On Software Design for Stochastic Processors

Joseph Sloan, John Sartori, Rakesh Kumar
University of Illinois,
Urbana-Champaign
jsloan,sartori2,rakeshk@illinois.edu

ABSTRACT

Much recent research [8, 6, 7] suggests significant power and energy benefits of relaxing correctness constraints in future processors. Such processors with relaxed constraints have often been referred to as stochastic processors [10, 15, 11]. In this paper we present three approaches for building applications for such processors. The first approach relies on relaxing the correctness of the application based upon an analysis of application characteristics. The second approach relies upon detecting and then correcting faults within the application as they arise. The third approach transforms applications into more error tolerant forms. In this paper, we show how these techniques that enhance or exploit the error tolerance of applications can yield significant power and energy benefits when computed on stochastic processors.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies, Representation*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Algorithms, Design, Reliability

Keywords

Application Error Tolerance, Stochastic Processors, ABFT

1. INTRODUCTION

With growing challenges to sustaining Moore's law with process scaling alone, stochastic computing [4, 13, 14] is being investigated aggressively as a possible path for realizing future high performance and low power technologies. The key to realizing the stochastic computing vision will rest with the ability of researchers and developers to design efficient techniques which both enhance and exploit the natural error tolerance of applications. Due to growing fault rates and diverse failure behaviors, software techniques will be especially important for providing low power and low energy stochastic computation. In this paper, we discus three approaches that can be used to design robust software on stochastic processors.

The first approach involves relaxing application correctness constraints. Many applications have inherent algorithmic and cognitive error tolerance. For such applications, significant performance and energy benefits can be obtained by selectively allowing errors. As an example, we consider GPU applications where control divergence [3] incurs large overheads. By selectively eliminating control divergence through a technique called *branch herding* (i.e., forcing all threads to take the same control path for certain branches), we show that performance can be improved significantly while maintaining acceptable output quality acceptable for many applications.

For the second approach, faults are handled in a more traditional manner by detecting and then correcting or recovering from the faults. Unfortunately, the overheads of many typical fault detectors (e.g. dual modular redundancy (DMR)) are simply too large for these techniques to be utilized as a viable solution for stochastic computing. For this reason, techniques for low-overhead algorithmic fault detection and correction are extremely important. We present one example of low-overhead Algorithmic Based Fault Tolerance (ABFT) [5] for an important class of algorithms (sparse linear algebra) that will be used in many future applications. These techniques exploit both inherent properties of the data, as well as inherent fault tolerance characteristics of common iterative linear solvers. As error rates increase, however, the overheads incurred from frequent recovery events can make a detection-only-based fault tolerance approach (even an algorithmic one) infeasible. Frequently, applications may not be concerned with correcting all errors exactly, but instead with simply reducing the amount of noise below a certain threshold. Below that threshold, applications can still make efficient forward progress by naturally tolerating noise. In this paper, we present an approach for algorithmic fault correction by approximately correcting errors that occur in Matrix-Vector operations of an iterative linear solver.

The third approach aims to take any arbitrary application and transform it into a more robust form capable of efficiently running on stochastic processors. We describe one such approach for application transformation for robustness that utilizes a numerical optimization framework to naturally tolerate errors. By converting applications to numerical optimization problems with minima corresponding exactly to the original programs' output, we can use robust solvers to efficiently compute the original programs' deterministic output.

Section 2 describes the approach of application relaxation by using branch herding. Sections 3 and 4 describe the approaches for algorithmic detection and algorithmic correction, respectively, with examples involving linear algebra-based applications. Finally, Section 5 introduces a general approach for transforming applications for increased robustness, which we call *Application Robustification*. Section 6 concludes.

2. RELAXING CORRECTNESS

One approach to building software for stochastic processors involves relaxing the correctness constraints of the applications to improve performance or power characteristics [1]. We focus on GPU applications with frequent control divergence [3]. GPUs utilize SIMD-based architectures where multiple execution pipelines are designed to run in lockstep. Applications incur large performance overheads for synchronization when threads running on different execution pipelines encounter control divergence. However, many GPU applications also exhibit typical fault tolerance characteristics commonly seen in data-parallel applications with

DAC 2012, June 3-7,2012, San Francisco, California, USA.

```
while (--i && (xx + yy < T(4.0))) {
    y = x * y * T(2.0) + yC;
    x = xx - yy + xC;
    yy = y * y;
    xx = x * x;
} return i;
```

Figure 1: The main computation loop for Mandelbrot. The loop is unrolled 20 times in the actual application kernel.

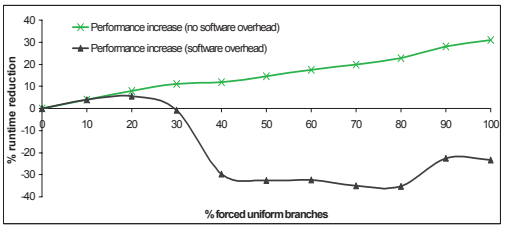

Figure 2: The performance of Mandelbrot can be increased by herding more branches. However, if software overhead is added to ensure branch uniformity, increasing the number of affected branches increases overhead and can even result in degraded performance.

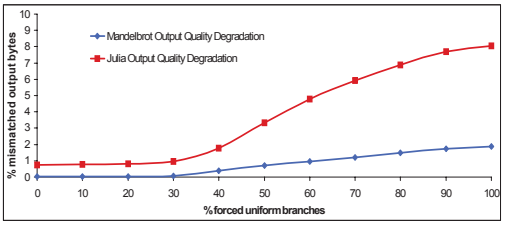

Figure 3: While eliminating control divergence can increase performance, blindly herding can result in degraded output quality.

inherent redundancy across time and space. So by carefully eliminating control divergence, we may achieve significant performance benefits.

Figure 1 shows an example of a kernel that exhibits control divergence, called Mandelbrot [18]. Control divergence arises in Mandelbrot because the number of iterations required to determine whether a particular is in the Mandelbrot (or Julia) set varies based on the point's location, especially in image regions near the set boundary, where some threads execute many iterations while others finish quickly.

The effect of control divergence on performance can be significant. Figure 2 shows the potential performance increase (runtime reduction) if control divergence is eliminated for a fraction of the static branches in Mandelbrot (from 0% to 100% of branches). The branches are chosen uniformly randomly when the fraction is less than 100%. Control divergence is eliminated by changing the source code to vote within a warp on the condition of a branch and forcing all threads in the warp to take the same (majority) direction at the branch.We call the technique *branch herding*. Branch herding can be implemented relatively efficiently in software, using the CUDA intrinsics ballot (_ballot) and population count (_popc). The ballot intrinsic is a warp vote function that combines predicates computed by each thread in a warp and sets the N_{th} bit in a 32-bit integer if the predicate evaluates to non-zero for the N_{th} thread in the warp. The ballot result is broadcasted to a destination register for each thread in the warp. We use the population count intrinsic to count the number of set bits in the ballot result. Branch herding can also be implemented easily in hardware by adding a 32-bit majority logic block.

While only 10% of dynamic instructions in Mandelbrot are branches, and less than 1% of branches diverge, performance can potentially be increased by 31% by eliminating control divergence. As the *no software overhead* performance series in Figure 2 demonstrates, performance increases for Mandelbrot as control divergence is eliminated for more branches. Figure 3 shows that the quality of the Mandelbrot output set degrades by less than 2%, even when divergence has been eliminated for all static branches. This shows

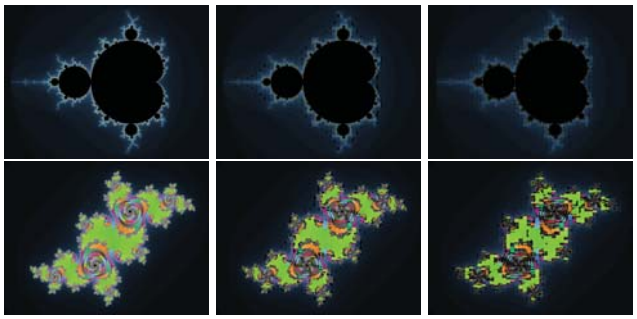

Figure 4: Progression of Mandelbrot (top) and Julia (bottom) images from 20% to 100% forced branch uniformity in 40% intervals.

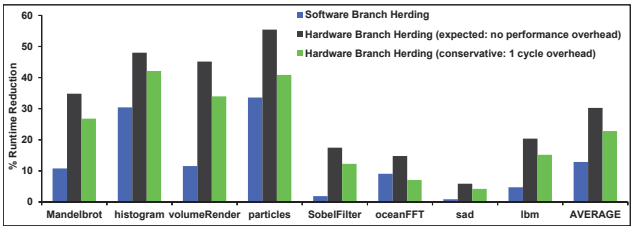

Figure 5: Performance for software and hardware branch herding.

that for certain error-tolerant applications, it may be possible to get significant performance benefits by relaxing correctness constraints related to control divergence for minimal output quality degradation.

Figure 4 shows the quality of the Mandelbrot and Julia output sets as the percentage of herded branches increases from 20% to 100% in increments of 40%.

A quick look at the last Julia output set, however, also suggests that an indiscriminate selection of branches for herding may result in significant output quality degradation for several applications. Therefore, any implementation of branch herding needs to carefully select the branches to target.

One policy for determining which branches are profitable for herding is to use profiling information in a feedback loop. After instrumenting a branch (or some fraction of the candidate branches) for herding (which can be done automatically by the compiler or manually by the programmer), the code is re-compiled and profiled to measure performance and output quality. If performance increases and output quality degradation remains below the acceptable threshold specified by the programmer, the branches are accepted for herding, else they are reverted to their original state. In terms of output quality, we find that in several cases, outputs may be considered acceptable even when herding is used for all the branches in a kernel function. In this case, herding can simply be switched with a compiler flag.

Experimental performance results for branch herding are shown in Figure 5. Software branch herding performance and output quality are measured directly at runtime (i.e., native execution on NVIDIA GeForce GTX 480.). The hardware branch herding technique in Figure 5 assumes some simple logic that eliminates the software overhead of herding. Hardware branch herding increases performance by 30% on average and up to 55% for individual applications. The software branch herding implementation achieves 13% performance benefits, on average.

Since branch herding exploits error tolerance to eliminate divergence, it may result in output quality degradation. Table 1 compares output quality degradation for the benchmarks with and without branch herding. Overall, branch herding does not result in much additional output quality degradation, while providing fairly substantial performance benefits.

3. ALGORITHMIC FAULT DETECTION

The second approach relies on detecting and then recovering from faults. In context of stochastic processors, we focus on techniques for algorithmic fault detection which exploit application er-

Table 1: Output Quality Degradation (%) for Branch Herding compared to Original

% Mismatch	Mandelbrot	histogram	volumeRender	particles	SobelFilter	oceanFFT	sad	lbm
Original	0.03	0.00	6.72	18.24	0.00	0.03	0.00	6.7E-7
Branch Herding	1.87	5.82	7.61	18.24	6.00	0.03	0.42	5.6E-5

ror tolerance as only those errors that adversely affect application output are considered . Below we discuss one example application class (sparse linear algebra) which exhibits inherent fault tolerance that algorithmic techniques can exploit.

Sparse linear algebra problems frequently have well defined structures. Common examples of structure are diagonal, banded diagonal, and block diagonal matrices. For example, *qpband* (Figure 6), which represents a canonical indefinite optimization problem, illustrates a typical banded diagonal structure (the nonzero pattern is on the left). Similarly, the matrix *msc00726* (Figure 7), representing a structural engineering problems from the Boeing test matrix group [2], also contain banded diagonal type structures.

Such structures in sparse problems commonly translate into uniform distributions of the column sums, which are directly used in algorithmic checks [17, 5] ($c^T(Ax) = (c^T A)x$ where $c = \bar{1}$). These matrices with well-defined distributions of column sums present an opportunity to sample only a fraction of the columns, which gives up a small degree of coverage (some errors may be missed) for a significant reduction in overhead. We call the technique using a random sampling and a sampling based upon clustering, Approximate Random and Approximate Clustering respectively. These checks are especially valuable since they exploit the fact that some errors can be tolerated by the application itself (e.g. iterative methods that converge to more accurate solutions).

Another opportunity for reducing algorithmic detection overhead for sparse linear algebra applications is that many such applications typically use the same matrix as part of many individual operations. For example, iterative solvers for linear systems ($Ax = b$) use MVM multiple times over thousands of iterations. This property of frequent data reuse makes it possible to analyze the structure of a given matrix or precondition the matrix to have a more amenable form for low overhead algorithmic fault detection, thus amortizing the setup cost by using lower overhead checks for subsequent MVM operations.

Identity conditioning (IC) transforms the high variance column sum distribution of the original matrix (A) into a more uniform set of values by using a check vector tailored to the given problem, instead of the traditional checksum: $c = \bar{1}$. IC finds such a tailored check vector by solving the system:

$$c^T A = \bar{1}^T \quad \text{(identity equation)}$$

The effect of A and the variance of the column sums can be minimized by then using:

$$c^T y = (c^T A)x = \bar{1}^T x = \sum x \quad \text{(IC)}$$

This makes the problem directly amenable to low-cost sampling as the variance in A now has a smaller effect on the product $c^T A$, making the sampling in AR and AC more representative than when sampling the check vector $c = \bar{1}$.

While Identity Conditioning eliminates the influence of A on the check, additional conditioning can also eliminate the influence of x. The *Null Conditioning* (NC) algorithm finds a check vector in the null space of the matrix A, solving the equation

$$c^T y = (c^T A)x = 0 \quad \text{(NC)}$$

This significantly reduces the runtime overhead of the check, since the right side of the check requires no additional computation (e.g. the sum equals zero) and the memory locality is improved since the input is no longer read in the check.

Finding a vector in (or near) the null space of A is done by computing its smallest singular value using singular value decomposition (SVD). The accuracy of fault detection for NC depends on the size of the problem's smallest singular value.

To evaluate the effectiveness of these detection techniques, we compare each technique when applied to a single MVM operation over a set 100 problems from the University of Florida Sparse Matrix Collection [2]. The analysis includes the overhead incurred during the execution of the MVM operation and excludes the setup cost, such as clustering and conditioning. The utility of our fault detection algorithms depends on both their detection accuracy and performance overhead.

Figure 8 presents the results. The three columns on the left-hand side correspond to the three full algorithms we're evaluating: the traditional dense check, the Oracle algorithm, and a Decision Tree based-algorithm (picks best technique and parameters based on learned parameters from matrix characteristics). The eight columns on the right-hand side correspond to each base technique, highlighting their individual capabilities. The four techniques on the far right are combinations of the others (e.g. ICAR is IC + AR, while NCAR is NC + AR). For a given technique and input problem, we choose the configuration parameters (detection threshold, sampling rate, conditioning quality) that minimize its overhead while meeting the F-Score bound. The bars (left vertical axis) show the fraction of problems on which each detection technique achieved the target F-Score. The empty circles show each technique's overhead on each of the problems meeting the F-Score target, and the red filled circle within each column is the detector's average overhead across all of these problems. Finally, the lines within each column indicate the range of overheads within $\pm$ one standard deviation of the average as well as the minimum and maximum overheads.

In general, the results show that the traditional dense check has an average overhead of 32%, ranging from 5% for denser problems to 80% for larger sparse problems with poor locality. While in contrast, the overhead of AR was 16% on average, over the same set of sparse problems (i.e. 16% lower than the traditional dense check).

Figure 8 illustrates a scenario where the dense check is significantly brittle, meeting the F-Score target with only 10% of the problems.In contrast, the Oracle can combine checks to cover 94% of the problems and the Decision Tree succeeds with 77%. This is because faults directly in the check are more likely to occur with the traditional dense check, which performs more operations than the proposed techniques.

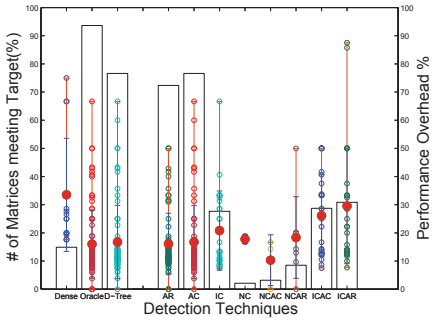

Figure 8: Right axis: Runtime overhead of each technique. Left axis: Number of problems meeting F-Score target. F-Score target=0.9, Fault Rate=1e-6, FaultModel=1

We also evaluate the fault detection techniques in the context of CG and IR sparse linear solvers. Errors affect linear solvers in two ways. First, since iterative algorithms converge from a poor solution to an accurate one, undetected errors are likely to slow down the algorithm's convergence or even cause it to diverge. Further, detected errors are managed using the classic rollback-restart technique, where the application is rolled back to some prior point in its execution and its execution is resumed. Our experiments use the simplest variant of this technique where the solver rolls back to the start of the current iteration every detected fault. In our experiments, each solver is executed until it reaches an error residual of 1e-6, meaning that if errors are detected, they may restart

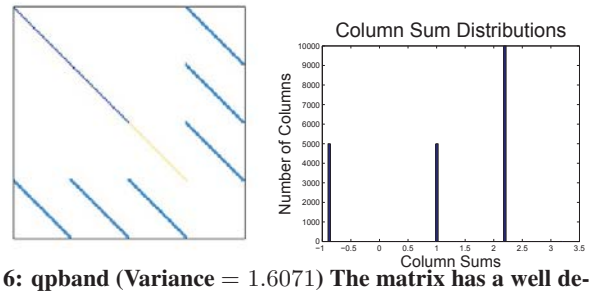

Figure 6: qpband (Variance $= 1.6071$) The matrix has a well defined and low variance ($< 1e3$) column sum distribution and is a good candidate for both Approximate Random and Approximate Clustering.

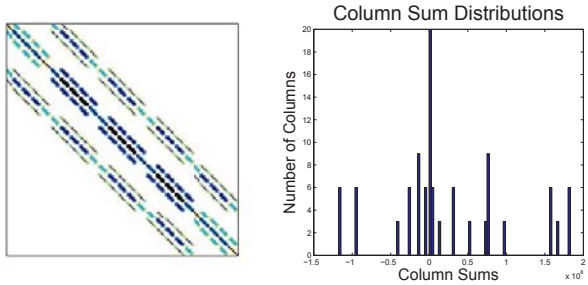

Figure 7: msc00726 (Variance $= 9.4724e14$) The matrix has high variance ($> 1e3$) column sums. This matrix is a good candidate for clustering given the finite sets of unique values shown above.

many iterations multiple times before they reach this goal. Figures 9 make this comparison between the execution time of the solver implementations employing a sparse check (on the x-axis) and the corresponding implementation employing the traditional dense check, via a sequence of graphs. The difference is measured as

$$overhead = \frac{Time_{sparse\_check} - Time_{dense\_check}}{Time_{dense\_check}}$$

which means that a difference of -50% corresponds to the linear solver executing twice as fast with the sparse detector than with the traditional dense detector. The overall difference in execution times of the linear solvers is consdiered in Figure 9.

Each detector and linear solver combination is evaluated on 5 different linear problems (separate sets for basic and preconditioned solvers). The average overhead over these matrices, for each detector, is shown as a red filled circle. Red lines correspond to the standard deviation and the max/min are shown using blue lines. The set of problems, for use with the basic solvers, were chosen randomly from those used in the MVM experiments.

The results also show that in the context of linear solvers the dense checks can have fairly large performance overheads (30-50%). For CG, the sparse check based implementation spent 17% less time in MVM operations on average than the traditional dense check-based implementations. This corresponds to a total execution time that is 9% lower on average. For IR, the sparse check based implementations spent 10% less time in MVM operations than the dense check-based implementations on average. This corresponds to 5% lower total execution time on average.

The results show that the impact of larger setup overheads for some of the techniques (e.g. clustering and preconditioning), in the context of both the IR and CG, is fairly negligible ($< 0.01\%$), since the amount of reuse is high. We observed that the absolute amount of reuse in the context of CG is dependent on the conditioning of the problem which impacts the number of iterations required to reach the desired solution. The error rate can also have an impact on the number of iterations and hence the amount of reuse within the algorithm

Upon analyzing the performance of the techniques in the different scenarios shown in Figure 9, we observed that the overall overhead can be reduced by 5%-20% by configuring the techniques to minimize the overhead from missed faults and false positives.

CG, and IR are two real application contexts that demonstrate that the sparse techniques are frequently able to exploit structure and reuse in sparse problems to reduce the overall overhead of algorithmic fault tolerance compared to the traditional dense checks. More details of our work on algorithmic fault detection for sparse linear algebra applications can be found in [17].

4. ALGORITHMIC FAULT CORRECTION

Previous algorithmic techniques for correcting errors [5] are primarily limited to scenarios involving rare error events, because of high overheads and the inability to make forward progress. Many applications contain inherent fault tolerance however, and are not always concerned with correcting all errors exactly. Therefore we can frequently use techniques that only approximately correct errors, ensuring that the aggregate effect of errors on the application's correctness and performance is bounded.

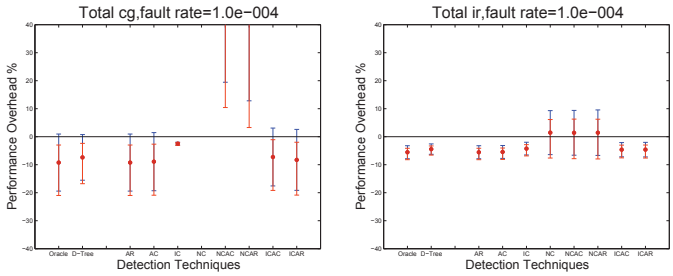

Figure 9: Percent difference between the execution time of the sparse techniques vs. dense check applied to CG & IR. Each column shows the total execution time overhead.

The general problem formulation of Algorithmic Fault Correction is, therefore: Given an application with an unknown correct output y, ensure that the application, even in the presence of faults produces an output y^* within a certain threshold of y.

As an example, in the context of Linear Algebra, we consider an MVM operation ($v = Au$) with k faulty entries in the output vector(v'). The traditional approach would explicitly detect and correct each of the k faults. In reality, the application may only care about approximately correcting the error ($e = v' - v$), and improving the accuracy (i.e. RMS $\|v' - v\|^2$). Therefore an algorithmic correction technique for the MV product could involve subtracting the projection of the error onto a code space. The partially corrected MV product(v'') in then found by:

$$v'' = v' - \frac{(c^T e)c}{\|c\|^2} \quad (1)$$

One of the primary advantages of this particular approach, is that this type of approximate correction is guaranteed to always improve accuracy:

$$\|v'' - v\|^2 = \|v' - v\|^2 - \frac{(c^T e)^2}{\|c\|^2}$$
$$\|v'' - v\|^2 \leq \|v' - v\|^2 \quad (2)$$

The above algorithmic correction can be easily adapted to account for the most important faults in terms of performance and accuracy. The developer has significant flexibility in the amount and types of codes chosen for the correction, depending on the accuracy targets which are desired.

5. APPLICATION TRANSFORMATIONS FOR ROBUSTNESS

The approaches described in prior sections are application-specific. Having the ability to transform arbitrary programs into more error tolerant forms is also an important consideration for stochastic processors. We call this *Application Robustification* [16]. In this section, we describe one approach for taking an arbitrary application and converting it into a more error tolerant form by reformulating it as a numeric optimization problem. We express the applications as constrained optimization problems, mechanically convert these to an unconstrained exact penalty form, and then

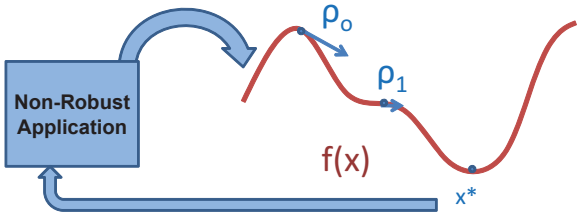

Figure 10: Application Robustification involves converting an application to an unconstrained optimization problem, where the minimum corresponds to the output of the original non-robust application.

solve them using gradient descent and conjugate gradient algorithms. This approach is quite generic, since linear programming, which is P-complete, can be implemented this way.

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 10:

- 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?

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

$$\underset{x\in\mathbf{R}^d}{\text{minimize}}\, f(x) \text{s.t. } g(x) \leq 0, h(x) = 0 \tag{3}$$

for some functions f, g, and h. Commonly, the transformation of a given problem into its general variational form (3) is often immediate from the definition of the problem. We provide several illustrative examples below.

Least Squares Given a matrix A and a column vector b, a fundamental problem in many problems is to find a vector x that minimizes $\|Ax - b\|^2$ This problem is typically implemented on current CPUs via the SVD or the QR decomposition of A. The robust formulation of this problem is constructed by minimizing the quadratic function: $f(x) = \|Ax - b\|^2 = x^\top A^\top Ax - 2b^\top x + b^\top b$. is The Least Squares problem is commonly thought of as a more intrinsically robust application, due to the continuous nature of much of it computation. We'll now consider the formulation of a problem not typically seen as fault tolerant, sorting.

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 $v = [1 \dots n]^\top$ 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

$$\max_{X\in\mathbf{R}^{n\times n}} v^\top Xu \quad \text{s.t. } X_{ij} \geq 0, \sum_i X_{ij} \leq 1, \sum_j X_{ij} \leq 1. \tag{4}$$

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 The maximum weight bipartite graph matching problem can also be solved with a linear program, similar to sorting but with a more generalized objective function. Typical implementations are again not considered error tolerant (e.g. Hungarian algorithm). Our methodology however produces a potentially error tolerant implementation of Bipartite Graph Matching.

Once we have converted the programs (both those that require precisely correct outputs –fragile applications, as well as those that do not–intrinsically robust applications) into optimization formulations,the best solver to compute the programs output can now be determined.

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 $x^1 \dots x^i \in \mathbf{R}^d$ via the iteration

$$x^i \leftarrow x^{i-1} + \lambda^i \nabla f(x^{i-1}), \tag{5}$$

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. 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 [12]. As long as the ∇f is unbiased, gradient descent can eventually extract a solution with arbitrarily high accuracy. [16]

We rely on an exact penalty method to convert constrained problems, such as (3) into unconstrained problems that can be solved by gradient descent:

$$f(x) + \mu \sum_i |h_i(x)| + \mu \sum_j [g_j(x)]_+ . \tag{6}$$

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.

For example, the linear program for Sorting (4) can be converted into a corresponding unconstrained problem by using an exact quadratic penalty function:

$$f(X) = -v^\top Xu + \lambda_1 \sum_{ij} [X_{ij}]_+^2 + \lambda_2 \sum_i \left[\sum_j X_{ij} - 1\right]_+^2 + \lambda_2 \sum_j \left[\sum_i X_{ij} - 1\right]_+^2 \tag{7}$$

where λ_1 and λ_2 are suitably large constants.

While we use gradient descent as a search strategy for most of our kernels, some kernels may warrant the use of other search strategies. For example,the conjugate gradient (CG) method can be used for well conditioned (quadratic objective functions) to obtain very fast convergence with large problems.

To evaluate the robust versions of the above algorithms, we built an FPGA-based framework with support for controlled fault injection [16]. To calculate the energy benets from application robustifcation, we also used circuit-level simulations to calculate the relationship between voltage and error rate for the FPU.

To explore the feasibility of the proposed approach to provide robustness and energy benefits, we evaluated stochastic gradient descent (SGD) on the problems for Bipartite Graph Matching and Sorting across a wide range of fault rates.

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.

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. Bipartite Graph Matching is performed for a graph with 11 nodes and 30 edges. State of the art deterministic applications are used for each of the application baselines (i.e. the C++ Standard Template Library (STL) and Hungarian Algorithm)

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 11) performs poorly with linear step size scaling, but with sqrt step size scaling is able to achieve 100% accuracy even with large fault rates.

Bipartite Graph Matching (Figure 12) using 10000 iterations of SGD showed little performance degradation with increasing fault rates. However, using a combination of preconditioning, step sizing, and annealing techniques with gradient descent showed that 100% accuracies were also obtainable across even the largest fault rates.

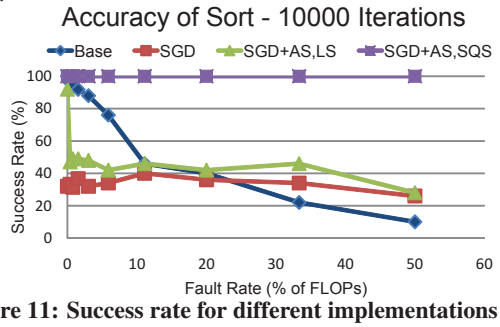

Figure 11: Success rate for different implementations of Sorting as a function of fault rate

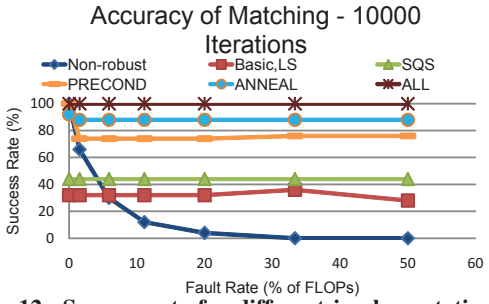

Figure 12: Success rate for different implementations of Bipartite Graph Matching as a function of fault rate

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. For some applications where the transformed implementation has complexity per iteration less than the original applications complexity, it may be possible to show energy benefits by voltage overscaling a processor and letting the processor have errors. For other applications where the complexity per iteration of the optimization form is equal or greater than the original implementation complexity, it may be more difficult to show energy benefits.

Figure 13 shows an example of a problem where the complexity of the transformed implementation is lower. In this Figure, the y-axis shows the normalized energy results for the FPU for a Least Squares problem ($Ax = B$) assuming a voltage overscaled processor (non-zero error rates). The quadratic nature of the problem allows for CG to converge in fewer iterations (compared to gradient descent), while also naturally tolerating certain types of errors. Additionally, by using a specialized accelerator for linear operations [9], more applications such as Graph Matching can achieve energy benefits by using stochastic processors. For some problems, such as Sorting, it will be remain difficult however, even with a accelerators, to achieve energy benefits with voltage/scaling type models.

6. CONCLUSIONS

Harnessing the power of stochastic computing systems depends heavily on the ability of researchers and developers to design efficient algorithms and techniques which both enhance and exploit the error tolerance of applications. This paper presented three approaches for building applications for stochastic processors. This included: relaxing the correctness of applications, algorithmic detection/correction as faults arise, and application transformation

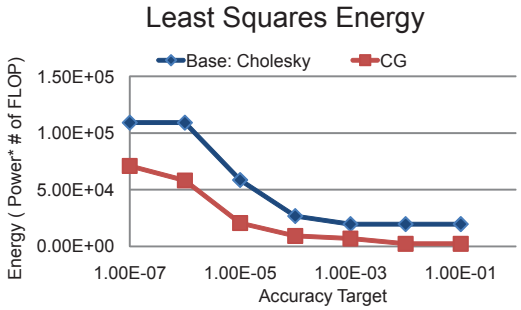

Figure 13: Energy for a CG-Based implementation of Least Squares

for robustness. In this paper, we show how these techniques that enhance or exploit the error tolerance of applications can yield significant power and energy benefits when computed on stochastic processors.

7. REFERENCES

[1] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, editors. *the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI), Beijing*. ACM, 2012.

[2] Timothy A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[3] W. Fung, I. Sham, G. Yuan, and T. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO*, pages 407–420, 2007.

[4] R. Hegde and N.R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, pages 30 – 35, 1999.

[5] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518 –528, 1984.

[6] 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)*, 2010.

[7] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Recovery-driven design: A methodology for power minimization for error tolerant processor modules. In *the 47th Design Automation Conference (DAC)*, June 2010.

[8] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *Asia and South Pacific Design and Automation Conference (ASPDAC)*, January 2010.

[9] D. Kesler, B. Deka, and R. Kumar. A hardware acceleration technique for gradient descent and conjugate gradient. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, june 2011.

[10] R. Kumar. Stochastic processors. In *NSF Workshop on Science of Power Management*, March 2009.

[11] S. Narayanan, J. Sartori, R. Kumar, and D.L. Jones. Scalable stochastic processors. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010.

[12] A Nemirovski, A Juditsky, G Lan, and A Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on Optimization*, 19(4), 2009.

[13] J. Sartori and R. Kumar. Architecting processors to allow voltage/reliability tradeoffs. In *CASES*, 2011.

[14] J. Sartori and R. Kumar. Compiling for energy efficiency on timing speculative processors. In *the 49th Design Automation Conference(DAC)*, June 2012.

[15] N. Shanbhag, R. Abdallah amd R. Kumar, and D. Jones. Stochastic computation. In *the 47th Design Automation Conference(DAC)*, June 2010.

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

[17] J. Sloan, R. Kumar, G. Bronevetsky, and T. Kolev. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012*, 2012-july 1 2012.

[18] Wikipedia. Mandelbrot set, 2011. `http://en.wikipedia.org/wiki/Mandelbrot_set`.