ASAR: Application-Specific Approximate Recovery to Mitigate Hardware Variability

Manish Gupta[1], Abbas Rahimi[1], Daniel Lowell[2], John Kalamatianos[2],
Dean Tullsen[1], Rajesh Gupta[1]
[1]UC San Diego [2]AMD Research
{manishg, abbas, tullsen, rgupta}@cs.ucsd.edu {daniel.lowell, john.kalamatianos}@amd.com

Abstract

Technology scaling in microelectronics has reached limits that are resulting in increasing variation in component design and performance characteristics. Chips and systems comprising of such components are starting to exhibit a rise in process-induced failures and soft errors. Conventional design time solutions such as conservative guardbands to hide such variations are increasingly not viable for cost and performance reasons. As an alternative, researchers have sought to expose hardware fault information to the software stack and enable a programmer to use the fault information during software development. In this work, we propose the use of *Software Recovery Blocks* (SRB) as a programming construct that enables a programmer to provide application-specific error recovery code. Recovery comes in two modes: a) rerunning or b) discarding the erroneous computation. While rerunning comes at a performance overhead, discarding erroneous computations could result in degraded output quality, giving a user two extreme operating points on the performance-quality trade-off curve. In order to exploit intermediate performance-quality trade-off points, this work proposes *approximate recovery* which is particularly beneficial to *approximate-computing* applications. Such applications offer a natural tolerance to errors and the work introduces a SRB extension called **A**pplication-**S**pecific **A**pproximate **R**ecovery (*ASAR*). *ASAR* provides 3.8%–29.9% speedup relative to rerun for six approximate-computing applications. Furthermore, the work proposes a *hybrid recovery* mechanism which allows a user to set desired output quality and exploit the performance-quality trade-off curve at a finer-granularity. Hybrid recovery uses a mixture of *ASAR* and rerun-based recovery to demonstrate 1.5%–11.6% speedup compared to only rerun, while maintaining user-specified output quality.

1. Introduction

As technology scaling results in ever smaller components it becomes expensive to produce reliable hardware. Components such as transistors no longer behave precisely with tight tolerances as to their timing or power consumption. Emerging hardware exhibits performance and power uncertainties – the effects commonly termed *variability* [16, 21]. More aggressive process technology nodes in the coming years will see increased variability, thus increasing fequency of voltage droops, timing errors, and soft errors [1, 22]. To mask the effects of variability-induced uncertainties and ensure error-free operation device and system designers use conservative voltage and frequency guardbands. These guardbands already occupy over 40% of the cycle time and lead to high active and sleep power draw. Alternatives to design guardbanding are an active area of research in microelectronic circuit design.

Hardware solutions include techniques such as redundancy [28], circuit-level techniques [13, 29], and non-trivial design-time approaches which aim to reduce architectural vulnerability factors [33]. Some of the popular software techniques involve redundant code execution [5, 17, 40, 46], checkpointing & re-execution [23], and compiler-driven vulnerability reduction [39, 48]. None of these techniques is a panacea: software-only solutions suffer more than 2x performance and memory overhead by duplicating every computation necessary for error detection. On the other hand, hardware-only solutions aim to mask every error and provide software an illusion of error-free execution that comes at a high overhead costs.

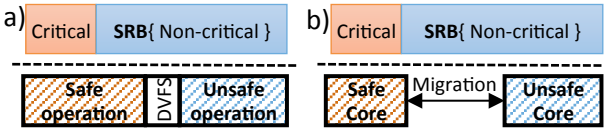


Figure 1: (a) DVFS for single core (b) Migration for multi-core.

Most of the hardware and software approaches maintain a clear separation where all hardware errors are masked from the software. This strict separation is expensive and unnecessary, especially for *approximate-computing* applications. Approximate-computing applications including search, multimedia, financial, and big-data have become key workloads that heavily influence the semiconductor industry. Conceptually, such programs have a vector of elastic outputs, and if execution is not 100% accurate, the program can still produce acceptable output quality from a user perspective [7, 10]. Relaxing the separation between hardware and software allows users to select one of many operating points on the performance-output trade-off curve offered by *approximate-computing* applications.

Our approach to addressing errors caused by variability is to make such processing part of existing software mechanisms for handling error. Therefore, we propose to expose hardware error information to the software akin to exception handling. For example, today hardware exposes a divide-by-zero or memory-access violation which allows software to terminate gracefully. We seek instead ways to recover from variability induced errors to ensure continued system operation. To achieve this goal, we extend and evaluate the use of *software recovery blocks* (SRB) for handling variability-induced hardware errors. For the code regions enclosed by SRB, hardware may be operating in an "unsafe" regime due to inadequate guardbands, for instance, a lower voltage and/or higher frequency. Any resulting errors in computation are exposed to the software as a part of SRB semantics. In case of an error, the runtime can a) rerun the code to ensure 100% accuracy, b) approximate the computation to ensure partial recovery, or c) discard sub-computations to ignore the error completely. Based on user provided output acceptability and algorithmic restrictions, the application developer can choose one of the above recovery options (a, b, and c) or a *hybrid recovery* which mixes two recovery options.

This way, approximate-computing applications can be partitioned into sections of code that require error-free operation and sections that can tolerate varying degrees of error. We label the former as *critical* and the latter as *non-critical* as shown in Figure 1. The non-critical code region is enclosed inside SRB to en-

(a) Software Recovery Blocks	(b) Proposed Extension
ensure <acceptance test> by	ensure <no hardware error> by
<primary module>	Try {<primary module>}
else by	else by
<alternative module>	Catch{<recovery module>}
else acceptance test failed	else software recovery failed

Figure 2: (a) Software Recovery Blocks (SRB) to handle programing faults (b) Extension of SRB to handle hardware errors.

able unsafe mode of operation. The unsafe modes of operation can be realized using adaptive DVFS [24] (Figure 1a), dual-voltage operation [14], and migrating execution between multi-cores [26, 45] (Figure 1b). The technique allows us to exploit the unsafe regions to either accelerate execution or run at reduced power. For example, we could either change DVFS settings at the SRB boundaries, or migrate between safe and unsafe cores. In this work we make three contributions:

I. We propose an extension of Software Recovery Blocks (SRB) to **A**pplication-**S**pecific **A**pproximate **R**ecovery (*ASAR*) which is particularly suitable for programming in a language with support for exception handling. *ASAR* extends the conventional `Try-Catch` mechanism, a high-level programming construct, to detect hardware errors and provide approximate recovery choices in software.

II. We demonstrate that *ASAR* achieves 3.8%–29.9% performance improvement relative to rerun and 5.4–84.3 percentage point increase in output quality relative to discard for six *approximate-computing* applications. Thus, ASAR provides a user with an intermediate operating point on performance-quality trade-off curve.

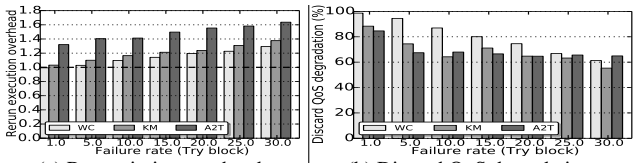
III. We introduce *hybrid recovery* combining *ASAR* and rerun-based recovery which allows user to specify a threshold on output quality. We show that hybrid recovery achieves an average 1.5%–11.6% speedup relative to rerun with an output quality of greater than user-specified threshold. Hybrid recovery enables application programmer to explore the performance-quality trade-off curve at a finer granularity.

2. Software Recovery Blocks (SRB)

Software recovery blocks (SRB) enable an application programmer to respond to software faults and are a well-known programming paradigm in real-time embedded systems. Traditional SRB implementation supports fault-tolerant programming and exceptions handling [37]. A typical recovery block structure is shown in Figure 2a. This style of programming ensures recovery from possible faults in the design of software components. Faults are detected using software acceptance tests and the program tries to ensure acceptability using `primary module`. If the `primary module` fails the acceptability test, the execution switches to `alternative module`.

SRB along with hardware support enables graceful handling of software faults, such as segmentation and divide-by-zero faults. For example, in an event of segmentation fault, software attempts to access an out-of-range memory segment. Thus hardware raises a *trap* and software exists gracefully. Faults such as segmentation faults are software-induced where hardware detects and assists software to take corrective measures. A system that doesn't have support to handle such faults may experience a system crash requiring reboot, loss of data, and silent data corruptions.

We propose extending the SRB mechanism, using `Try-Catch` mechanism, for a system that not only exposes software but also hardware faults similar to *relax-recover* mechanism by Kruijf et al. [11]. A software developer can use hardware error information and application-specific knowledge to recover from timing errors. Figure 2b shows the proposed `Try-Catch` mechanism, `Try` block executes the `primary module` using unsafe mode at a faster execution speed and higher probability of encountering hardware errors. If an error occurs during the `Try` block execution, hardware generates an interrupt. The software attempts to recover from the error using the `recovery module` – implemented inside the `Catch` block. The `Catch` block runs at safe operating conditions to ensure error free execution. Kruijf et al. evaluate two implementations for software error recovery: *rerun* and *discard* [11].



(a) Rerun timing overhead. (b) Discard QoS degradation.

Figure 3: (a) Rerun mechanism results in performance overhead of close to 2x. (b) Discarding erroneous computation may result in unacceptable output quality (QoS).

Rerun Mechanism. In the rerun or re-execution mechanism, we run the `Try` block code inside the `Catch` block until the hardware error subsides. This software compensation technique is analogous to multiple issue instruction replay [6]. Rerun ensures perfect output quality (QoS). However, rerun-based recovery comes at execution time overhead. The overhead with the single rerun for WordCount (WC), K-Means (KM), and A2Time (A2T) is shown in Figure 3a. The x-axis shows the rate of `Try` block failure and the y-axis is the total execution time normalized to the runtime of error-free execution. Execution time overhead increases with increasing failure rate with a worst case overhead of 60% for A2T and an average overhead of 27.7%.

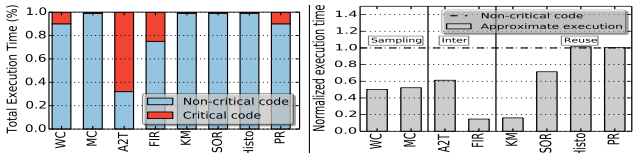
Discard Mechanism. In the discard mechanism, sub-computations by an erroneous `Try` block are dropped. The `Catch` block is programmed to account for the number of dropped sub-computations which can be used to adjust the final result. Although the discard mechanism does not incur a recovery cost, we observe a significant degradation in the output quality (QoS), as shown in Figure 3b. Our preliminary investigation reveals that the discard mechanism results in best performance at the cost of potentially unacceptable degradation in QoS. Even low error rate of 1% introduces QoS drop of 15% for A2T. We observe an average QoS drop of 27.33% and a maximum of 44%.

The rerun and discard mechanisms provide a direct way to implement `Try-Catch`. However, these mechanisms operate at two extremes on the performance-quality trade-off curve. Our extension to software recovery block, described in the next section, provides a software programmable mechanism to exploit intermediate performance-quality trade-off points for *approximate-computing* applications.

3. Application-Specific Approximate Recovery

We extend the SRB mechanism described in Section 2 to explore intermediate performance-quality trade-off points using approximate recovery. The use of a particular approximate recovery technique such as *sampling*, *interpolate*, and *reuse* is specific to the application's algorithm. Hence, we propose and evaluate **A**pplication-**S**pecific **A**pproximate **R**ecovery (*ASAR*) to provide an approximate recovery alternative. *ASAR* provides a mechanism that lies in between two extreme recovery options, i.e. rerun and discard, targeting *approximate-computing* applications which can operate reliably by trading output quality for performance.

In *approximate-computing*, the program execution is composed of two parts, a critical part and a non-critical part [8, 14, 41]. The critical part mostly consist of setup code, configurations, system calls and I/O operations. The critical code sections cannot tolerate errors and are not good candidates for software-based recovery. Hence, critical code sections must run using safe mode to ensure error-free operation. The non-critical code sections should be side-effect free sub-computations (idempotent regions) which mostly in-



(a) Critical vs non-critical code. (b) Approximate execution.

Figure 4: (a) On average applications have 86% of dynamic non-critical code which can be executed at aggressive operating point. (b) Application-specific approximations. Sampling [WC, MC], Interpolate [A2T, FIR], and Reuse [KM, SOR, Histo, PR]

clude hot code regions, or inherently error-tolerant floating point operations. Several compiler-based automated techniques can be employed to ensure idempotent property for non-critical code sections [11, 12, 44]. The impotence of non-critical regions is also essential when using the rerun and discard recovery techniques. Thus, non-critical code section can be enclosed inside a `Try` block to run using unsafe mode in order to achieve higher performance and/or lower power. In case of an error, the `Catch` block recovers using rerun (paying maximum recovery overhead) or discard (suffering maximum QoS degradation), as shown in Section 2. Because non-critical code sections do not have to be precise, an approximate software implementation can reduce both the recovery overhead and QoS degradation relative to the rerun and discard methods.

Critical vs. Non-critical code. Performance and energy gains by execution under unsafe operating conditions for non-critical code regions are only possible if applications spend substantial portion of their time in non-critical regions. Figure 4a shows the dynamic ratio of the critical vs. non-critical code normalized to the total execution time for eight applications. Six out of eight applications, WordCount (WC), K-Means (KM), SOR, MonteCarlo (MC), Histogram (Histo), and PageRank (PR) spend 90% of their time in non-critical code region. Ranger et al. [38] observe similar ratios of critical vs. non-critical code sections for WordCount, K-Means and Histogram. A2Time (A2T) and FIR spend 30% and 75% of their time in non-critical region respectively. A2T has significant setup computations and FIR has instructions that arrange coefficients before every call to non-critical code section. Overall, we observe that eight applications spend 86% of their total execution time in the non-critical regions. Thus, accelerating non-critical regions, via unsafe modes, for these applications can potentially result in performance and energy gains.

The non-critical code region executing under unsafe conditions may encounter an error. To recover from the error the rerun-based recovery re-execute the non-critical code inside `Catch` block using safe operating conditions which amounts a substantial recovery overhead. In order to reduce the software recovery time we introduce application-specific approximate recovery (*ASAR*). We implement *ASAR* using *sampling (samp)*, *interpolation (inter)*, and *reuse* to cover all eight applications used in this study. The core mechanisms presented in this work are not restricted to the above three approximate recovery techniques, and an application developer can implement their choice of approximate recovery. In our *ASAR* implementation, `Try-Catch` blocks can be considered as process nodes viewed from a standard data-flow programming paradigm [18]. `Try-Catch` consumes input tokens and produces output tokens. During the process neither `Try` nor `Catch` saves any information so they are called stateless data processing nodes. Figure 4b shows the approximate recovery overhead normalized to rerun-based recovery of non-critical code region. Lower the bar faster is the approximate implementation of the non-critical code region.

ASAR Sampling (ASAR$_s$). Figure 5a shows ASAR using sampling. The `Try` block executes a process node `foo` which consumes four input tokens. The `Catch` block uses a sample of the total input

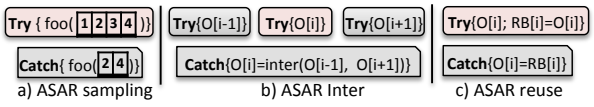


Figure 5: Basic structure of `Try-Catch` blocks using approximation.

tokens, two in the example. Sampling is a widely popular approximation technique for multimedia [30] and databases [2].

We utilize sampling-based approximation technique for two applications, WC and MC. The first two bars in Figure 4b show a 50% reduction in execution time of non-critical code section using approximation via ASAR$_s$. Hence, in the case of an error, the `Catch` block using ASAR$_s$ recovers failed `Try` block in half the time compared to rerun.

ASAR Interpolate (ASAR$_i$). Figure 5b shows ASAR using interpolation. `Try` blocks process a stream of input tokens (i-1, i, and i+1) to produce a stream of output tokens (O[i-1], O[i], and O[i+1]). In case a process node, `Try` {O[i]} block in Figure 5b fails, the execution moves to the `Catch` block. The `Catch` block employs a user-defined interpolate function (inter) which uses previous output token, O[i-1], and triggers the future process node, `Try` {O[i+1]}, to use the next output token i.e. O[i+1]. The simplest interpolate function averages O[i-1] and O[i+1] to approximate O[i]. Interpolation-based approximation is well-suited for DSP algorithms which operate on continuous time-domain signals. Whatmough et al. present hardware-based interpolation using pipeline lookahead to recover from errors for DSP accelerators [47]. We implement and evaluate ASAR$_i$ to accelerate software-based recovery for A2T and FIR, two DSP applications used in this study. The second and third bars in Figure 4b show 40% and 83% reduction in execution time of the non-critical code section using ASAR$_i$ for A2T and FIR, respectively.

***ASAR* Reuse (ASAR$_r$).** Figure 5c shows *ASAR* using reuse. A reuse buffer element is initialized for every new process node and every time a `Try` block processes input i successfully the result from the current iteration is stored in reuse buffer (RB[i]). In case of an error, the output token O[i] is assigned using the corresponding value in the reuse buffer. Reuse-based recovery is applicable to iterative benchmarks and applications with high degree of input locality such as multimedia [3].

We implement reuse-based approximate recovery for four applications, as shown by last four bars in Figure 4b, two iterative applications (KM and PR) and two applications with high input locality (SOR and Histo). K-Means is a clustering algorithm where input i to the `Try` block is a point in space and the output O[i] is a label to one of the available clusters. In every iteration, the K-Means algorithm re-assigns all the points to the closest cluster. In case of an error the `Catch` block can re-assign the point to the cluster from the previous iteration. PageRank implements reuse similar to K-Means. For SOR and Histogram, the reuse buffer holds the output token of the nearest input. K-Means and SOR show 82% and 26% reduction in execution time of non-critical code section using ASAR$_r$. Reduction in execution time for Histogram and PageRank using ASAR$_r$ is negligible or negative. The execution time of the original implementation of non-critical code region for Hsitogram and PageRank use the same number of instructions as reading from the reuse buffer. Hence, replacing rerun which re-executes the original implementation of non-critical code in the `Catch` block with approximate recovery won't result in reduction in recovery time for Histogram and PageRank. For the rest of the paper we will discuss and evaluate the first six applications excluding Histogram and PageRank.

3.1 Hybrid Recovery (ASAR+Rerun)

ASAR provides reduced recovery time relative to rerun and better output quality relative to discard. Using *ASAR* to recover failed

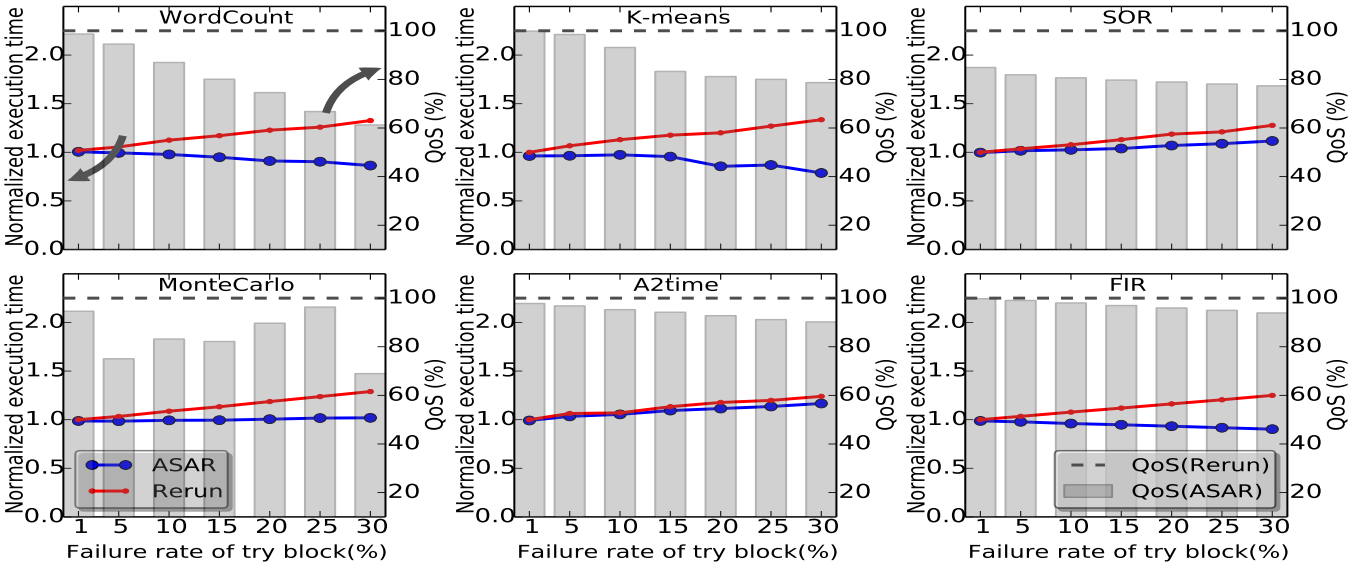


Figure 6: ASAR vs. Rerun: Execution time and output quality of *ASAR* vs. rerun. *ASAR* error recovery alternative such as WC, MC, KM and FIR is much lower with ASAR compared to rerun.

`Try` block provides user with an alternative option in between two extremes: rerun (worst performance, best quality) and discard (best performance, worst quality). *ASAR* results in faster recovery times relative to rerun and improved output quality relative to discard. However, it only provides one more operating point on the performance-quality trade off curve. Additionally, using only approximate recovery may also result in an unacceptable output quality. Hence, we propose a *hybrid recovery* mechanism which uses both rerun and approximate recovery via *ASAR*. The ratio in which *ASAR* and rerun are triggered is called *approximation ratio* and is selected using quality of service model.

Quality of Service Model ($\mathbf{QoS_{mod}}$). The QoS requirements are defined based on the quality of output or timing deadlines [3, 7, 30, 49]. To meet the QoS requirements, a model derives rules for selecting between rerun and *ASAR* block. In other words, the *QoS model* ($\mathbf{QoS_{mod}}$) assists runtime determine *approximation ratio* in order to meet the desired QoS requirement. The following subsections, we describe the details of the QoS model generation and utilization, as shown in Figure 7.

QoS Model Generation. The upper dashed block in Figure 7 encloses the Qos model generation process. We generate $\mathbf{QoS_{mod}}$ by executing an application for a wide range of `Try` block failure rates (fi) and approximation ratio (rj). The failure rate fi represents percentage of `Try` blocks which fail due to unsafe operation and approximation ratio rj determines how many of failed `Try` blocks recover approximately via *ASAR* vs. rerun. We run experiments for each pair of (fi, rj) on training inputs. For each experiment, the output is compared with the golden output. The golden output is the output of error-free execution ($fi = 0$). The final output of the QoS model generator is a discretized table $\mathbf{QoS_{mod}}$ with QoS values for each combination of fi and rj. We use a coarser granularity `Try` block failure rate and approximation ratio to reduce the profiling time one-time $\mathbf{QoS_{mod}}$ generation.

QoS Model Utilization. The runtime takes as input the generated $\mathbf{QoS_{mod}}$ and user specified *QoS threshold* (QoS_{thd}), as shown by the lower dashed block in Figure 7. The runtime failure rate can be estimated using standard hardware monitor detectors [6] and/or hardware error models [11, 42]. For the estimated `Try` block failure rate and user specified QoS threshold we select an approximation ratio to ensure *observed QoS* (QoS_{obs}) greater than QoS threshold (QoS_{thd}). Our results in Section 4 confirm that

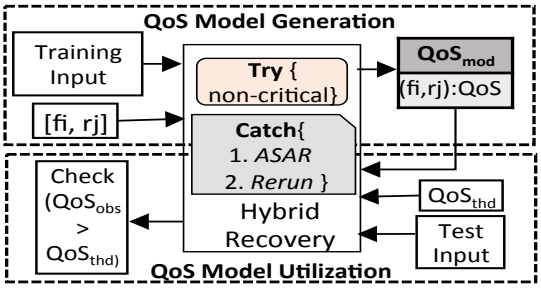


Figure 7: Hybrid recovery (*ASAR* + Rerun) and QoS modeling.

QoS model ensures QoS_{obs} is always greater than QoS_{thd} for test inputs (different from training inputs) and *unseen* finer-granularity failure rates. Thus, using $\mathbf{QoS_{mod}}$ and hybrid recovery enables a user to specify an acceptable output quality threshold and explore various points on performance-quality trade-off curve.

4. Experimental Results

We use six applications from the embedded and DSP domain to evaluate *ASAR* and hybrid recovery. Two applications (WordCount, K-Means) from `Phoenix++` [43], two (A2Time and FIR) from the EEMBC [36] suite, and two (MonteCarlo and SOR) from Scimark2 [34]. Applications are compiled using GCC 4.6.3 and run on a Linux machine with an Intel core i5. We measure the number of cpu cycles spent in critical, non-critical, and recovery code regions. We refactor the code to employ `Try-Catch` blocks. We implement `Catch` block using four recovery mechanisms; rerun, discard, ASAR, and hybrid recovery for all six applications. We simulate random `Try` block failures to simulate the unsafe mode of operation. On average we assume `Try` block fails in the middle of unsafe execution. Hence, for each failed `Try` block we add half the number of failed `Try` block cycles plus recovery overhead cycles. In the following subsections, we compare the performance and output quality (QoS metric) for the four recovery mechanisms.

4.1 Application-Specific Approximate Recovery (ASAR)

Figure 6 shows the performance and output quality for six applications using *ASAR*. The x-axis represents the rate of `Try` block failure, the left y-axis shows the execution time, normalized to the run-

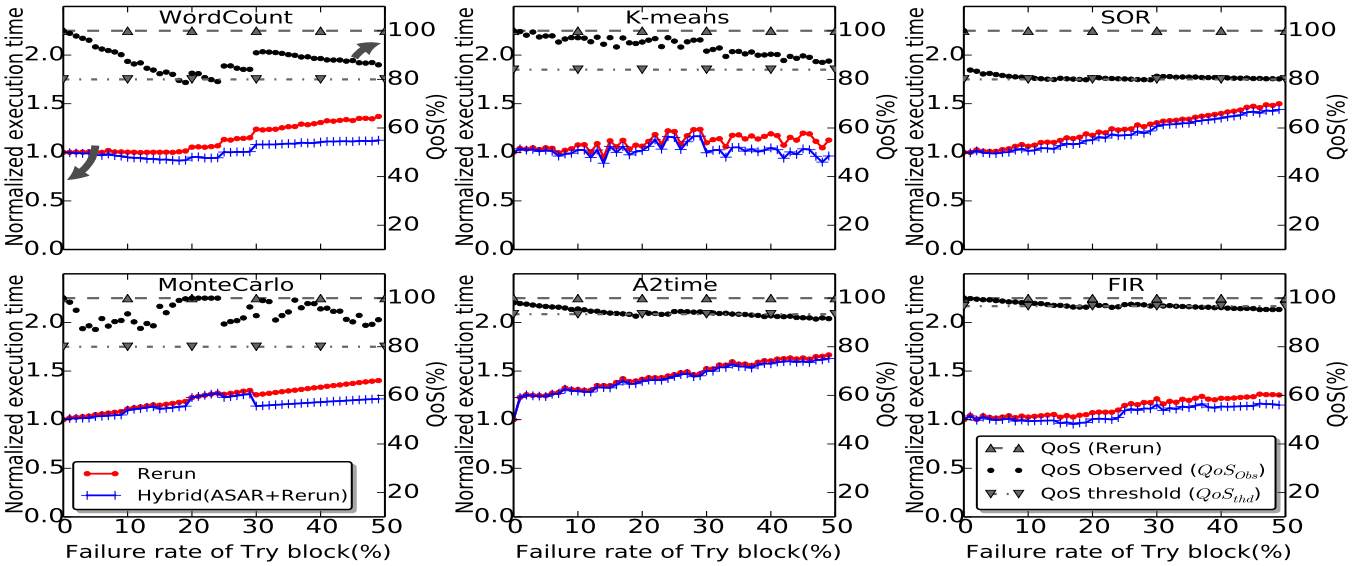


Figure 8: Hybrid vs. Rerun recovery: Hybrid recovery uses ASAR and rerun-based recovery based on a approximation ratio to explore the entire performance-quality trade-off curve. For all six applications hybrid recovery maintains observed QoS above a user specified threshold and runs faster than pure rerun-based recovery.

	ASAR				Hybrid (ASAR + Rerun)			
	QoS Loss (%)		Performance Improvement (%)		QoS Loss (%)		Performance Improvement (%)	
Benchmark	Avg	Max	Avg	Max	Avg	Max	Avg	Max
WordCount(WC)	19.6	38.8	24.8	53.6	12.2	21.3	11.6	21.7
MonteCarlo(MC)	15.7	31.1	13.7	26.5	5.7	12.4	6.3	15.5
SOR	19.8	22.7	7.5	14.4	18.8	20.1	3.9	4.1
K-means(KM)	12.2	21.3	29.9	69.5	6.2	13.0	8.1	16.9
A2time(A2T)	5.9	9.8	3.8	6.2	5.9	8.6	1.5	2.2
FIR	3.0	6.1	5.7	18.8	2.9	4.7	5.8	9.0

Table 1: Average and maximum QoS loss and performance improvement for *ASAR* and Hybrid recovery.

time of error-free execution, and the right y-axis shows the output quality using the QoS metric. The lines on each subgraph show the normalized execution time measurements. Approximate recovery via *ASAR* performs better than rerun-based recovery for all six applications. The average reduction in recovery overhead with *ASAR* relative to rerun is highest for WordCount, K-Means, MonteCarlo, and FIR. These applications spend substantial amount of their time in non-critical code region so the impact of *ASAR* is more pronounced. A2Time and SOR show only 3.8% and 7.5% reduction in recovery time relative to rerun. The gap between ASAR and rerun performance widens with the increase in the rate of `Try` block failure because at higher `Try` block failure rate faster approximation is employed more frequently. The reduction in performance overhead ranges from 3.8% to 29.9%. For all six applications the QoS loss increases monotonically with the increase in `Try` block failure rate except for applications which are inherently random. MonteCarlo, an application that employs a randomized algorithm, computes the value of π by randomly choosing points in a two dimensional plane. The inherent random nature of the application attributes to the non-monotonicity in the QoS loss. We observe a maximum QoS loss of 38.8%. The average and maximum QoS loss and performance improvement relative to the rerun method for each application using *ASAR* are also listed in Table 1.

Figure 6 is also used to generate $\mathbf{QoS_{mod}}$ for hybrid recovery. We generate $\mathbf{QoS_{mod}}$ for seven coarser-granularity `Try` block failure rates $fi \in [1\%, 5\%, 10\%15\%, 20\%, 25\%, 30\%]$. This range of failure rate is representative of variability-induced hardware failures [10, 35].

4.2 Hybrid Recovery using $\mathbf{QoS_{mod}}$ (ASAR + Rerun)

ASAR significantly reduces error recovery overhead (6.2%–69.5% at maximum) compared to rerun-based error recovery. However, it suffers from variable levels of QoS loss. *ASAR*-only recovery cannot guarantee a bound on the final application QoS. In order to address this issue and explore performance-quality trade-off curve at a finer-granularity, we propose a *hybrid recovery* (*ASAR* + Rerun) mechanism. In this subsection, we evaluate the effectiveness of hybrid recovery using $\mathbf{QoS_{mod}}$.

The execution time and QoS quality with hybrid recovery for all six applications are shown in Figure 8. All axes have the same units as Figure 6. The first horizontal line at the top of each subgraph represents the perfect QoS achieved using rerun-based recovery or error-free execution. The second horizontal shows the QoS threshold (QoS_{thd}) specified by the user. We test our $\mathbf{QoS_{mod}}$ for a much finer-granularity of *unseen* failure rates (1%--50%). The wider failure rate explores more aggressive near-threshold operations. We use test inputs for the results shown Figure 8 which are different from training inputs used for Figure 6.

The hybrid recovery mechanism attempts to conservatively match a characterized point in $\mathbf{QoS_{mod}}$ and selects an approximation ratio. The results show that hybrid recovery is able to maintain observed QoS (QoS_{obs}) greater than the threshold (QoS_{thd}) for all applications. The performance improvement with hybrid recovery depends on the following factors: 1) ratio of critical vs. non-critical code, 2) aggressiveness of non-critical code execution or the failure rate of `Try` block, and 3) the QoS threshold QoS_{thd}. QoS_{thd} of 100% will not result in any performance improvement because the hybrid recovery will select approximation ratio of zero to maintain the perfect output quality. In order to demonstrate the effectiveness of hybrid recovery we select QoS_{thd} in the range of 80% to 98% for different applications. However, the user can specify any QoS_{thd} to obtain a specific point on performance-quality trade-off curve. For a failure rate of 1%–50%, we observe a maximum QoS loss of 21.3% with a maximum error recovery speedup of 21.7% over six applications, as shown in Figure 8 and Table 1.

4.3 Discard Recovery

In this subsection, we evaluate the discard mechanism to recover from failed `Try` blocks. The execution time and the QoS for six applications using the discard mechanism are shown in Figure 9. The two horizontal lines at the top of each subgraph show the perfect QoS using rerun-based recovery and user-specified QoS

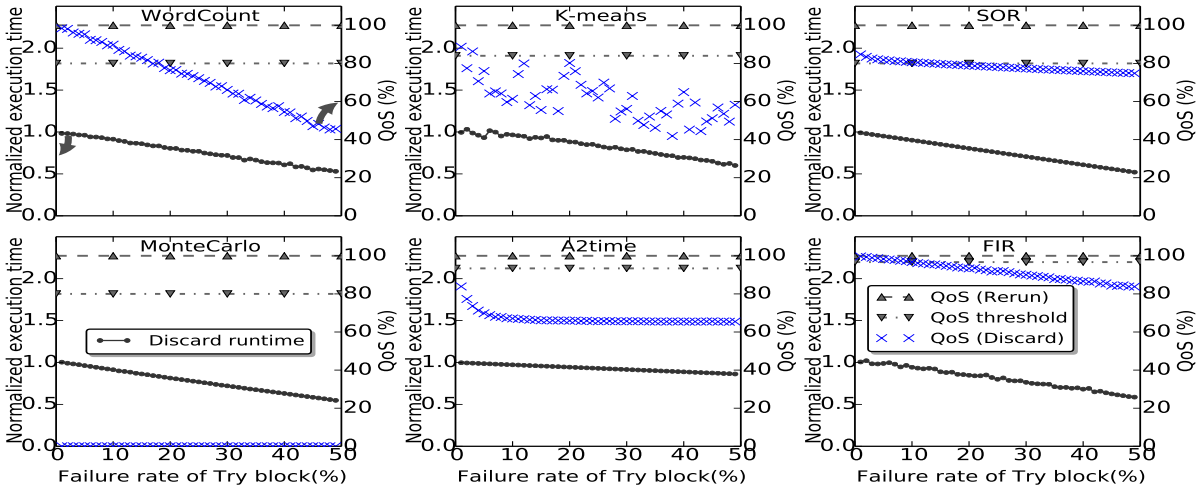


Figure 9: Discard recovery mechanism performs the fastest among rerun, ASAR, and hybrid recovery, however, for all applications discard fails to keep QoS (output quality) above the user specified threshold.

threshold (QoS_{thd}), same as in Figure 8. The discard mechanism achieves the highest performance improvement with a range of 30%–70% faster execution time relative to *ASAR*. However, the observed QoS (Discard) of output falls below the QoS threshold (QoS_{thd}). The `Try` block failure rate at which the QoS drops below QoS_{thd} is called cutoff failure rate (f_c). Some applications exhibit f_c of as low as 1%, for example MC and A2T. Applications such as MonteCarlo doesn't support discard-based recovery and dropping a sub-computation deters the algorithm to compute the final answer resulting in 100% QoS loss. Overall, the discard mechanism suffers from QoS loss ranging from 16% to 100%. These drawbacks limit the usage of the discard scheme if the user requires strict guarantees on output quality.

5. Related Work

Approximate-computing domain offers an opportunity to tradeoff output quality for performance and/or energy [9, 15, 20, 32]. Rinard et al. propose program transformations for approximate-computing trading output quality for increased performance under error-free environment [30, 31, 50]. Relay [8] is a programming language that enables developers to provide bounds on probability of error given an output quality executing under unsafe modes. Green [4] proposes an online monitoring system to trade off quality of service for reducing in energy consumption. Kulkarni et al. propose under designed multiplier architecture by approximate circuit implementation of multiplier blocks to gain speed in lieu of quality for image processing applications [25].

EnerJ [41] is a programming language supporting *disciplined* approximate computation. It lets programmers mark critical and non-critical code sections at an instruction granularity. Truffle [14], a dual-voltage micro-architecture design, supports mapping of approximate EnerJ programs through ISA extensions. Truffle applies a high voltage (safe mode) for critical operations and a low voltage (unsafe mode) for non-critical operations. Truffles demonstrate up to 43% energy saving by using dual-voltage operation which incurs no overhead for transition between safe/unsafe modes for statically partitioned code in to critical and non-critical regions. ERSA isolates the execution of iterative algorthims in to critical and non-critical code at a coarser granularity by separating control-intensive tasks from data-intensive tasks [27]. While ERSA employs software checks on sub-computations to ensure bounds on execution time and final output, Truffle relies on the programming language support to provide safety guarantees and doesn't employ recovery for the non-critical executing under unsafe mode. Relyzer [19] is a resiliency analyzer which can help prune fault sites up to five order of magnitude and enable a software developer to locate sites vulnerable to SDCs.

Relax proposes a compiler/architecture system to expose hardware errors during unsafe non-critical code execution and allow software-based recovery [11]. Relax employs software recovery using rerun (worst performance, best quality) and discard (best performance, worst quality). *ASAR* is an extension for a Relax-like system which provides a user with an approximate recovery (good performance, good quality) alternative in between rerun and discard. We further propose a hybrid recovery mechanism which allows exploiting the performance-quality trade-off curve at much finer granularity.

6. Conclusion

We propose *ASAR*, application-specific approximate recovery, scheme that enables re-factoring a program in critical and non-critical code. The critical code is sought to perform exactly as conventional software whereas the non-critical code enables the programmer to specify application-specific flexibility. Together, these can be used in an approximate computing system model. This lowers the cost of software-based error recovery relative to rerun by using an approximate alternative. To guarantee the output acceptability and explore the performance-quality curve at a finer granularity we propose a hybrid recovery mechanism. Hybrid recovery uses a well characterized QoS model and a mixture of available software-based recovery schemes. We implement an instance of hybrid recovery using *ASAR* and rerun recovery mechanism. We also characterize a QoS model using training inputs and show that the proposed hybrid recovery can operate at any intermediate point on the performance-quality trade-off curve for test inputs. Our results demonstrate that hybrid recover provides 1.5%–11.6% faster execution time relative to the rerun mechanism and guarantees an output quality greater than the user-specified threshold. We also show that the discard mechanism reaches on average 30%–70% faster execution relative to *ASAR*, but could suffer from an unacceptable QoS degradation.

Acknowledgements

This work was supported by the NSF Expedition in Computing grant CCF-1029783.

References

[1] ITRS [online]. available: http://public.itrs.net.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[3] C. Álvarez, J. Corbal, and M. Valero. Dynamic tolerance region computing for multimedia. *IEEE Trans. Computers*, 61(5):650–665, 2012.

[4] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[5] V. Balasubramanian and P. Banerjee. Craft: Compiler-assisted algorithm-based fault tolerance in distributed memory multiprocessors. In *ICPP (1)*, pages 501–504, 1991.

[6] K. A. Bowman, J. W. Tschanz, S.-L. Lu, P. A. Aseron, M. M. Khellah, A. Raychowdhury, B. M. Geuskens, C. Tokunaga, C. Wilkerson, T. Karnik, and V. K. De. A 45 nm resilient microprocessor core for dynamic variation tolerance. *J. Solid-State Circuits*, 46(1):194–208, 2011.

[7] M. A. Breuer. Multi-media applications and imprecise computation. In *Proc. IEEE Euromicro Conference on Digital System Design (DSD)*, pages 2–7, 2005.

[8] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, New York, NY, USA, 2013. ACM.

[9] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006*, pages 1110–1115, 2006.

[10] H. Cho, L. Leem, and S. Mitra. Ersa: Error resilient system architecture for probabilistic applications. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(4):546–558, 2012.

[11] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture*, 2010.

[12] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 475–486, New York, NY, USA, 2012. ACM.

[13] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 7–, Washington, DC, USA, 2003. IEEE Computer Society.

[14] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. *SIGARCH Comput. Archit. News*, 40(1):301–312, Mar. 2012.

[15] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 449–460, Dec 2012.

[16] S. Ghosh and K. Roy. Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era. *Proceedings of the IEEE*, 98(10):1718–1751, 2010.

[17] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. M. Tullsen, and R. Gupta. Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In *Proceedings of the 54th Annual Design Automation Conference*, DAC '17, 2017.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, sep 1991.

[19] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. *SIGPLAN Not.*, 47(4):123–134, Mar. 2012.

[20] R. Hegde and N. R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings of the 1999 international symposium on Low power electronics and design*, ISLPED '99, pages 30–35, New York, NY, USA, 1999. ACM.

[21] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Hartig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. Tahoori, J. Teich, N. When, and H. Wunderlich. Design and architectures for dependable embedded systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, pages 69–78, Oct 2011.

[22] J. Henkel et al. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. DAC '13.

[23] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *International Workshop on Electronic Design, Test & Applications*, pages 440–447, jan 2006.

[24] W. Kim, M. S. Gupta, G.-Y. Wei, and D. M. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *HPCA*, pages 123–134. IEEE Computer Society, 2008.

[25] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *Proceedings of the 2011 24th International Conference on VLSI Design*, VLSID '11, pages 346–351, Washington, DC, USA, 2011. IEEE Computer Society.

[26] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[27] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. Ersa: error resilient system architecture for probabilistic applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1560–1565, 3001 Leuven, Belgium, Belgium, 2010. European Design and Au-

tomation Association.

[28] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM J. Res. Dev.*, 6(2):200–209, Apr. 1962.

[29] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[30] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[31] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[32] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *DATE*, pages 950–955, 2011.

[33] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, Washington, DC, USA, 2003. IEEE Computer Society.

[34] B. Pozo, R. Miller. Scimark2. http://math.nist.gov/scimark2.

[35] A. Rahimi, L. Benini, and R. K. Gupta. Analysis of instruction-level vulnerability to dynamic voltage and temperature variations. In W. Rosenstiel and L. Thiele, editors, *DATE*, pages 1102–1105. IEEE, 2012.

[36] M. C. Rajiv Adhikary. EEMBC. http://www.eembc.org/.

[37] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM.

[38] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[39] S. Rehman, M. Shafique, and J. Henkel. Instruction scheduling for reliability-aware compilation. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1292–1300, New York, NY, USA, 2012. ACM.

[40] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[41] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[42] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. In *in IEEE Transactions on Semiconductor Manufacturing*, 2008.

[43] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[44] H.-W. Tseng and D. M. Tullsen. CDTT: Compiler-generated data-triggered threads. In *HPCA*, pages 650–661. IEEE Computer Society, 2014.

[45] A. Venkat and D. M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press.

[46] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 73–84, Piscataway, NJ, USA, 2014. IEEE Press.

[47] P. Whatmough, S. Das, and D. Bull. Hybrid circuit and algorithmic timing error correction for low-power robust dsp accelerators. In *Solid-State Circuits Conference (A-SSCC), 2013 IEEE Asian*, pages 29–32, Nov 2013.

[48] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 203–209, New York, NY, USA, 2005. ACM.

[49] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *SIGOPS Oper. Syst. Rev.*, 50(2):33–47, Mar. 2016.

[50] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 441–454, New York, NY, USA, 2012. ACM.