Stochastic Computing: Embracing Errors in Architecture and Design of Processors and Applications

John Sartori, Joseph Sloan, and Rakesh Kumar
University of Illinois at Urbana-Champaign

ABSTRACT

As device sizes shrink, device-level manufacturing challenges have led to increased variability in physical circuit characteristics. Exponentially increasing circuit density has not only brought about concerns in the reliable manufacturing of circuits, but has also exaggerated variations in dynamic circuit behavior. The resulting uncertainty in performance, power, and reliability imposed by compounding static and dynamic non-determinism threatens to halt the continuation of Moore's law, which has been arguably the primary driving force behind technology and innovation for decades. As the marginal benefits of technology scaling continue to languish, a new vision for *stochastic computing* has begun to emerge. Rather than hiding variations under expensive guardbands, designers have begun to relax traditional correctness constraints and deliberately expose hardware variability to higher levels of the compute stack, thus tapping into potentially significant performance and energy benefits, while exploiting software and hardware error resilience to tolerate errors. In this paper, we present our vision for design, architecture, compiler, and application-level stochastic computing techniques that embrace errors in order to ensure the continued viability of semiconductor scaling.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance
General Terms: Design, Algorithms, Theory, Reliability, Performance
Keywords: stochastic computing, error resilience

1. INTRODUCTION

The primary driver for innovations in computer systems has been the phenomenal scalability of the semiconductor manufacturing process, governed by Moore's law, that has allowed us to literally print circuits and systems growing at exponential capacities for the last three decades. Moore's law has come under threat, however, due to the resulting exponentially deteriorating effects of material properties on chip reliability and power. Non-determinism is due to different transistors being doped or etched differently during the manufacturing process, as well as workload and environmental variations that cause uncertain timing characteristics on the chip. The most immediate impact of non-determinism is decreased chip yields. A growing number of parts are thrown away since they do not meet timing and power-related specifications. Area and power costs are enormous as well. Rough calculations indicate that the average power cost of dealing with non-determinism in today's server-class processor is 35%. This is expected to get worse with late-CMOS / post-CMOS technologies. Moreover, hardware continues to be unreliable in spite of these costs, affecting resilience in harsh, radiation-heavy environments. A recent DARPA Exascale study [8] suggests that if an exascale system were to be constructed using present components, failures will be common, with estimates ranging from one failure every 37 minutes [8] to one failure every 3 minutes [9]. Corrupted data and computations will also become more common [5], which has the potential to compromise the correctness of application results. Clearly the status quo cannot continue. Left unaddressed, the entire computing and information technology industry will soon face the prospect of parts that neither scale in capability nor cost.

Paradoxically, the problem is not non-determinism, per se, but the current contract between hardware and software. This contract guarantees that hardware will return correct values for every computation, under all conditions. In other words, we demand hardware to be overdesigned to meet the mindsets in computer systems and software design of the past. Guardbands imposed upon hardware result in increased cost, because getting the last bit of performance incurs too much area and power overhead, especially if performance is to be optimized for all possible computations. Conservative guardbands also leave enormous performance and energy potential untapped, since the software assumes lower performance than what a majority of instances of that platform may be capable of attaining most of the time.

As the marginal benefits of technology scaling continue to languish, a new vision for computing has begun to emerge. Rather than hiding variations under expensive guardbands, designers have begun to relax traditional correctness constraints and deliberately expose hardware variability to higher levels of the compute stack, thus tapping into potentially significant performance and energy benefits, but also opening up the potential for errors. Rather than paying the increasing price of hiding the true, stochastic nature of hardware, emerging *stochastic computing* techniques [17, 21, 18, 16, 28,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'11, October 9–14, 2011, Taipei, Taiwan.
Copyright 2011 ACM 978-1-4503-0713-0/11/10 ...$10.00.

27, 26] account for the inevitable variability and exploit it to increase efficiency.

In this paper, we present our vision of stochastic computing, encompassing multiple levels of the compute stack, from design-level techniques that manipulate the error distribution of hardware to effectively and efficiently exploit error resilience, to architectural optimizations that enable processors to make efficient energy / reliability tradeoffs, to compiler optimizations that increase the efficiency of programmable stochastic processors, and algorithmic optimizations that make applications robust to errors.

2. DESIGN-LEVEL TECHNIQUES FOR STOCHASTIC COMPUTING

In this section, we discuss design-level techniques for stochastic computing that manipulate the error distribution of hardware to improve the efficiency of computing in the face of errors. The extent of energy benefits provided by stochastic computing techniques depends on the error rate of the error resilient design. For example, in the context of voltage overscaling, benefits depend on how the error rate changes as voltage decreases. If the error rate increases steeply, only meager benefits are possible [16] due to high error recovery overheads or limited scalability. If the error rate increases gradually, greater benefits are possible.

While energy benefits depend on the error rate of the processor, the error rate itself depends on the timing slack and activity of the paths of the processor in the context of overscaling. Figure 1 shows an example slack distribution. The slack distribution is a histogram that shows the number of paths in a circuit at each value of timing slack. As voltage is scaled down, path delay increases, and path slack decreases. Likewise, as frequency is scaled up, the clock period gets shorter, and path slack decreases. The slack distribution shows how many paths can potentially cause errors because they have negative slack (shaded region). Negative slack means that path delay is longer than the clock period.

From the slack distribution, it is clear which paths can cause errors at a given voltage or frequency. In order to determine the error rate of a processor, the activity of the negative slack paths must be known. A negative slack path causes a timing error when it toggles. Therefore, knowing the cycles in which any negative slack path toggles reveals the number of cycles in which a timing error occurs.

For example, consider the circuit in Figure 2 consisting of two timing paths. P_1 toggles in cycles 2 and 4, and P_2 toggles in cycles 4 and 7. At voltage V_1, P_1 is at critical slack, and P_2 has 3ns of timing slack. Scaling down the voltage to V_2 causes P_1 to have negative slack. Since P_1 toggles in 2 of 10 cycles, the error rate of the circuit is 20%. At V_3, the negative slack paths (now P_1 and P_2) toggle in 3 of 10 cycles, and the error rate is 30%. Note that although two paths violate timing in cycle 4, recovery for both violations happens simultaneously, so the cost of recovery is the same as it would be for a single violation.

2.1 Recovery-Driven Design

Conventional hardware is designed and optimized using techniques that aim to ensure correct operation of the hardware under all possible PVT variations. Better-than-worst-case design techniques [2] save power by eliminating guard-

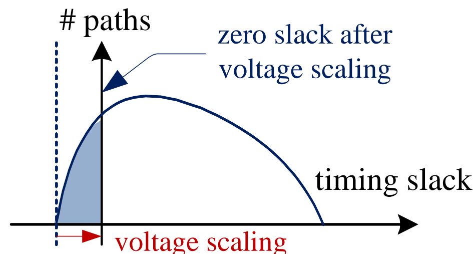

Figure 1: Voltage or frequency scaling shifts the point of critical slack. Paths in the shaded region have negative slack and cause errors when toggled.

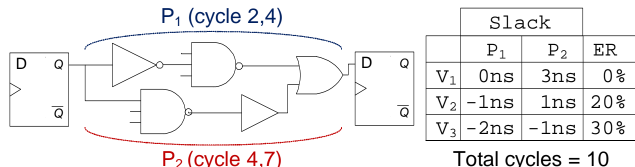

Figure 2: Slack and activity distributions determine the error rate. Error rate increases with overscaling, since more paths cause errors.

bands, but are still aimed at ensuring correct hardware operation under nominal conditions.

Recovery-driven design [17] contends that the use of error resilient design techniques should fundamentally change the way that hardware is designed and optimized. I.e., given that mechanisms exist to tolerate hardware errors, rather than designing and optimizing hardware for correct operation hardware should be optimized for a target error rate, even during nominal operation. A recovery-driven design deliberately allows timing errors ([15, 7]) to occur during nominal operation, while relying on an error resilience mechanism to tolerate these errors. The error rate target of a recovery-driven design is chosen such that any errors produced in the optimized design can be gainfully tolerated by hardware [7, 13] or software-based [15] error resilience. The expectation behind recovery-driven design is that the "underdesigned" hardware will have significantly lower power or higher performance than hardware optimized for correct operation. Also, because errors are allowed, the design methodology can exploit workload-specific information (e.g, activity of timing paths, architecture-level criticality of timing errors, etc.) to further maximize the power / performance benefits of underdesign.

The energy benefits of exploiting error resilience are maximized by redistributing timing slack from paths that cause very few errors to frequently-exercised critical paths that have the potential to cause many errors. This reduces the error rate at a given voltage or frequency, and hence reduces minimum supply voltage and power or increases maximum frequency and performance for a target error rate. Figure 3 demonstrates the goal of recovery-driven design.

Recovery-driven design work [17] proposes a heuristic implementation based on slack redistribution. The heuristic targets energy reduction with a two-pronged approach – extended voltage scaling through cell upsizing on critical and frequently-exercised circuit paths (*OptimizePaths*), and leakage power reduction achieved by downsizing cells in non-critical and infrequently-exercised paths (*ReducePower*). The heuristic searches for the combination of these two techniques that provides the lowest total power consumption for

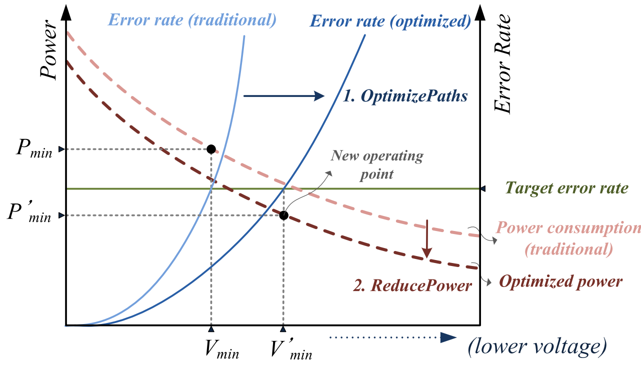

Figure 3: A recovery-driven design flow redistributes slack from infrequently-exercised paths to frequently-exercised paths and performs cell downsizing to accommodate average-case conditions. These optimizations reduce the power consumption of a circuit and extend the range that voltage can be scaled before a target error rate is exceeded. The combination of these factors produces a design with significantly reduced power consumption. [17]

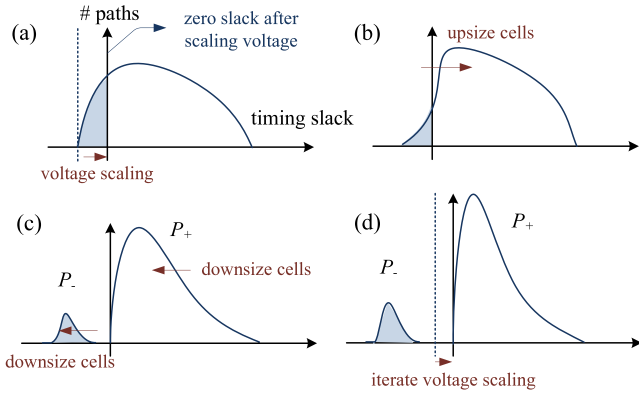

Figure 4: The power minimization heuristic for recovery-driven design reshapes a circuit's path slack distribution by redistributing slack from paths that rarely toggle to paths that toggle frequently. This extends the amount of ovescaling that can be done for a given error rate target. [17]

a circuit, by performing path optimization and power reduction at each voltage step and then choosing the operating power at which minimum power is observed.

Figure 4 illustrates the evolution of a slack distribution throughout the stages of the power minimization procedure. Each iteration begins as voltage is scaled down by one step (a). After partitioning the paths into sets containing positive and negative slack paths, *OptimizePaths* attempts to reduce the error rate by increasing timing slack on negative slack paths (b). Next, the heuristic allocates the error rate budget by selecting paths to be added to the set of negative slack paths, and downsizes cells to achieve area / power reduction while respecting the dichotomy between negative and non-negative slack paths (c). This cycle is repeated over the range of potential operating voltages to find the minimum power netlist and corresponding voltage (d). In Figure 4, P_+ is the set of paths that must have non-negative slack after power reduction, and P_- is the set of paths that are allowed to have negative slack.

By optimizing a design to achieve maximum efficiency at a non-zero error rate rather than for correct operation, recovery-driven design achieves significantly increased energy efficiency for voltage overscaling-based timing specu-

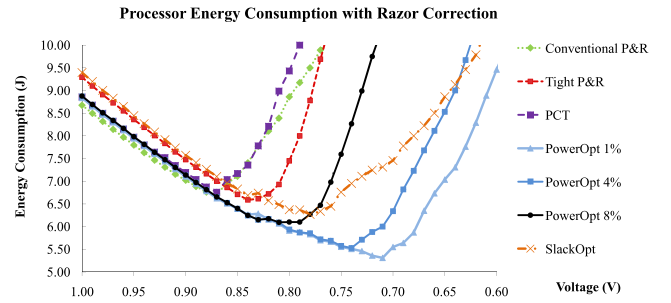

Figure 5: By optimizing the processor for a certain error rate, then correcting the errors with an error-resilience mechanism, recovery-driven design achieves additional energy savings over competing approaches. Results are shown for a processor that uses Razor-based timing speculation. The additional energy benefits of the recovery-driven design are due to extended overscaling and reduced area and leakage.

lation [17]. Figure 5 compares recovery-driven design for a Razor-based [7] timing speculative processor against several alternative design styles. Conventional P&R and Tight P&R are highly-optimized conventional CAD flows with different timing constraints. PowerOpt X% denotes a recovery-driven design that targets an error rate of X%, and SlackOpt denotes a gradual slack design [16]. PCT denotes a path constraint tuning approach similar to the technique used by BlueShift [14]. Figure 5 demonstrates that adopting a recovery-driven design style enables additional energy savings over competing techniques, due to extended overscaling and lower area and leakage. Designing the processor for the error rate target at which Razor operates most efficiently allows the recovery-driven design to extend the range of voltage scaling from 0.84V for the best "designed for correct operation" processor to 0.71V for the recovery-driven design optimized for an error rate of 1%, affording an additional 19% energy reduction.

2.2 Gradual Slack Design

Gradual slack design [18] is an extension of recovery-driven design that reshapes the slack distribution of a design to create a gradual failure characteristic, rather than the typical critical wall. While error rate-optimized, recovery-driven designs achieve better energy efficiency at a single target error rate, gradual slack designs have the ability to trade reliability, throughput, or output quality for energy savings over a range of error rates. Gradual slack design techniques are used to create *soft processors* [16] – processors that degrade gracefully under variability. Figure 6 describes the optimization approach for gradual slack design.

3. ARCHITECTURAL PRINCIPLES FOR STOCHASTIC PROCESSORS

Architecture-level stochastic optimization work [26] demonstrates that the error distribution of a design depends strongly on architecture and that the error distribution of a design that has been architected for error-free operation may limit the ability to perform voltage / reliability tradeoffs. Thus, optimizing architecture for correctness can result in significant inefficiency when the actual intent is to exploit error resilience. On the other hand, architectural optimizations can be used to increase the efficiency of a design that employs error resilience. In other words, one would make different, sometimes counterintuitive, architectural design choices to

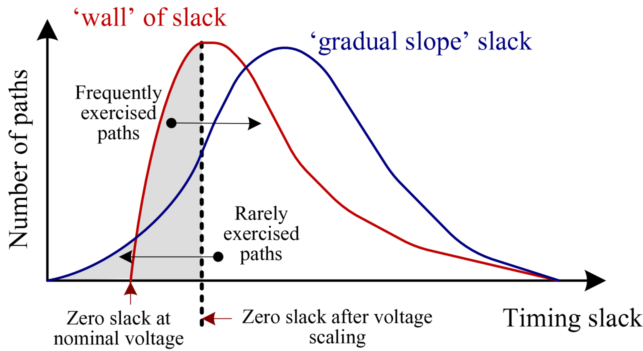

Figure 6: The goal of a gradual slack [18] or soft processor [16] design is to transform a slack distribution characterized by a critical "wall" into one with a more gradual failure characteristic. This allows performance / power tradeoffs over a range of error rates, whereas conventional designs are optimized for correct operation and recovery-driven designs, are optimized for a specific target error rate.

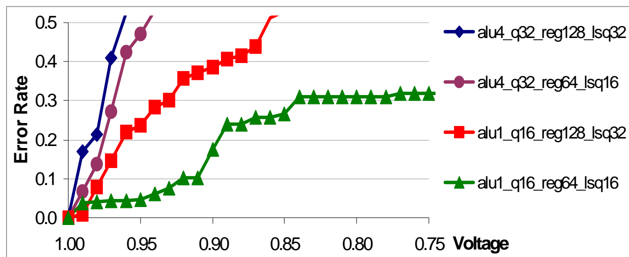

Figure 7: Different microarchitectures exhibit different error rate behaviors, demonstrating the potential to influence the energy efficiency of a timing speculative architecture through microarchitectural optimizations. (Notations describe the number of ALUs, issue queue length, number of physical registers, and load store queue size for a microarchitecture.)

optimize the error distribution of a design to exploit error resilience.

Figure 7 demonstrates that microarchitecture can have a significant effect on error rate. The figure shows the error rate behavior for four different microarchitectural configurations of the FabScalar [6] processor. Each microarchitecture has a significantly different error rate behavior, demonstrating that slack, activity, and error rate indeed depend on microarchitecture. Differences in the error rate behavior of different microarchitectures are due to several factors. First, changing the sizes of microarchitectural units like queues and register files changes logic depth and path delay regularity, which in turn effects the slack of many timing paths. Secondly, varying architectural parameters such as superscalar width has a significant effect on logic complexity [23]. Complexity, fanout, and capacitance change path delay sensitivity to voltage scaling and cause the shape of the slack distribution to change. Finally, changing the architecture alters the activity distribution of the processor, since some units are stressed more heavily, depending on how the pipeline is balanced. High activity in units with many critical paths can cause error rate to increase more steeply. Likewise, an activity pattern that frequently exercises longer paths in the architecture limits overscaling.

Typically, energy-efficient processors devote a large fraction of die area to structures with very regular slack distributions, such as caches and register files [24]. These structures

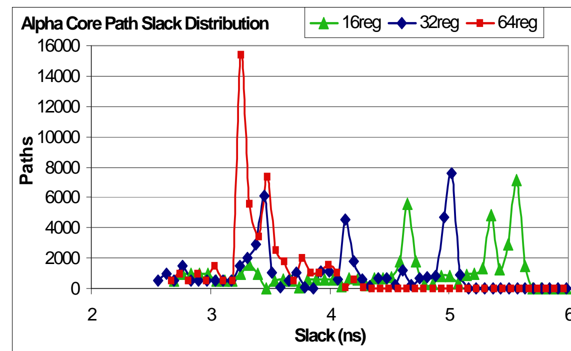

Figure 8: The sizes of regular structures can significantly influence the slack distribution of a microarchitecture. Reducing the size of the register file (a regular structure) increases the spread of the slack distribution, resulting in fewer paths bunched around the point of critical slack.

typically have high returns in terms of energy efficiency (performance/watt) during correct operation.

While regular structures are architecturally attractive in terms of processor efficiency for correct operation, such structures have slack distributions that allow little room for overscaling. This is because all paths in a regular structure are similar in length, and when one path has negative slack, many other paths also have negative slack. For example, consider a cache. Any cache access includes the delay of accessing a cache line, all of which have the same delay. So, no matter which cache line is accessed, the delay of the access path will be nearly the same. Compare this to performing an ALU operation, where the delay can depend on several factors including the input operands and the operation being performed. When many paths fail together, error rate and recovery overhead increase steeply upon overscaling, limiting the benefits of timing speculation. Reducing the number or delay of paths in a regular structure can reshape the slack distribution, enabling more overscaling and better timing speculation efficiency.

For an example Alpha core [32], the register file is the most regular critical structure. Figure 8 shows slack distributions for the Alpha core with different register file sizes. As the size of the register file increases, the regularity of the slack distribution also increases, as does the average path delay. Figure 8 confirms that the spread of the slack distribution decreases with a larger register file. Additionally, path slack values shift toward zero (critical) slack due to the many critical paths in the register file.

Architectural design decisions that reshape the slack distribution by devoting less area to regular structures or moving regular structures off the critical path can enable more overscaling and increase energy efficiency for timing speculative processors. In other words, additional overscaling enabled by architectures with smaller regular structures can outweigh the energy benefits of regularity when designing a resilience-aware architecture. Since regularity-based decisions may also impact power density, yield, and performance, architectural decisions should consider these constraints in addition to the chosen optimization metric.

Figure 9 shows energy consumption for the Alpha core with Razor-based timing speculation, confirming that the architecture with a smaller register file exploits timing error resilience more efficiently. The 16-register architecture reduces energy by 21% with respect to the optimal architec-

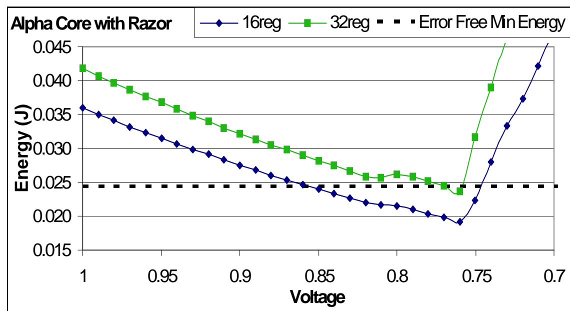

Figure 9: The 16-register design, having reduced regularity and activity, achieves significant energy savings with Razor, while the 32-register design, which was optimal for correct operation, achieves almost no benefit.

ture for correctness, while the optimal error free architecture barely achieves any energy savings (2%) when using Razor.

Other architectural optimizations that impact logic complexity, regularity, and utilization have also been demonstrated to improve the potential for energy savings from voltage / reliability tradeoffs [26].

4. COMPILER OPTIMIZATIONS FOR STOCHASTIC COMPUTING

Altering the error distribution of a timing speculative processor has the potential to increase the energy benefits of exploiting error resilience. As discussed above, architecture and design-level optimizations that modify the slack and activity distributions of a processor can increase the energy efficiency and performance of a timing speculative design [16, 18, 17, 26]. Since the program binary, in conjunction with the processor architecture, determines a processor's activity distribution, *timing speculation-aware compilation* [27] can also significantly increase the energy efficiency of a design that exploits timing error resilience.

The goal of TS-aware binary optimizations is to minimize the energy consumption of a timing speculative processor by manipulating its activity distribution to reduce the error rate for a given voltage or reduce the cost of error recovery. As such, the optimizations fall into one of the following four categories.

Critical Path Avoidance.

The first category of TS-aware optimization techniques change the set of frequently exercised paths in a processor to avoid activating the longest paths. Since these paths are the first to have negative slack when the processor is overscaled, throttling their activity reduces early onset timing violations. Optimizing the instruction stream can *prevent* activity patterns that exercise the static critical paths of hardware structures, resulting in more timing slack, on average, and more room for overscaling.

Activity Throttling.

The second category of techniques reduces error rate by throttling activity in structures of the processor that cause the most errors. A program binary, in conjunction with the processor microarchitecture, determines how often different processor units are stressed. When a structure that exercises its critical paths for most instructions also has high utilization, the structure generates many errors when the processor is overscaled. Thus throttling the activity of such structures can significantly reduce the processor error rate for a given voltage, allowing the processor to scale to a lower voltage for a given error rate.

Overlapping Errors.

The previous categories deal with error avoidance. Another way to increase TS efficiency is to reduce the cost of error recovery. For many error recovery mechanisms, such as rollback, flushing, counterflow, and replay, when multiple timing errors occur in the same cycle, a single error recovery process can correct all the errors. The third category of compiler optimizations for stochastic computing focuses on overlapping the occurrence of timing violations so that multiple errors share a single error recovery process.

Recovery Cost-Aware Scheduling.

The final category of techniques aim to schedule accesses to processor structures such that activity is inversely proportional to error recovery cost. This strategy aims to utilize timing speculation more efficiently by avoiding high cost errors.

The exact optimization techniques that are most effective for a processor depend on the processor architecture and the exact modules of the processor that cause the most errors. Different pipeline stages cause errors at different rates, depending on their slack and activity distributions. Figure 10 shows the static slack distributions for the pipeline stages that cause the most errors in one specific superscalar pipeline implementation (FabScalar [6]). Figure 10 demonstrates that two stages in particular – the issue queue (IQ) and the load store unit (LSU) have the highest number of critical paths. Based on Figure 10, one might expect that IQ, having many more critical paths than all other modules, would produce the most errors in the processor. However, the static slack distribution only shows the *potential* for paths to cause errors. To know how many errors the paths actually generate, their activity must also be known. Figure 11 shows activity-weighted, *dynamic* slack distributions that quantify the sum of toggle rates for all the paths at each value of timing slack (activity from SPEC benchmarks). The more timing critical *activity* a module has, the more errors it is likely to cause. From Figure 11, it is clear that the LSU dominates the error distribution of the FabScalar architecture.

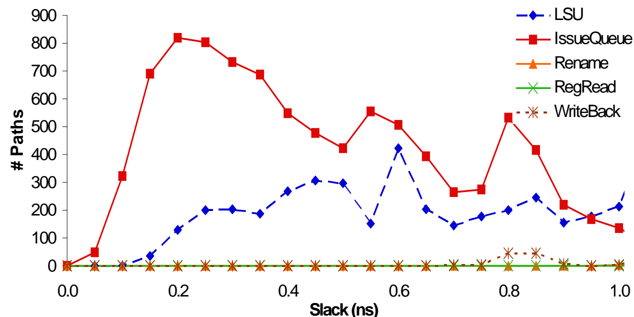

Figure 10: The static slack distributions for the pipeline stages show how many critical paths they have. The static slack distribution does not, however, provide information about how often the paths toggle, which is essential in characterizing error rate.

LSU delay depends on program characteristics for several reasons. The primary reason is that the store-to-load forwarding path is on the static critical path of the LSU. Since many RAW dependencies in the code lead to more for-

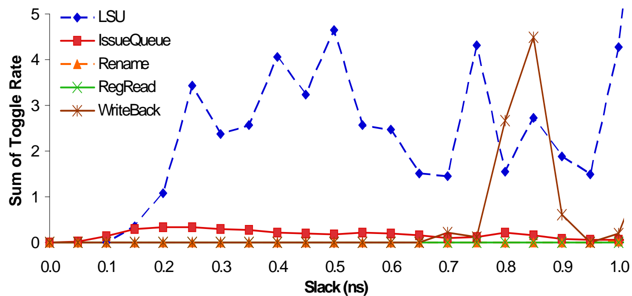

Figure 11: The activity weighted (dynamic) slack distributions for different pipeline stages indicate how much timing critical activity they have, and by extension, how frequently they will produce errors for a given level of overscaling.

warding, the timing error rate will be higher for code with a relatively large number of RAW dependencies. Program characteristics also determine the utilization of the LQ and the SQ, which, in turn, dictates access delays.

As described above, activity on the static critical paths of the LSU can be reduced by avoiding dependent memory operations and scenarios that cause the LSQ to fill up. This can enable significantly deeper voltage overscaling, since the LSU is the source of many timing violations.

Loop unrolling is a classic compiler optimization that can eliminate and spread out loop carried dependencies, and thus has the potential to reduce LSU delay. TS-aware compilation provides a new use for unrolling – avoiding errors to increase the efficiency of TS by grouping often independent instructions (like vector math) and eliminating often dependent instructions (like branches and index updates). Unrolling also allows optimization of register allocation over multiple loop iterations that can eliminate loads and stores, thus reducing pressure on the LSU.

Figure 12 demonstrates that the compiling to avoid dependencies and activity on the forwarding paths with loop unrolling alters the activity distribution of the LSU and significantly reduces the error rate of the processor. Figure 12 shows the activity-weighted (dynamic) slack distribution of the LSU for an unoptimized summation loop and an unrolled version of the loop. The figure demonstrates that activity on the critical paths of the LSU is greatly reduced when loop unrolling is used. The dependencies in the original code are spread apart so that they do not simultaneously reside in the LSQ, and forwarding is not required.

Figure 13 shows the error rate of the processor when executing the two code sequences. Unrolling significantly reduces the error rate by reducing activity on the forwarding paths in the LSU. This error rate reduction enables additional overscaling and results in a substantial energy reduction for a Razor-based TS processor, as shown in Table 1.

In the normal, error-free case, the same unrolled loop provides a meager energy benefit of only 2%, on average, as unrolling causes dynamic power to increase significantly, even as it increases throughput. This further demonstrates the need for TS-aware compiler analysis and optimization.

Other compiler optimizations that target critical path avoidance and critical activity throttling have also been demonstrated to be effective in reducing the error rate and improving the energy efficiency of error resilient processors [27]. Balancing the amount of instruction-level parallelism exposed to the processor backend, and splitting or fusing loops have potential to manipulate the activity distribution of the processor and throttle or avoid activity on error prone paths. Additionally, TS-aware compilation work has demonstrated that gcc optimization flags can be manipulated to influence the energy efficiency of binaries on TS processors.

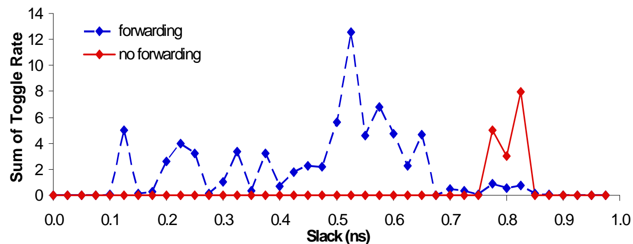

Figure 12: Since forwarding paths are critical in the LSU, eliminating dependencies and the need for store-to-load forwarding reduces activity on the critical paths of the LSU.

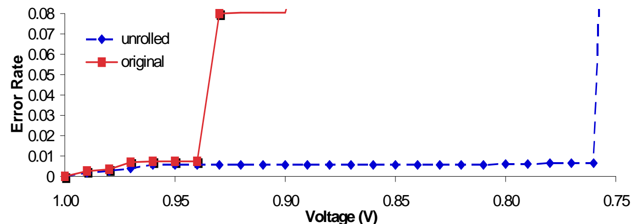

Figure 13: Loop unrolling reduces activity on LSU forwarding paths, resulting in a significant error rate reduction.

5. DESIGNING APPLICATIONS FOR ROBUSTNESS

Algorithmic approaches for error correction allow us to execute applications on a stochastically correct processor (*stochastic processor*) [20, 16, 28, 17] by replacing the original computation with one that may take slightly longer to complete. We call this general methodology for converting applications into a form that may be robust to variation-induced errors, *application robustification* [30].

One approach for *application robustification* presented in this section consists of reformulating applications as stochastic optimization problems. 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. This approach is quite generic, since linear programming, which is P-complete, can be implemented this way. In fact, we present examples for both fragile (e.g. combinatorial problems, sorting, minimum cut, graph matching) and intrinsically robust applications (e.g. solving *least squares* problems).

The goal of the numerical-optimization approach 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.

Table 1: Razor-based TS and error-free energy savings (%) for loop unrolling. (ss = superscalar width

CORE	original	unrolled	unrolled error-free
ss1	11.8	43.1	1.6
ss2	6.4	20.8	2.0
ss4	4.0	42.9	3.2

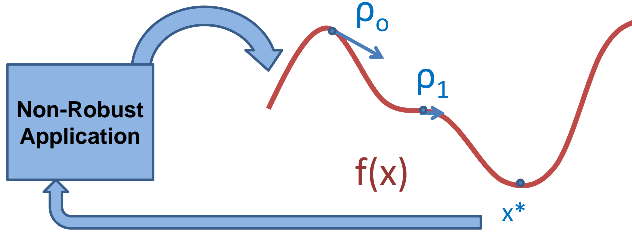

Figure 14: ***Application robustification*** **involves converting an application to an unconstrained optimization problem, where the minimum corresponds to the output of the original non-robust application.**

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 14, are

- 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. We rely on gradient descent as the primary optimization engine to construct algorithms that tolerate noise in the CPU's numerical units. 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.

To minimize a cost function $f : \mathbf{R}^d \rightarrow \mathbf{R}$, gradient descent generates a sequence of steps $x^1 \ldots x^i \in \mathbf{R}^d$ via the iteration

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

starting with a given initial iterate $x_0 \in \mathbf{R}^d$. The vector $\nabla f(x^{i-1})$ is a gradient 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, which is where variation-induced errors may occur.

The suitability of gradient descent for stochastic processors is due to the fact that under various assumptions of local convexity on f, x^i is known to always approach the true optimum, even under errors, as iterations progress [30]. As long as the ∇f approximation is unbiased, gradient descent can eventually extract a solution with arbitrarily high accuracy.

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

$$\underset{x \in \mathbf{R}^d}{\text{minimize}} \quad f(x), \text{s.t. } g(x) \leq 0, h(x) = 0 \quad (2)$$

for some functions f, g, and h. We rely on an exact penalty method to convert constrained problems into unconstrained problems that can be solved by gradient descent [3, 22]:

$$f(x) + \mu \sum_i |h_i(x)| + \mu \sum_j [g_j(x)]_+ . \quad (3)$$

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 $f(x) + \mu \ \sum_i h_i(x)^2 + \mu \ \sum_j [g_j(x)]_+^2$ also hold [25].

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. To alleviate some of the artifacts of ill-conditioned problems, we also consider some additional variants of the basic gradient decent algorithm by using *momentum* in the search direction calculation, different step sizing strategies, and *preconditioning* of the objective function [30].

For certain problems, such as *least squares*, the structure of the problem can be exploited further to construct better search directions and step sizes. One approach is the conjugate gradient (CG) method [12]. 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). CG is guaranteed to converge in at most n iterations without errors (where n is the number of variables in the problem). The convergence of CG under noise is also well understood [29].

5.1 Application Transformations for Robustness

Transforming a given problem into its variational form (2) is often immediately obvious from the definition of the problem. 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 5.2 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 $\nabla f(x) = A^\top (Ax - b)$. Filtering a signal with an *infinite impulse response (IIR)* filter, a basic operation in signal processing, is another example application similar to the **least squares** problem.

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 $v = [1 \ldots n]^\top$ [4]. 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

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

The corresponding unconstrained exact quadratic penalty

function is

$$f(X) = -v^\top X u + \lambda_1 \sum_{ij} [X_{ij}]_+^2 + \lambda_2 \sum_i \left[\sum_j X_{ij} - 1\right]_+^2 + \lambda_2 \sum_j \left[\sum_i X_{ij} - 1\right]_+^2 \quad (5)$$

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

$$[\nabla f(X)]_{ij} = -u_i v_j + 2\lambda_1 [X_{ij}]_+ + 2\lambda_2 \left[\sum_j X_{ij} - 1\right]_+ + 2\lambda_2 \left[\sum_i X_{ij} - 1\right]_+ . \quad (6)$$

Note that sorting is traditionally not thought of as an application that is error tolerant. Our methodology produces an 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 [11]. 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*.

To summarize, the 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., *Least Squares*, etc.

5.2 Experimental Results for Application Robustification

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 [10] soft core processor. Error injection was done using a software-controlled fault injector FPGA module, which 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 [19]. The time between corruptions was drawn using a uniform distribution generated by a linear feedback shift register. This fault model is a surprisingly reasonable approximation of voltage overscaling-induced errors in the FPU. To calculate the energy benefits from application robustifcation, we also used circuit-level simulations to calculate the relationship between voltage and error rate for the FPU.

Gradient Descent

To explore the feasibility of the proposed approach to provide robustness and energy benefits, we evaluated stochastic gradient descent (SGD) on the problems, *least squares*, *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 an adaptive stepping strategy called *aggressive stepping (AS)* In the figures, 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 *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. 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. *Bipartite graph matching* was implemented using the OpenCV library [1].

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.

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 15) 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 16), on the other hand, performs better with linear step size scaling. It is also possible to get highly accurate results, within 10^{-6}% of the exact value computed offline with an SVD-based baseline. The benefits of aggressive stepping for the applications are most pronounced for low fault rates ($< 1\%$).

Bipartite graph matching (Figure 17) using $10,000$ 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%. When considering gradient decent with the additional variants described in the beginning of Section 5, the *Bipartite graph matching* application is also shown to have high accuracy(100%) in Figure 18. These additional variants of gradient descent also be used to solve problems which have poorly conditioned objective functions several orders of magnitude faster.

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 of iterations required for convergence. The Conjugate Gradient method, on the other hand, allows efficient generation of

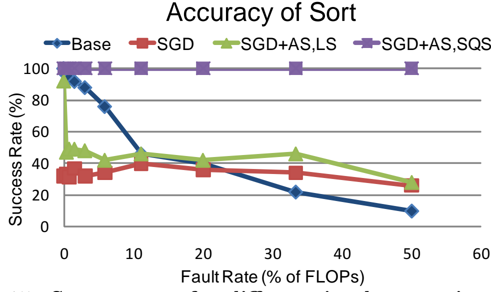

Figure 15: Success rate for different implementations of *sorting* as a function of fault rate. 10000 Iterations.

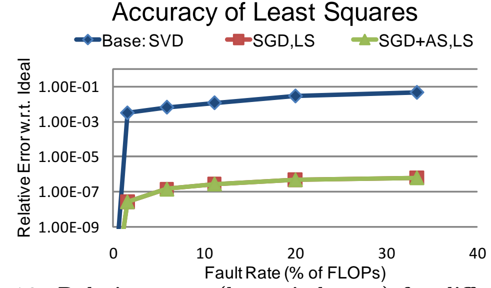

Figure 16: Relative error (lower is better) for different implementations of *least squares*. SQS errors > 1.0. 1000 Iterations.

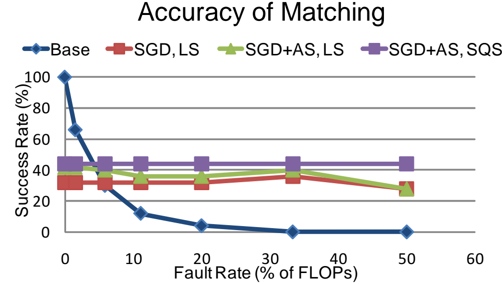

Figure 17: Success rate for different implementations of *bipartite graph matching* as a function of fault rate.10000 Iterations.

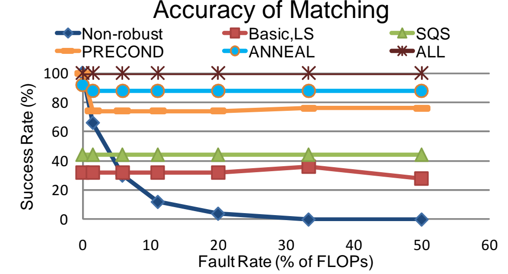

Figure 18: The effect of enhancements to gradient descent on the success rate of *bipartite graph matching*. 10000 Iterations.

conjugate directions by taking a linear combination of the steepest descent direction and the previous directions. In general, the CG method can guarantee convergence in at most n iterations for an $Ax = B$ problem where n is the dimension of x. Figure 19 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 cannot be met using CG). This is because it becomes possible to scale down the voltage and the number of iterations concurrently. Figure 20 shows the normalized energy results for the FPU for the *least squares* problem. 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.

5.3 Algorithmic Approximate Correction

An alternative approach for designing applications for stochastic processors focuses on algorithmic techniques for approximate correction, relying on the inherent fault tolerance that many applications contain within. The general problem formulation is then as follows: 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.

We'll now describe one example of algorithmic fault correction for a common linear algebra operation, the matrix-vector (MV) product [31]. Given a MV product($v = Au$) 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 vector error ($e = v' - v$), and improving the accuracy (i.e. RMS $\|v' - v\|^2$). Therefore, we propose an alternative and simpler technique for correction. The approximate correction technique for the MV product involves subtracting the projection of the error onto the code space (a predetermined check vector called c). The partially corrected MV product(v'') can then be computed as follows:

$$v'' = v' - \frac{(c^T e)c}{\|c\|^2} \tag{7}$$

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

$$\|v'' - v\|^2 = \|v' - v\|^2 - \frac{(c^T e)^2}{\|c\|^2}$$
$$\|v'' - v\|^2 \leq \|v' - v\|^2 \tag{8}$$

Approximate error correction can also efficiently provision the correction technique to account for the most important faults in terms of performance and accuracy. This is important since applications typically see faults manifested in different ways. The developer has significant flexibility in the amount and types of codes chosen for the correction, depending on the accuracy targets which are desired.

6. CONCLUSIONS

Shrinking device sizes and growing static and dynamic non-determinism challenge the reliable manufacturing and

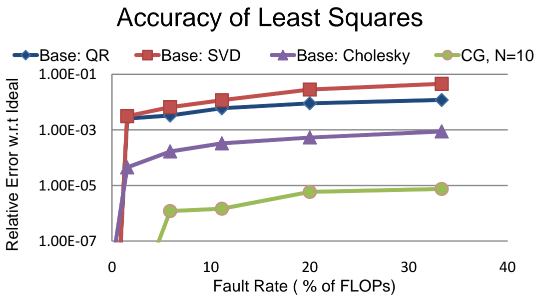

Figure 19: Accuracy for a CG-based implementation of *least squares* (10 iterations)

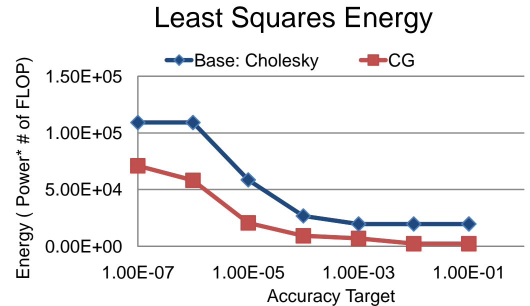

Figure 20: Energy for a CG-based implementation of *least squares*

operation of circuits. As conventional design approaches that ensure determinism, even under worst case variations, become increasingly inefficient, a new paradigm for *stochastic computing* has arrived. Rather than hiding variations under expensive guardbands, stochastic designs relax traditional correctness constraints and deliberately expose hardware variability to higher levels of the compute stack, thus tapping into potentially significant performance and energy benefits, while exploiting software and hardware error resilience to tolerate errors.

In response to the challenge of growing variability, we propose stochastic computing techniques throughout the compute stack, from design-level techniques that manipulate the error distribution of hardware to effectively and efficiently exploit error resilience, to architectural optimizations that enable processors to make efficient energy / reliability tradeoffs, to compiler optimizations that increase the efficiency of programmable stochastic processors, and algorithmic optimizations that make applications robust to errors. As static and dynamic non-determinism continue to increase, stochastic computing techniques that embrace errors stand to reap significant yield and energy benefits and ensure the continued viability of semiconductor scaling.

7. ACKNOWLEDGMENTS

Authors would like to acknowledge the many collaborators, colleagues, and reviewers that helped in the development, refinement, and exploration of the ideas presented in this paper. Our work on Stochastic Computing has been generously supported by GRC, GSRC, NSF, Intel, and LLNL.

8. REFERENCES

[1] The OpenCV Library, http://http://opencv.willowgarage.com/wiki/.

[2] T. Austin, V. Bertacco, D. Blaauw, and T. Mudge. Oppotunities and challenges for better than worst-case design. *Proc. ASPDAC*, pages 2–7, 2005.

[3] D. P. Bertsekas, A. Nedic, and A. E. Ozdaglar. *Convex Analysis and Optimization*. Athena Scientific, 2001.

[4] 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.

[5] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23:374–388, 2009.

[6] N. Choudhary, S. Wadhavkar, T. Shah, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar. In *WARP*, 2009.

[7] D. Ernst, N. 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. *Proc. IEEE/ACM MICRO*, pages 7–18, 2003.

[8] B. et. al. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. Technical report, DARPA IPTO, 2008.

[9] V. S. et. al. Exascale software study: Software challenges in extreme scale systems, 2009.

[10] A. Gaisler. Leon3 processor, 2008.

[11] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. pages 136–146, 1986.

[12] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[13] B. Greskamp and J. Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. *PACT*, 2007.

[14] B. Greskamp, L. Wan, W. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles. Blueshift: Designing processors for timing speculation from the ground up. *Proc. IEEE HPCA*, pages 213–224, 2009.

[15] R. Hegde and N. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *ISLPED*, pages 30–35, 1999.

[16] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing processors from the ground up to allow voltage/reliability tradeoffs. In *IEEE HPCA*, 2010.

[17] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Recovery-driven design: A methodology for power minimization for error tolerant processor modules. In *ACM/IEEE DAC*, 2010.

[18] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Slack redistribution for graceful degradation under voltage overscaling. In *IEEE/SIGDA ASPDAC*, 2010.

[19] C. Kong. *Study of Voltage and Process Variation's Impact on the Path Delays of Arithmetic Units*. UIUC Master's Thesis, 2008.

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

[21] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. In *DATE*, 2010.

[22] 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.

[23] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. *ISCA*, 1997.

[24] Y. Pan, J. Kong, S. Ozdemir, G. Memik, and S. Chung. Selective wordline voltage boosting for caches to manage yield under process variations. In *DAC*, pages 57–62, 2009.

[25] G. D. Pillo. Exact penalty methods. In I. Ciocco, editor, *Algorithms for Continuous Optimization*, 1994.

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

[27] J. Sartori and R. Kumar. Compiling for timing error resilient processors. In *TECHCON*, 2011.

[28] N. Shanbhag, R. Abdallah, R. Kumar, and D. Jones. Stochastic computation. In *Proc. DAC*, pages 859–864, 2010.

[29] 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.

[30] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *DSN*, 2010.

[31] J. Sloan, R. Kumar, G. Bronevetsky, and T. Kolev. Algorithmic Techniques for Fault Detection for Sparse Linear Algebra. In *TECHCON*, 2011.

[32] University of Michigan. *Bug Underground*, 2007.