ASAR: Application-Specific Approximate Recovery to Mitigate Hardware Variability

Manish Gupta, Abbas Rahimi, Daniel Lowell, John Kalamatianos, Dean Tullsen, Rajesh Gupta

1UC San Diego {manishg, abbas, tullsen, rgupta}@cs.ucsd.edu 2AMD Research {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 Application-Specific Approximate Recovery (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 [15, 21]. More aggressive process technology nodes in the coming years will see increased variability, thus increasing frequency 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 [23], 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 [8, 17, 40, 46], checkpointing & re-execution [23], and compiler-driven vulnerability reduction [39, 83]. 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 cost.

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-
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 [25][26] (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 Application-Specific Approximate Recovery (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 by 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 Kruifj 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 module at a faster execution speed and higher probability of encountering hardware errors. If an error occurs during the Try block execution, hard-

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

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

ware 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. Kruifj et al. evaluate two implementations for software error recovery: rerun and discard [11].

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, subcomputations by an erroneous Try block are dropped. The Catch block is programmed to account for the number of dropped subcomputations 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 Application-Specific Approximate Recovery (ASAR) to provide an approximate recovery alternative. ASAR provides a mechanism that lies 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][41]. The critical part mostly consists 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-
Excuse me, but it seems there's an error in the page dimensions. The page dimensions are not properly formatted. Could you please provide the correct page dimensions?
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 (QoSmod). The QoS requirements are defined based on the quality of output or timing deadlines \[3\] \[30\] \[49\]. To meet the QoS requirements, a model derives rules for selecting between rerun and ASAR block. In other words, the QoS model (QoSmod) 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 QoSmod 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 QoSmod 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 QoSmod generation.

QoS Model Utilization. The runtime takes as input the generated QoSmod and user specified QoS threshold \(QoS\text{thd}\), as shown by the lower dashed block in Figure 7. The runtime failure rate can be estimated using standard hardware monitor detectors \[4\] and/or hardware error models \[11\] \[32\]. For the estimated try block failure rate and user specified QoS threshold we select an approximation ratio to ensure observed QoS \(QoS\text{obs}\) greater than QoS threshold \(QoS\text{thd}\). Our results in Section 4 confirm that

![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.](Image)

![Figure 7: Hybrid recovery (ASAR + Rerun) and QoS modeling.](Image)

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 Smark2 \[34\]. 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 9 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.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ASAR Loss (%)</th>
<th>ASAR Improvement (%)</th>
<th>Hybrid (ASAR+Rerun) Loss (%)</th>
<th>Hybrid (ASAR+Rerun) Improvement (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>19.6 25.9</td>
<td>24.0 39.6</td>
<td>12.2 21.1</td>
<td>31.8 21.4</td>
</tr>
<tr>
<td>MonteCarlo</td>
<td>15.7 31.1</td>
<td>13.7 26.5</td>
<td>5.7 12.4</td>
<td>6.3 13.5</td>
</tr>
<tr>
<td>SOR</td>
<td>19.8 21.5</td>
<td>7.5 14.4</td>
<td>19.8 20.1</td>
<td>3.9 4.1</td>
</tr>
<tr>
<td>K-means</td>
<td>12.2 21.3</td>
<td>29.9 69.5</td>
<td>6.2 13.0</td>
<td>8.1 16.9</td>
</tr>
<tr>
<td>A2Time</td>
<td>5.9 9.8</td>
<td>7.9 12.6</td>
<td>5.9 8.6</td>
<td>1.5 2.2</td>
</tr>
<tr>
<td>FIR</td>
<td>3.0 6.1</td>
<td>5.7 18.8</td>
<td>2.9 4.7</td>
<td>3.8 9.0</td>
</tr>
</tbody>
</table>

The hybrid recovery mechanism attempts to conservatively match a characterized point in QoSmod and selects an approximation ratio. The results show that hybrid recovery is able to maintain observed QoS (QoSmod) greater than the threshold (QoSthd) 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 QoSthd. QoSthd 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 QoSthd in the range of 80% to 98% for different applications. However, the user can specify any QoSthd 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.
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 \[20\]\[31\]\[50\]. 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 \[7\] 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 \[23\].

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 algorithms 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-factorising 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


tomation Association.


