable to the non-robust version for up to a 2% fault rate. SGD, with preconditioning, substantially outperforms the non-robust Bipartite Graph Matching fault rates above 2%.

5.3.2 *Momentum*

We also examined the use of a momentum of 0.5, so that the search direction for iteration t, can be expressed as $d(t) = 0.5 * \delta f(t) + 0.5 * d(t-1)$. For the Sorting problem, utilizing momentum improved the success rate $20 - 40\%$ relative to the basic gradient descent. However, the addition of momentum provided only a marginal benefit ($< 5\%$), for Bipartite Graph Matching. For both applications, the success rate was still well below 100%.

5.3.3 *Alternate Step Size Scaling*

Baseline gradient descent scales the step size as $\frac{1}{t}$, where t is the number of iterations executed so far. In later iterations, this may cause the step size to be so small that insufficient progress is made per iteration. We thus examine scaling the step size more slowly, as $\frac{1}{\sqrt{t}}$. Again, utilizing step scaling we see some improvement in performance relative to the basic gradient descent. However, the solver success rate continues to be less than 40%.

5.3.4 *Annealing*

The contribution of the penalty function (corresponding to the constraints) to the gradient calculation can impede progress towards the solution, especially if these constraints are poorly scaled compared to the actual objective. This can be mitigated by annealing the penalty parameter (α). The parameter α is periodically increased as the solver moves closer towards the minimum. As we see in Figure 9, using annealing provides substantial benefits. It achieves a 88% success rate even with roughly half of the floating point operation containing noise.

5.3.5 *All Enhancements*

While incorporating annealing in the penalty function calculation provides the most benefit of any individual tech-

nique, gradient descent can perform even better if we utilize all of the above techniques together. In fact, utilizing all of these techniques, stochastic gradient descent is able to achieve a 100% success rate even when there are fault rate is scaled to 50%.

5.4 Conjugate Gradient

While stochastic gradient descent-based techniques provide high robustness, it often comes at the expense of significantly increased runtime due to the large number iterations required for convergence. The Conjugate Gradient method, on the other hand, allows efficient generation of conjugate directions by taking a linear combination of the negative residual (which is simply the steepest descent direction) and the previous direction. In general, the CG method can guarantee convergence in at most n iterations for a $Ax = B$ problem where n is the dimension of x. Figure 10 shows the accuracy of output for our CG-based implementation of the Least Squares problem, when using 10 iterations of CG. We consider three baseline implementations (SVD, QR, and Cholesky Decompositions). The SVD based solver allows for the highest accuracy, even with ill-conditioned problems. The Cholesky based solver is the fastest baseline implementation but can only be used for a subset of problems. The QR-based implementation is slower than Cholesky-based implementations, but is also more accurate.

Experimentally, the CG implementation was on average 30% faster than the QR/SVD baselines. And 10 iterations of the CG were comparable to the execution time of the Cholesky baseline.

The relatively small time of convergence allows CG-based implementations of the LSQ problem to have lower energy than the baseline implementations for the entire range of accuracy targets when voltage overscaling is used (accuracy targets lower than 1.00E-07 can't be met using CG). This is because it becomes possible to scale down the voltage and the number of iterations concurrently. Figure 11 shows the normalized energy results for the FPU for the Least Squares problem assuming the voltage / error rate curve from Figure 4. The results show that there is considerable potential for using the proposed numerical optimization-based methodology to reducing the energy of software execution by voltage overscaling a processor and then letting the applications tolerate the errors.

6. LIMITATIONS AND FUTURE WORK

There are several simplifying assumptions that the above methodology makes. For example, certain control phases of execution are assumed to be error-free. While the assumption may be reasonable for a large class of data-intensive applications (such as the ones presented in the paper) where the control phases of the stochastic implementation can be identified and protected using software and hardware techniques (e.g., increasing the voltage during control phases), it may be difficult to distinguish between data and control phases for more complex applications. Our future work will explore the effectiveness of the proposed methodology for a larger class of applications.

Second, it may not be uncommon for an iterative methodology such as ours to have higher overall energy consumption than the baseline implementation for certain applications because of the larger number of operations required for convergence. In fact, we observed that the number of floating point operations required by our applications could be up to 10-1000X higher than that for the baseline implementations. Note, however, that it is not an indictment of the proposed approach as the energy benefits depend greatly on optimization engine chosen for solving the stochastic optimization problems. Our future work will attempt to identify the most appropriate optimization engine for the stochastic implementation of each problem. Finding ways to decrease the number of iterations required for convergence will also be key in making this methodology more useful.

Additionally, future work will involve investigating the robustness of the proposed methodology for different fault models. Note that the ultimate feasibility of the proposed approach will be determined also by issues related to scheduling, runtime management, programmer annotations to identify critical variables, automation of the program transformation flow, competitiveness against guardbanding, etc. These issues are the subject of future work.

7. CONCLUSION

Environmental and manufacturing variations coupled with reduced guardbands can cause timing errors in processors. Rather than utilizing hardware approaches to detect and mask these errors, in this paper, we propose to allow these timing errors to occur and to cope with them in software. We proposed a formal methodology to make applications robust against the noise of such processors. The methodology consists of recasting the application as an optimization problem and applying off-the-shelf stochastic optimization procedures to find the solution. To the best of our knowledge, this is the first work on a generic methodology to transform application code for timing error tolerance geared at both fragile and intrinsically robust applications. Experiments on an FPGA show that the proposed methodology indeed has potential to tolerate noise in a processor's numerical units under certain simplifying assumptions. Results show that the proposed methodology may be capable of producing high quality results for both intrinsically robust algorithms such as IIR filter and Least Squares, and for fragile applications such as Sorting and Bipartite Graph Matching. Moreover, we showed that energy benefits may also be possible for certain applications/inputs (e.g. when using a CG-based solver for the Least Squares problem).

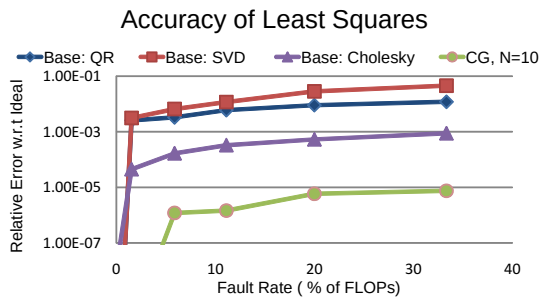

Figure 10: Accuracy for a CG-Based implementation of Least Squares (10 iterations)

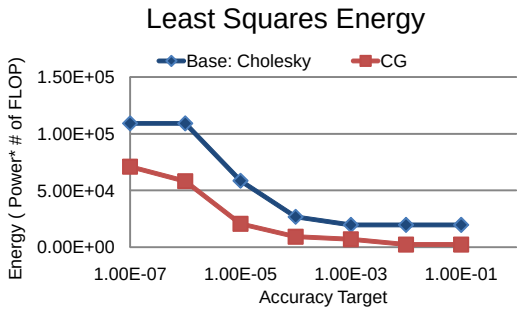

Figure 11: Energy for a CG-Based implementation of Least Squares

Future work will focus on evaluating and mitigating costs of the proposed approach.

8. ACKNOWLEDGMENTS

This work was supported in part by Intel, NSF, GSRC, and an Arnold O. Beckman Research Award. Feedback from Naresh Shanbhag, Doug Jones, and anonymous reviewers helped improve this paper.

9. REFERENCES

[1] International Technology Roadmap for Semiconductors 2008, http://public.itrs.net.
[2] The OpenCV Library, http://http://opencv.willowgarage.com/wiki/.
[3] A. Kahng, S. Kang, R. Kumar and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *IEEE International Symposium on High-Performance Computer Architecture(HPCA)*, January 2010.
[4] J. Sartori, R. Kumar, S. Kang and A. Kahng. Recovery-driven design: A methodology for power minimization for error tolerant processor modules. In *the 47th Design Automation Conference (DAC)*, June 2010.
[5] J. Bau, R. Hankins, Q. Jacobson, S. Mitra, B. Saha, and Adl A. Tabatabai. Error Resilient System Architecture (ERSA) for Probabilistic Applications. In The 3rd Workshop on System Effects of Logic Soft Errors (SELSE), April 2007.
[6] D. P. Bertsekas, A. Nedic, and A. E. Ozdaglar. *Convex Analysis and Optimization*. Athena Scientific, 2001.
[7] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *DAC*, pages 338–342, 2003.
[8] L. Bottou. Online algorithms and stochastic approximations. In D. Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, 1998.
[9] R. W. Brockett. Dynamical systems that sort lists, diagonalize matrices and solve linear programming problems. In *Linear Algebra and Its Applications*, volume 146, pages 79–91, 1991.
[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 7, Washington, DC, USA, 2003. IEEE Computer Society.
[11] Aeroflex Gaisler. Leon3 processor, 2008.
[12] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *ACM symposium on Theory of Computing (STOC)*, pages 136–146, 1986.
[13] G. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.
[14] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance, 1999.
[15] N. J. Higham. *Accuracy and Stability of Numerical Algorithms, Chapter 12*. SIAM, second edition, 2002.
[16] K. Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518–528, June 1984.
[17] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *in Proc. Intl. Symp. Microarch. (MICRO)*, pages 347–358, 2006.
[18] C. Kong. Study of voltage and process variation's impact on the path delays of arithmetic units. In *UIUC Master's Thesis*, 2008.
[19] R. Kumar. Stochastic processors. In *NSF Workshop on Science of Power Management*, March 2009.
[20] N. Shanbhag, R. Abdallah, R. Kumar and D. Jones. Stochastic computation. In *the 47th Design Automation Conference(DAC)*, June 2010.
[21] A Nemirovski, A Juditsky, G Lan, and A Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4):1574–1609, 2009.
[22] C. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover, 1998.
[23] G. Di Pillo. Exact penalty methods. In I. Ciocco, editor, *Algorithms for Continuous Optimization*, 1994.
[24] J. Sartori and R. Kumar. Three scalable approaches to improving many-core throughput for a given peak power budget. In *International Conference on High Performance Computing*, 2009.
[25] J. Sartori and R. Kumar. Overscaling-friendly timing speculation architectures. In *In the 20th ACM/IEEE Great Lakes Symposium on VLSI, GLSVLSI*, May 2010.
[26] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM. In *International Conference on Machine Learning (ICML)*, 2007.
[27] M. L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
[28] V. Simoncini and D. B. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing*, 25:454–477, 2003.
[29] D. Blaauw T. Austin, V. Bertacco and T. Mudge. Oppotunities and challenges for better than worst-case design. In *Proc. Asia South Pacific Design Automation*, pages 2–7, 2005.
[30] S.J. Wright. On the convergence of the newton/log-barrier method. Technical report, ANL/MCSP681 -0897, Mathematics and Computer Science Division, Argonne National Laboratory, 2001.
[31] D.B. Yudin and A. Nemirovskii. *Problem Complexity and Method Efficiency in Optimization*. John Wiley and Sons, 1983.