A Survey of Techniques for Modeling and Improving Reliability of Computing Systems

Sparsh Mittal, *Member, IEEE* and Jeffrey S. Vetter, *Senior Member, IEEE*

Abstract—Recent trends of aggressive technology scaling have greatly exacerbated the occurrences and impact of faults in computing systems. This has made 'reliability' a first-order design constraint. To address the challenges of reliability, several techniques have been proposed. This paper provides a survey of architectural techniques for improving resilience of computing systems. We especially focus on techniques proposed for microarchitectural components, such as processor registers, functional units, cache and main memory etc. In addition, we discuss techniques proposed for non-volatile memory, GPUs and 3D-stacked processors. To underscore the similarities and differences of the techniques, we classify them based on their key characteristics. We also review the metrics proposed to quantify vulnerability of processor structures. We believe that this survey will help researchers, system-architects and processor designers in gaining insights into the techniques for improving reliability of computing systems.

Index Terms—Review, classification, reliability, resilience, fault-tolerance, vulnerability, architectural vulnerability factor, soft/transient error, architectural techniques

1 Introduction

Recent trends in processor design have made them increasingly vulnerable to faults. Ongoing scaling of transistors and operating voltages have led to a dramatic rise in susceptibility of processors to faults. For example, the soft-error failure rate at 16 nm is expected to be more than 100 times that at 180 nm [1]. Further, as computing systems ranging from mobile devices to large data centers and supercomputers become increasingly employed, a system malfunction is likely to have more severe financial and social consequence than ever before. For example, it has been reported that in one incident, a single soft error crashed an interleaved system farm, while in another incident, a single soft error disrupted the operation of a billion-dollar automotive factory every month [2]. Several other incidents have also been reported [3], [4]. Even short duration disruptions can have major impact in large-scale enterprises, for example, merely one hour of downtime can cost more than $6,450,000 and $2,600,000, in brokerage operations and credit card authorization, respectively [5]. Thus, ensuring the reliability of computing systems has become the primary requirement to achieve exascale performance [6]. These trends and incidents clearly indicate that partial retrofitting of existing techniques to address reliability is likely to be insufficient. Instead, the designers need to consider reliability as the first-order design constraint.

- *S. Mittal is with the Future Technologies Group, Oak Ridge National Laboratory, Oak Ridge, TN 37830. E-mail: mittals@ornl.gov.*
- *J.S. Vetter is with the Future Technologies Group, Oak Ridge National Laboratory, Oak Ridge, TN 37830, and with the Georgia Institute of Technology, Atlanta, GA 30332. E-mail: vetter@ornl.gov.*

Manuscript received 17 Nov. 2014; revised 16 Apr. 2015; accepted 20 Apr. 2015. Date of publication 23 Apr. 2015; date of current version 16 Mar. 2016. Recommended for acceptance by Y. Solihin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2426179

To address this challenge, design of reliable processors has received significant amount of interest in recent years. In this paper, we present a survey of architectural techniques for modeling and improving reliability of computing systems. We especially focus on techniques for improving reliability of microarchitectural components, such as registers, functional units, cache and main memory etc. We discuss techniques proposed for both CPU and GPU, and those proposed in context of non-volatile memories and 3D integration technologies. Further, we discuss various metrics proposed for modeling and quantifying the vulnerability (or reliability) of computing systems. We classify the techniques based on several key characteristics to highlight their similarities and differences. We discuss techniques evaluated using both real machines and architectural simulators.

To keep a balance between the breadth (coverage) and depth (detail), we limit the scope of the paper in the following manner. We discuss (micro)architecture level techniques and not circuit/device-level or algorithm/application-level techniques for characterizing and improving reliability. Out of hard and soft errors, we mainly focus on techniques dealing with soft errors, since their mitigation at architectural level presents interesting challenges and opportunities (refer Section 3 for a background on the errors). We do not include works which intentionally compromise reliability for achieving performance/energy gains, e.g. near-threshold voltage operation. Of different NVMs, viz. Flash memory, phase change RAM (PCM), spin transfer torque RAM (STT-RAM), resistive RAM (ReRAM), we only discuss techniques proposed for the last three NVMs, since given the huge amount of research work done to improve reliability of Flash memory, doing full justice to it in a paper of this length is not possible.

The rest of the paper is organized as follows. Section 2 motivates the need of techniques for addressing reliability challenges. Section 3 presents a brief background on errors and a classification of techniques on several dimensions.

Sections 4 and 5 discuss the techniques for modeling and improving reliability. Finally, Section 6 discusses conclusion and future challenges.

2 Need of Designing for Reliability

The design of processors has traditionally been optimized for performance. However, several factors and recent trends present compelling reasons for significantly improving the reliability of computing systems.

2.1 Trends of Device Scaling

With shrinking transistor sizes, the processor circuits and components are becoming increasingly susceptible to soft-errors. With scaling of operating voltage, the critical charge required to flip a stored value has been decreasing [7]. Further, in atmospheric radiation, particles of lower energy occur far more frequently than those of higher energy and hence, with voltage scaling more particles can cause soft errors [4]. Since a single energetic particle can lead to burst of consecutive errors, the likelihood of multi-bit errors is also on rise [8], [9].

With increasing system core-count, the size of last level cache (LLC) in modern processors has also been increasing [10]. As an example, Intel's 32 nm Itanium processor had a 32 MB SRAM LLC [11] while the 22 nm Ivytown processor has a 37.5 MB SRAM LLC [12]. Similarly, IBM's 32 nm POWER7+ processor had an 80 MB eDRAM (embedded DRAM) LLC [13], while the 22 nm POWER8 processor has a 96 MB eDRAM LLC [14]. Similarly, while NVIDIA's GT200 architecture GPU did not feature an L2 cache, the Fermi GPU has 768 KB LLC, the Kepler GPU has 1,536 KB LLC and the Maxwell GPU has 2,048 KB LLC [15]. Increasing the size of a cache can, however, lead to a superlinear increase in its soft error rate [16], [17], [18]. Further, with increasing cache size, the overhead of protection techniques such as scrubbing or flushing will grow since an increasingly large number of blocks need to be accessed within a fixed time to maintain reasonable error rates [19], [20]. The similar trends are also true for other structures such as main memory [21].

2.2 Vulnerability of Emerging Memory Technologies

Non-volatile memories, such as STT-RAM consume near-zero static power and are generally considered immune to radiation-induced soft-errors [22], [23]. However, recent research has shown that retention failure can lead to stochastic bit-flips in STT-RAM and ReRAM [24], a phenomenon which resembles soft-errors in charge-based memories (e.g. SRAM). Similarly, the resistance of a PCM cell increases with time. In multi-level cell (MLC) PCM cells, this drift can lead to errors, since over time, a memory cell can start representing different value than what was originally stored in it [21]. Due to this, a four-level PCM may show 10^6 times higher soft errors than the DRAM [25]. Further, the peripheral circuits of NVMs still use CMOS which are susceptible to soft errors. For these reasons, a drop-in replacement of conventional charge-based memories with NVMs is unlikely to solve the reliability issues.

2.3 High Demands of Reliability in Several Applications

Several mission-critical applications/domains such as medical equipments demand very high reliability. Similarly, several equipments are operated at high altitudes or under harsh environments where the error-rates are high. For example, compared to sea level, the rate of neutron flux is 3.5 times higher at 1.5 km [3] and 300 times higher at 10–12 km [4] which is the typical altitude where airplanes fly. For ensuring reliability under such scenarios, the device vendors need to perform rigorous and costly quality assurance tests which increase the cost of the system.

2.4 Performance/Energy Consequences of Errors

To handle errors, chip-designers typically use complex error-correction and detection circuits. However, these circuits cause additional slowdowns on each access and their overheads grow rapidly as their correction capability is increased. Due to the latency-optimized design of L1 cache, use of complex error-correcting code (ECC) in L1 cache imposes prohibitive overheads. Further, in the case of an actual error, modern processors raise an interrupt that must be serviced by the system firmware which incurs a latency three to four orders of magnitude higher than the cache/memory access latency and thus, leads to unpredictable slowdowns. When errors in a page exceed a certain threshold, the page needs to be retired to avoid error-correction overheads. This, however, degrades the capacity of the memory. Uncorrectable errors can lead to machine failure and undetected errors can cause even more severe harms.

3 A Background on and Classification of Techniques for Reliability

We now briefly discuss about the errors and some ideas used in techniques for improving reliability. We refer the reader to previous work for a detailed background and taxonomy of related terms [6], [7], [26].

There are two main types of errors, namely hard (or recurring) errors and soft (or transient) errors. Hard errors are permanent faults in the system and they occur due to thermal stress, wearout, and process variation. Soft errors are logical faults in circuit's operation and they occur at random due to charged particle emissions from atmosphere or resistance drift in MLC NVMs etc. [2], [21].

To make an estimate of actual system errors, researchers utilize an important observation that several soft errors occurring at "lower" level (e.g. circuit level) are masked at "higher" level (e.g. architectural level) and thus, they do not result in a visible system error. Based on this, several metrics have been proposed which measure the probability that a fault within a structure will result in a visible system error (Section 4.2).

In a write-back cache, clean and dirty data have different vulnerability to soft errors. Since dirty data will be written back to lower caches, a fault in it will propagate to memory hierarchy. Conversely, a fault in a clean block can be corrected by consulting other copies in the memory hierarchy, however, the dirty block stores the only updated value available in the memory hierarchy and hence, a fault in it cannot be corrected. For this reason, several techniques provide extra or exclusive

TABLE 1
A Classification of Research Works

Classification	References
Processor component	
Cache	[7], [8], [16], [17], [19], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57], [58], [59]
Register file	[46], [48], [54], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79]
Main memory	[21], [25], [43], [46], [57], [59], [80], [81], [82], [83], [84], [85], [86]
Other components	[3], [58], [66], [69], [73], [74], [75], [76], [78], [79], [87], [88], [89], [90], [91], [92], [93]
Key approach/feature	
Reliability metrics proposed	[3], [36], [37], [41], [46], [48], [53], [54], [61], [62], [66], [79], [80], [91], [94], [95], [96], [97]
Computing reliability metrics	[9], [35], [66], [72], [73], [75], [88], [93], [95], [98]
Use of compiler	[38], [43], [49], [60], [61], [62], [65], [77], [79], [89], [97], [99]
Use of redundancy/duplication	[8], [27], [28], [44], [50], [51], [55], [58], [59], [64], [68], [71], [73], [76], [77], [90], [96], [99], [100], [101], [102], [103], [104], [105]
Use of error correcting code	[7], [8], [21], [27], [29], [40], [45], [55], [56], [59], [62], [63], [71], [81], [82], [83], [84], [86]
Use of cache scrubbing or flushing (write-back)	[7], [18], [19], [29], [30], [37], [49], [53], [55], [57]
Main memory scrubbing	[21], [57], [83], [86]
Exploiting narrow data values	[52], [63], [68], [74], [76], [78]
Backup in or from protected/ reliable memory	[33], [60], [90]
Use or context of NVMs	[21], [23], [25], [82], [90], [106]
Fault injection experiments	[7], [9], [16], [17], [27], [43], [45], [46], [47], [63], [66], [67], [68], [70], [71], [72], [75], [81], [87], [96], [97], [99], [104], [107], [108]
Multi-bit faults	[7], [9], [19], [21], [30], [32], [34], [42], [43], [55], [57], [81], [83], [86], [104]
Selectively protecting more vulnerable components/data	[7], [8], [27], [29], [31], [45], [55], [56], [58], [62], [64], [70], [77], [92], [104]
Trade-off with energy	[7], [16], [25], [49], [51], [71], [78], [91], [106], [109]
Trade-off with QoS	[45], [50], [70], [77]
Effect of cache size on reliability	[16], [18], [19], [35], [39], [44], [45], [49]
Effect of 3D design on reliability	[55], [56], [74], [103], [106]
Techniques for GPU	[59], [66], [69], [70], [87], [101], [110]
Evaluation platform	
Simulator	[3], [8], [9], [16], [17], [18], [21], [25], [26], [27], [28], [29], [30], [33], [35], [37], [38], [39], [40], [41], [42], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [58], [60], [61], [62], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [76], [77], [79], [80], [82], [84], [87], [88], [89], [90], [91], [92], [93], [95], [96], [97], [99], [100], [101], [102], [103], [104], [105], [107], [109]
Real hardware	[17], [57], [81], [83], [85], [86], [87], [110]

protection to dirty data in the cache (e.g. [7], [18], [27], [28], [29], [30]). Since an instruction cache is mainly read-only, while a data cache sees both reads and writes, error detection/correction is more critical for data caches.

Table 1 presents a classification of the research works based on several dimensions, such as the processor component they protect, their essential approach and objective etc. Several works use redundancy for detection and correction of errors (refer Table 1). For example, since storage structures have regular patterns, they are easily protected by parity and error correction codes. By comparison, combinational logic structures have irregular patterns and hence, they are protected using redundant execution, e.g., by executing an application on multiple disjoint structures or multiple times on the same structure. Some researchers study cache scrubbing, where the cache blocks are periodically examined for errors using ECC and an inline correction of errors is performed to prevent the number of errors from becoming larger than the correction capability of ECC (refer Table 1).

In Table 1, we also classify the works based on whether they are evaluated on a simulator or on a real machine. While the simulators provide the flexibility to experiment with certain components, techniques or fault-injection campaigns within reasonable amount of time and in a repeatable manner, their modeling inaccuracies and simplifying assumptions may lead to misleading conclusions. On the other hand, while the field studies on real machines provide valuable data and insights, they also require experiments over a large number of machines over a large period of time to draw statistically significant conclusions. Field study of fault-injection experiments is particularly challenging, since it requires controlled experiments and availability of neutron (or proton) beam sources [17] which may incur significant financial overheads and may not be commonly

available. For these reasons, a large number of studies have been conducted using simulators.

4 Techniques for Modeling Reliability

4.1 Reliability Metrics

Several researchers propose metrics which enable a systematic and quantitative evaluation of vulnerability (or reliability) and thus, allow the designers to avoid both under-protection and over-protection. We now discuss a few of these works.

Mukherjee et al. [3] propose "architectural vulnerability factor" (AVF) to evaluate the susceptibility to soft errors of a processor structure. AVF is estimated by tracking the bits in the structure which are needed for architecturally correct execution (ACE), which implies that an error in ACE bits will lead to an error in the final application output. Unless proven otherwise, all bits are considered as ACE and using this, an upper bound on AVF can be determined. The bits which are unnecessary for ACE are referred to as unACE bits. Using this, the error rate of a structure is computed as the product of its raw error rate and the AVF.

The AVF metric measures full-system vulnerability. Several researchers have defined metrics which quantify vulnerability of individual layers in the system stack. Sridharan and Kaeli [79] propose program vulnerability factor (PVF) which is a microarchitecture-independent method to quantify fault-masking inherent to a program. Sridharan and Kaeli [95] also propose hardware vulnerability factor (HVF) which quantifies the hardware portion of AVF, independent of program-level masking. The advantage of defining PVF and HVF is that they provide useful insights into the application behavior and the hardware design, respectively and using them, AVF can be computed as the product of HVF and PVF. Further, runtime monitoring of HVF enables runtime estimation of AVF by combining this with compile-time estimate of PVF. Sridharan and Kaeli [95] further introduce the notion of "system vulnerability stack", which enables calculation of a vulnerability factor for each layer of the system stack to finally compute AVF from its individual components.

Yan and Zhang [62] propose register vulnerability factor (RVF) which measures the probability that soft errors in registers can impact the system reliability. RVF is defined as the average time the register values are susceptible to errors. They also propose two compiler-guided techniques which aim to improve the register file reliability by lowering the RVF value. The first technique delays the write operations as late as possible and schedules the read operations as early as possible. To avoid performance loss, the compiler analyzes the dependence graph to exploit the scheduling slack. The second technique works for the case where only a few registers are protected by ECC due to the overhead of ECC. This technique uses profiling to direct the register allocation such that the registers with ECC always have the highest RVF values which improves the reliability of the register file.

Borodin and Juurlink [96] propose instruction vulnerability factor (IVF) which shows the probability that an error in instruction affects the final program output. IVF is computed by offline profiling using simulation or fault-injection experiments. Using the AVF value, at runtime, each instruction can be protected at the level required by its AVF value.

Zhang [37] defines the cache vulnerability factor (CVF) as the probability that a fault in the cache can propagate to other components of the processor. This is computed based on the average time the cache blocks are susceptible to soft errors. Using CVF as a quantitative metric, he shows that the vulnerability of a write-through L1 cache is much smaller than that of a write-back L1 cache. Since a write-back cache provides higher performance than a write-through cache, Zhang [37] also shows that using early write-back of dirty data, the reliability of a write-back cache can be improved at the cost of a small loss in its performance.

Asadi et al. [18] present a method to compute the reliability of cache memory. They define the notion of 'critical word' (CW), which is a cache word that is guaranteed to propagate to other locations in memory hierarchy or CPU. The critical time associated with a CW is the time period in which the context of that CW is important. Using this, the vulnerability factor of cache is defined as the average critical time of all the critical words. Using this metric, they observe that the reliability of cache varies widely with applications. They show that increasing the cache size can cause superlinear increase in vulnerability and thus, by computing the vulnerability factor for a given cache size, a designer can balance the performance and reliability of a processor.

Soundararajan et al. [91] study the impact of several DVFS (dynamic voltage/frequency scaling) algorithms on the architectural vulnerability of GALS (globally asynchronous, locally synchronous) architectures. They note that DVFS affects the utilization of structures and also changes the instruction flow through the pipeline and hence, DVFS impacts the AVF. They define "vulnerability efficiency" of a DVFS algorithm as AVF $\times$ Watts/IPC, which shows the ability of a DVFS algorithm to reduce AVF and power without degrading the performance (IPC = instruction per cycle). They observe that most DVFS algorithms increase AVF and thus, applying DVFS can increase the failure rate of a system. In comparison, non-DVFS (where all domains are run at same frequency) performs better in the terms of vulnerability efficiency.

Mukherjee et al. [3] also discuss timing vulnerability factor (TVF). For example, if a circuit accepts data instead of holding data, and operates 50 percent of the time, its TVF is 50 percent, since for remaining time, an erroneous value will be overridden by a correct value. TVF is generally assumed to be included in the raw error rate.

Weaver et al. [36] present a technique to reduce the probability of a transient fault affecting the ACE state (e.g., an instruction) by keeping them in protected memory and bringing them in vulnerable structures only when needed. A cache miss on load stalls the processor and during this time, instructions stay in pipeline for extended periods. To reduce their vulnerability, their technique squashes those instructions that are younger than the load that missed. To study its trade-off with performance, they propose a metric named mean instructions to failure (MITF). They show that as long as the decrease in IPC due to their technique is smaller than the decrease in the vulnerability, their technique allows the processor to complete more work on average before seeing an error.

4.2 Methods for Computing Reliability Metrics

Several researchers present methods for accurate/efficient estimation of these metrics, as shown in Table 1. Some of these studies use fault-injection campaigns [9], [66], [72], [75]. Others use offline-analysis based estimation [73], [88], [95], [98] and runtime-only estimation [35], [79], [93] approaches.

Biswas et al. [35] present lifetime analysis approach for computing AVF of address-based structures, such as data cache, data translation buffer and store buffer. Whether a bit in the cache is ACE depends on the sequence of accesses to the cache. For a write-back cache, a cache block is ACE between Fill/Read/Write $\Rightarrow$ Read and Write $\Rightarrow$ Evict. During other sequences, such as Fill/Read/Write $\Rightarrow$ Write and Fill/Read $\Rightarrow$ Evict, the block is unACE. Using this, the AVF of a bit can be computed as the fraction of its lifetime during which it contained ACE state. The AVF of a structure is defined as the average AVF of all its bits.

Li et al. [72] present an approach for computing AVF of both logic and storage structures of the processor at runtime. Their approach works by injecting a fault in a structure during program execution and determining whether the fault propagates to create a failure. Several such fault-injections are performed and the percentage of injections that lead to failure gives an estimate of AVF.

Nair et al. [88] present a first-order mechanistic analytical model for computing AVF of important microarchitecture structures (such as reorder buffer, load and store queues, issue queue, and functional units) by correlating their utilization with the AVF. They logically divide the program execution into multiple intervals and model the occupancy of state in each interval. A weighted average of occupancy of correct-path state during each interval is computed. Using this, the AVF of a structure is estimated by derating the occupancy with the average proportion of un-ACE bits induced in the structure by the workload.

Walcott et al. [73] use regression to study the relationship between AVF and various microarchitectural variables such as number of instructions executed, structure occupancy etc. In their approach, regression analysis is performed in an offline manner to identify the relevant subset of processor events which can help in lightweight prediction of AVF at runtime. Using their approach, they compute AVF of load/store queue, issue queue and the combined reorder buffer and physical register file. Based on their analysis, they enable redundancy only when the vulnerability of multiple processor structures goes above a threshold. This helps in mitigating the overhead of redundancy in case of low vulnerability, which translates into performance improvement. Similarly, Duan et al. [98] use boosted regression trees to identify correlation between a small set of processor metrics and AVF, using which AVF can be quickly computed.

4.3 Simulation Methodology for Modeling Reliability

Coskun et al. [109] present a simulation methodology to analyze reliability of multi-core SoCs (system-on-chip). They note that both statistical- and architectural-level models have limitations, since statistical models use high abstraction level, while architectural models are too detailed and time-consuming to be feasible for large multi-core SoCs. Their simulation methodology fills this gap by analyzing reliability at core level. It allows studying the reliability implications of design-choices such as thermal packaging and placement, workload distribution and different power management strategies.

5 Techniques for Improving Reliability

We now discuss several techniques for improving reliability. For convenience, we roughly divide them in several categories.

5.1 Redundancy Based Techniques

Kim and Somani [31] propose an error protection approach for L1 caches. Due to temporal locality of caches, a small portion of data items are frequently accessed. Based on this observation, their technique uses separate circuits to provide error protection for only the frequently accessed cache blocks. Compared to the conventional approach of providing error correction for all cache blocks, their technique reduces the area overhead of error protection.

Zhang et al. [27] propose an in-cache replication scheme for protecting caches. In their technique, the blocks which have not been used for a certain period as marked as dead. The space of these blocks is used to hold replicas of the active cache block. They explore two replication strategies. In the first strategy, a block is replicated at cache misses and write-accesses, while in the second, a block is replicated only at write-accesses. The second strategy does not replicate the read-only data, and hence, it can replicate a higher percentage of modified data. For this reason, it provides a good balance of performance and reliability.

Zhang [28] present replication cache (R-cache) where a small fully associative cache is added to keep the replica of every write to the L1 data cache. Due to the high temporal locality in L1 cache, only a few (e.g. 8) blocks can provide replicas for almost all the read hits in L1 cache. However, using a R-cache for L2 cache (or LLC) becomes infeasible due to the reduced temporal locality in L2 caches [22], [111]. Due to this, a large-sized R-cache would be required which, due to its fully-associative design will also incur high dynamic energy overheads.

Kim [29] proposed an approach to ensure cache reliability with minimal area overhead. In his scheme, only dirty cache blocks are protected using ECC. The clean blocks are protected using parity check code only, which reduces the requirement of ECC protection. Further, periodically the dirty cache blocks which are not expected to be modified in near future are written back. This reduces the number of dirty cache blocks in a manner that does not significantly increase the traffic to main memory.

Sugihara et al. [44] present two schemes for improving the reliability of caches. In the error-detection approach, the data of one cache way is copied in another way and thus, two cache ways are regarded as a redundant pair to constitute a single reliable cache way. By comparing the contents of these cache ways, any error can be detected. In the error-correction approach, the data of one cache way is copied in two other cache ways and thus, three cache ways constitute a redundant pair. If an error occurs in a cache way, it is detected and corrected by using the data value from

majority of cache ways. The limitation of their schemes is that error-detection and error-correction schemes reduce the cache capacity to 50 and 33 percent, respectively.

Lee et al. [45] present an approach for error protection using partially protected caches. In embedded systems, multiple (e.g. two) caches are used at the same level of memory hierarchy and each memory address is mapped exclusively to one of these caches. They propose protecting one of the caches against soft errors by ECC and leaving the other one unprotected against soft errors. They partition the application data into failure critical and failure non-critical and map the former into the protected cache while the latter into the unprotected cache. Using this, the failure rates of multimedia applications can be kept same as that in an architecture with a single ECC protected cache while also minimizing the overheads with minimal degradation in the quality of service (QoS).

Sundaram et al. [77] use a selective instruction replication approach for multimedia applications to minimize the overhead of replication while incurring only small loss in fidelity. Their technique uses compiler to identify the instructions that are intolerant to errors and then protects only such instructions. Address calculations and conditional branches are protected since an error in them may lead to segmentation fault or loading of incorrect data values. The instructions that operate on data are not protected since small errors can be tolerated in multimedia applications.

Memik et al. [71] note that during the execution of applications, many registers are not used. Based on this, they propose two strategies for duplication of actively-used physical registers in unused registers. In the 'conservative' strategy, the active registers are duplicated in the inactive registers without affecting the performance of the processor. During high register pressure, this strategy does not perform any duplication. The 'aggressive' strategy marks the registers which are not used for a long time as dead and uses them for duplicating the active registers. This strategy aims to increase the accesses to registers with duplicates at the cost of small performance loss.

Many embedded processors use heterogeneous register files, consisting of general purpose and special purpose registers. Tabkhi and Schirner [64] note that in several applications, general purpose registers are frequently used and are thus susceptible to errors, while special purpose registers remain mostly unused. For such use patterns, they propose mirroring the contents of vulnerable registers into unused registers for improving the overall reliability of register file.

Wells et al. [105] note that in systems using redundant execution for ensuring reliability, not all applications may require high reliability; in fact, some applications may require high-performance and hence, incurring the overhead of redundant execution for such applications is wasteful. They propose a technique which allows the applications that demand high performance to avoid the penalty of dual-mode redundancy (DMR), while ensuring that most applications, including the system software can run with high reliability in DMR mode. For performance-oriented applications, their technique turns off the DMR mode. To preserve the integrity of reliability-oriented application's memory and register state, they maintain a small amount of redundancy for non-DMR (i.e. performance-oriented) applications by revalidating permission for any stores that miss in L1 cache. If access happens to a physical address which is not owned by the non-DMR application, suitable actions can be taken to avoid any corruption. To allow additional software threads for performance-oriented applications, their technique uses hardware virtualization approach to flexibly assign threads to cores. Their technique also ensures that all privileged software always runs in reliable mode. They show that compared to a DMR system, their technique improves performance significantly.

5.2 Minimizing Vulnerability and Protecting Vulnerable Data/Components

Gandhi and Mahapatra [78] present a technique to reduce vulnerability of combinational circuits. Their technique divides an input operand into subwords and encodes the subwords with all zeros or all ones with smaller number of bits. This reduces the average number of vulnerable bits which helps in reducing the vulnerability of structures where the data may reside such as register file, issue queue etc. Further, for certain input subword combinations, the complex operations inside a functional unit are replaced by simple alternative operations. For example, the result of addition of a subword with all zeros and a subword with all ones can be easily obtained without doing the actual addition. This reduces the vulnerability of the functional unit.

Fu et al. [92] propose a technique to reduce vulnerability of issue queue in simultaneous multithreaded (SMT) processors. Their technique performs offline profiling to identify reliability-critical instructions and prioritizes their scheduling to reduce the residency time of vulnerable bits in the issue queue. Further, when an instruction is in the dispatch stage, the processor checks for a free entry in the issue queue. An instruction is dispatched to the issue queue only when its utilization drops below a threshold. Using this, the quantity of vulnerable bits that can enter into the issue queue is controlled.

A large number of blocks in the cache store "dead" data, which will not be used in the future [10], [20], [112]. These data blocks remain vulnerable to soft errors. To address this, some researchers present techniques to write-back dirty data or flush the cache to reduce the chances of vulnerable blocks in the cache getting corrupted. Asadi et al. [18] propose periodically (e.g. every 1M cycles) writing the dirty data of L1 cache back to the L2 cache. Li et al. [7] propose an early-write-back policy which writes-back the dirty data of L1 cache after a fixed time period (e.g. 10 K cycles) have elapsed from the last write operation to a block. Gold et al. [30] present a prediction mechanism which accurately determines when a cache block is written for the last time and initiates an early write-back of the data.

Kadayif and Kandemir [53] present techniques to improve the reliability of caches. Their first technique writes-back only the modified words of a cache block to prevent an error from propagating to lower levels of memory hierarchy. Their second technique selectively invalidates cache blocks to reduce their vulnerable periods, which decreases their chances of catching soft errors. To minimize the performance overhead of cache block invalidations, they propose another technique which attempts to bring a

fresh copy of the invalidated block into the cache via prefetching, although this technique also loses a portion of the reliability enhancement achieved by the second technique.

Sridharan et al. [33] present a technique for write-through L1 cache which is not protected by ECC, assuming that the L2 cache and main memory are protected by ECC. They note that the utilization of L2 bus remains low which provides scope for slightly increasing L2 accesses to reduce the vulnerability of L1 cache. Their technique selectively refetches cache blocks from the L2 cache to the L1 cache to *refresh* the cache blocks which reduces the vulnerability of the L1 cache. They propose two refetch strategies. The 'static' strategy refetches selected cache blocks at a fixed frequency. The 'event-driven' strategy triggers a refetch on an L1 read hit to those blocks which are expected to be accessed in near future. Compared to the first strategy, the second strategy aims to take the workload access pattern into account to effectively reduce the vulnerability.

Yoon and Erez [8] observe that due to low soft-error rate, an error occurs only once a few weeks or months in a processor, and hence, a high latency of error-correction at last-level cache can be tolerated. Also, a large fraction of clean data stored in LLC is also stored elsewhere in the memory hierarchy, and such data can be easily corrected by restoring from a copy. Based on these, they propose decoupling error-detection from error-correction and storing the ECC in the low-cost off-chip DRAM instead of dedicated SRAM arrays. Further, since on-chip resources are not employed for error-correction, stronger ECC codes can be used for providing higher protection level. Sadler and Sorin [40] also decouple error-detection from error-correction and use low-cost error detection code (EDC) for reads and compute ECC for every write. This ECC is used for correcting an error.

Li et al. [7] study the impact of soft errors on a cache with no leakage energy optimization, and with state-preserving (viz. drowsy cache [113]) and state-destroying (viz. decay cache [112]) leakage control techniques [10], [111]. Drowsy cache technique uses dynamic voltage scaling to reduce leakage power while still retaining the data. When in a reduced voltage ("drowsy") state, the cache cells are more susceptible to soft errors since the charge required to flip the value is also reduced. Decay cache technique detects dead cache blocks and scales the supply voltage to these blocks to zero. This reduces the residency time of dead blocks in cache which reduces the susceptibility of cache to soft errors. Due to these factors, they observe that compared to a baseline cache, drowsy cache technique, while bringing large leakage energy savings, also sees significant increase in soft errors. By comparison, decay cache technique reduces both the leakage energy and the soft errors and thus, optimizes for both reliability and energy efficiency.

5.3 Compiler Based Techniques

Compiler optimizations can impact reliability in several ways, e.g. by changing the memory access pattern of a program or the occupancy of a processor component, reducing the execution time which reduces the time an application is exposed to errors etc. Similarly, code transformations such as loop interchange and loop fusion, data layout transformations such as array placement and interleaving change the read/write pattern of variables in the cache. Some researchers study these interactions in detail [38], [61], [79], [89] and show that by carefully selecting the compiler optimizations, a fine balance between performance and reliability can be obtained.

Feng et al. [99] present a compiler-based technique for finding static instructions that are likely to result in user-visible faults without first exhibiting symptomatic behavior such as memory access exceptions. Their technique protects only these instructions using software-based duplication. Their approach trades-off full coverage of errors for reducing the performance and cost overhead of duplication and hence, their approach is not suitable for mission-critical systems.

Xu et al. [65] present a compiler optimization approach to improve the reliability of register files by reordering the instructions. Their basic-block scheduling algorithm uses dynamic programming strategy under the constraint of instruction dependencies. Their approach aims to decrease the time periods during which registers are susceptible to soft errors by re-arranging the code execution flow. For example, for an operand, the using instructions is moved closer to the defining instruction.

Lee and Shrivastava [60] present a compiler technique which uses inter-procedural code analysis to reduce the register file vulnerability by temporarily writing live variables to protected memory. Their technique uses integer linear programming to find the program points where save/restore instructions can be inserted to maximally reduce register vulnerability for a given performance bound and with minimum code size transformation overhead. Their technique tries to identify long vulnerable intervals and the variables with long live-range and then selects the most profitable ones using a cost-benefit analysis. They also use several optimizations to reduce the overhead of code transformations.

Tavarageri et al. [43] present a compiler technique to detect soft errors. They note that between the definition of a variable and its subsequent uses, the variable is susceptible to errors. In their technique, a variable definition contributes to a definition checksum and each use of the variable contributes to a use checksum. The number of uses of the variable is also tracked. At program termination, an error can be detected if the definition checksum scaled by the number of uses does not equal the use checksum. In presence of multiple errors, a seemingly correct checksum may still be produced. To increase the fault coverage by addressing this issue, they propose use of multiple (e.g. two) checksums.

5.4 Exploiting Narrow Data Values

Ergin et al. [52] note that many data values generated in a processor are narrow. Based on this, they use additional flags to identify the narrow values and any soft error in zero portions of these values are detected by using these flag bits. Further, the soft errors in the narrow values can be detected or corrected by replicating the vulnerable portion of the data values inside the storage space of the zero bits.

Amrouch and Henkel [63] present a technique to improve the reliability of register files by exploiting narrow values. They propose using the upper unused bits of a register to store the ECC bits. This avoids the need of extra bits, which reduces the area and power overhead.

Hu et al. [68] present an approach to exploit narrow-width register values by making a duplicate of the value within the same data item. For example, for a 64-bit processor, the narrow-width values which require no more than 32 bits can be easily duplicated in the 64-bit register. To avoid signaling unnecessary errors in the duplicate value, their technique checks the parity bit of only the lower 32-bit half for error detection and utilizes the duplicate value in the upper half for error recovery. Thus, their technique eliminates the requirement of additional copy registers and its impact on bandwidth.

5.5 Reducing the Overhead of Reliability Improvement Approaches

Hari et al. [67] present an approach termed 'Relyzer' to reduce the number of faults that require detailed simulation in a fault-injection campaign. Their approach systematically analyzes all application fault sites and selects a small subset to perform transient fault injections. For pruning the faults, they use static analysis and dynamic profiling of the fault-free execution to predict the outcome of faults or show them equivalent to other faults.

Hari et al. [107] present an approach named 'GangES' to reduce the simulation time of fault-injection experiments. Their approach is based on the observation that a set of fault simulations that result in the same intermediate execution state after fault-injections will produce the same outcome and hence, only one of those faults need to be simulated. The decision about when to compare simulations and what state to compare is made based on the program structure. They have also shown that using GangES in conjunction with Relyzer can further reduce the simulation requirements of fault-injection experiments.

Luo et al. [81] present an approach to improve the reliability of the memory system while reducing the overall system-cost. They show that due to masking of errors, data-intensive applications show different amount of tolerance to memory errors, both within an application's data and among different applications. For example, in WebSearch application, an error in stack memory region is much more likely to lead to a crash than one in the heap and private memory regions. Thus, applying uniform protection to all memory-regions/applications is cost-inefficient. Towards this, they propose classifying the applications based on the memory error tolerance and then mapping the application to heterogeneous-reliability memory system. For example, error-tolerant portions of data from an application can reside in inexpensive less-tested memory with no ECC and the portions of data which are vulnerable to errors can reside in ECC memory.

5.6 Techniques Related to NVMs

Swaminathan et al. [90] use STT-RAM as a soft-error immune memory to provide protection to storage structures such as reorder buffer, issue queue and load-store queue. Their method identifies time periods when a snapshot of the invariant micro-architectural state can be stored in the STT-RAM and later restored to reduce soft error vulnerability. Their technique tracks the AVF of storage structures and the system throughput. Using this, protection is provided during periods of high AVF and low throughput, to maximally reduce the vulnerability while incurring minimal performance loss since the writes to STT-RAM are slow and hence, a snapshot should be written to STT-RAM when the throughput of the pipeline is low. The checkpoint is restored after a fixed time period. The benefit of STT-RAM is that it can be power-gated when idle without losing the data which isolates it from supply voltage fluctuations and leakage concerns.

Sun et al. [106] study the use of STT-RAM technology in improving the reliability of caches since STT-RAM is immune to particle-strike induced soft errors. 3D stacking technology enables modular integration of STT-RAM caches with minimal impact on processor design [114]. They evaluate several configurations where different levels of cache hierarchy are implemented in SRAM, STT-RAM etc. For example, they study (a) STTRAM L2 + STTRAM L3, (b) SRAM L2 + STTRAM L3 and (c) SRAM L2 + STTRAM L3 with ECC of L2 in L3 cache. To reduce the soft errors in L1, they also study an STT-RAM L1 design. Each configuration shows different trade-off between performance, reliability, temperature and power characteristics. For example, they observe that replacing all levels of caches with STT-RAM eliminates nearly all soft errors and also improves performance and energy efficiency.

Awasthi et al. [21] present architectural scrub mechanisms to address drift phenomenon in MLC PCM. On using an ECC code which can correct Z errors, if a scrubbing operation can be performed after Z errors but before the occurrence of $Z + 1^{\text{th}}$ error, the data can be protected. They note that with increasing Z value, the gap between Z^{th} and $Z + 1^{\text{th}}$ error also increases and hence, they suggest use of multi-bit ECCs which can allow longer scrub intervals. They also propose optimizations to simple scrub mechanisms. For example, if the Z^{th} and $Z + 1^{\text{th}}$ errors happen in quick succession, the data value becomes uncorrectable. To avoid this, they propose a 'headroom' scheme, where a scrub operation is triggered on detecting $Z - h$ errors (instead of Z errors), where h signifies the headroom. Since scrub operations introduce extra writes and thus increase hard error rates due to limited write endurance of PCM, the choice of h allows a tradeoff between hard error rates and uncorrectable soft error rates.

Seong et al. [25] note that one of the four states (levels) in a four-level PCM sees rapidest resistance drift. By removing this state, the errors produced by this state itself and the nearby state can be removed. Based on this, they propose a three-level PCM which achieves five orders of magnitude reduction in the error rate over the four-level PCM by removing the most error-prone state. They show that the three-level PCM achieves performance close to that of single-level cell (SLC) PCM and the soft error rate lower than that of DRAM without requiring ECC or scrubbing schemes. To utilize three-level cells, they also present methods to convert binary information to the ternary number system and vice versa.

5.7 Effect of 3D on Reliability

3D integration technology brings novel opportunities and challenges for ensuring reliability. For 2D planar chips, energetic particles have an unobstructed path to the active

surface of the entire chip. In a 3D chip, multiple dies are stacked on top of each other and thus, the particles need to penetrate through multiple layers before reaching the inner layers. Due to this, different layers have different vulnerability to errors. At the same time, for a 3D die-stacked memory, a failed dual in-line memory module (DIMM) cannot be easily replaced since this may destroy the whole package including the processor die. To address these issues, several techniques have been proposed, as shown below.

Zhang and Li [74] characterize soft error vulnerabilities across the stacked layers under 3D integration technology. They show that the outer-dies can shield more than 90 percent particle strikes for the inner-dies which leads to a heterogeneous error rate across layers in a 3D chip. 3D integration also provides opportunities of heterogeneous integration, since different layers can be designed using different technologies [114]. Using this feature, the outer layer can be designed using silicon-on-insulator (SOI) technology which is more resilient to particle strikes. This helps in achieving the reliability target with significantly lower costs. They also propose microarchitecture design optimizations to take advantage of the 3D integration for improving the reliability. For example, mapping ACE bits to the robust layers while leaving unACE bits to the vulnerable layers leads to protection of vulnerable program states by the shielding effect of 3D integration. Further, for narrow-width operands, the most significant bits (MSBs) are likely to be zero, and hence, they use reliability hardened circuits to implement the MSB segments of the 3D register file entries. These circuits make it harder for cell value to flip from 0 to 1 and thus reduce the soft error rate of the 3D register files.

Sun et al. [56] note that in a four-layer 3D stacked architecture, the innermost layer is invulnerable to soft-errors. Based on this, they propose allocating the most vulnerable component of cache memory onto the invulnerable layer to improve reliability and using no ECC for that layer which reduces the latency and energy incurred in accessing that layer. To maximize the accesses to the invulnerable layer, they propose dynamically moving the most recently used data-items to the invulnerable layers since they are likely to be reused [10], [22].

Madan and Balasubramonian [103] propose using 3D stacking to facilitate redundant execution. They implement a checker core on the die which is placed above the CPU die in a 3D stacked architecture to verify the execution results. The benefit of this is that the CPU die and the checker die can be independently fabricated and the checker core can be implemented with older process technology that is more resilient to soft errors and also helps in reducing the temperature and power-density. Further, 3D stacking allows use of shorter inter-die wires [114] which enables efficient communication of results between cores.

Sim et al. [55] note that in a 3D die-stacked DRAM last-level cache architecture, adding an extra chip in the stack for storing ECC information may not be practical since this leads to significant bottleneck for ECC check. To avoid this, they propose relying on main memory to correct errors in clean blocks in the DRAM cache and use duplication of dirty blocks in other banks to provide error-correction for those blocks. Further, to avoid the capacity overhead of duplication, they propose writing dirty blocks from the DRAM cache to main memory. To further optimize it, they suggest providing duplication to only critical applications or memory regions.

5.8 Studies on Real Machines

Several researchers perform reliability studies on real machines which provide valuable insights and directions for designing resilient systems. We now summarize a few of them.

Hwang et al. [86] study data on DRAM errors collected from a wide range of production systems. They analyze the characteristics of the errors, including their type (e.g. soft or hard error), their distribution to different components of the system, and their correlation etc. Based on the error patterns observed, they also evaluate the potential of different protection mechanisms. For example, they observe that simple page retirement policies can mask a large number of errors while sacrificing only a small amount of available DRAM.

Using experiments on real hardware, Biswas et al. [17] confirm the results obtained from simulations [16], [18] that an increase in cache size can lead to superlinear increase in its soft error ratio. The larger-sized cache sees fewer early write-backs due to lower miss-rate which increases the ACE residency times of dirty data in the cache and thus, increases the vulnerability of the cache.

Sridharan et al. [57] study DRAM and SRAM faults in large-scale computing systems to get insights into the factors that affect the faults in production settings. They observe that both transient and permanent fault rates vary significantly across different DRAM vendors. They study the effect of aging on DRAM and observe that over time, the nature of faults observed changes from primarily permanent faults to primarily transient faults. Further, the faults were observed to be uniformly distributed across DRAM (e.g. row, column and bank addresses). Furthermore, based on the study of faults in SRAM, they confirm that higher fault-rates are observed at higher altitude which is due to increased cosmic ray-induced neutron strikes at higher altitude.

5.9 Techniques for GPUs

GPUs have been conventionally used for graphics applications such as video gaming, where 100 percent accuracy is not necessary. Due to this and performance-optimized design of GPU, the reliability improvement techniques in GPU have received less attention. However, as GPUs become employed in general-purpose applications, their architecture is undergoing major changes [15] and hence, improving their reliability has become an issue of paramount importance. We now discuss a few techniques proposed specifically for GPUs.

Palframan et al. [70] present a technique for improving reliability of GPUs. They note that for several floating-point intensive GPU applications, small magnitude errors have negligible effect on results while large magnitude errors may get amplified to have a significant negative impact. Based on this, they propose a precision-aware protection approach for GPU register file and execution logic to mitigate large magnitude errors. Their approach uses selective logic hardening to protect significand value in a fused multiply-add (FMA)

floating-point unit, such that the choice of gates to be hardened is made depending on the relative magnitude of error that a fault in that gate can create. Further, a low-cost checker circuit is used to perform redundant computations for verifying only the MSBs of the significand computation. Furthermore, for protecting the register file, the most significant bits of the register are stored in the hardened cells which allows protection against large magnitude errors.

Farazmand et al. [66] perform fault injection experiments for three GPU structures, namely register file, local memory and active mask stack (AMS) to compute their AVF. Since their benchmarks do not fully utilize these structures, they observe low AVF values for these structures, for example, the AVF of AMS was only 0.58 percent. To address this, they propose 'AVF-util' metric which is computed by limiting the fault injection to 'utilized' portions of the structures and this value for AMS was found to be 40 percent. They note that structures with similar function in GPU and CPU are not necessarily similar in terms of AVF which highlights the need of performing GPU-specific study of reliability improvement techniques.

Tan et al. [69] study soft error vulnerability of GPUs. They observe that several microarchitectural structures in GPUs such as warp scheduler, registers, and streaming processors (SPs) exhibit high vulnerability. Also, the vulnerability depends on workload characteristics such as its memory access pattern, branch divergence etc. Further, since the threads running in different SMs have different memory access pattern and utilization, the vulnerability also varies across the SMs. They also study the effect of different architectural policies and optimizations on the vulnerability of GPUs. For example, the number of threads supported by GPU has significant effect on the vulnerability while the warp scheduling policy has little effect on the vulnerability.

Tan and Fu [101] note that while running GPU applications, the streaming processors remain idle for a large fraction of execution time. They propose two techniques which utilize these idle resources for soft error detection via redundant execution. The first technique reuses the *fully* idle SPs caused by long latency memory accesses and load imbalance among the streaming multiprocessors (SMs). During branch divergence, several SPs become *partially* idle and their second technique utilizes such SPs. This technique uses idle SPs in the diverged warp to redundantly execute threads of another warp for improving reliability.

Dimitrov et al. [59] present three techniques to improve the reliability of GPUs. First technique simply duplicates the GPU kernel and thus, reduces the throughput of the program by half. Remaining two techniques take advantage of the parallelism between the original code and its redundant version to interleave their execution. The second technique is based on the fact that redundant instructions are independent from the original code and thus, can be interleaved to create compact schedules. The third utilizes unused thread-level parallelism and thus, uses extra threads to fully utilize the GPU hardware. Using experiments, they show that while the first technique provides consistent performance, the benefit of the last two techniques varies across the applications and the GPU devices.

Fang et al. [87] present a fault-injection technique for measuring reliability of GPU applications. In their technique, the threads are grouped based on similarity of behavior and one thread is chosen from each group. Then, the GPU execution trace is obtained for the chosen thread. This information is used to map the CUDA source-code lines to the corresponding assembly instructions. Then, the application is executed and their technique chooses an instruction to inject a fault using information from the profiling phase. At this instruction, a conditional breakpoint is set up using cuda-gdb. When the chosen thread reaches the chosen source line, the breakpoint is triggered. When this happens and the chosen instruction is reached, a fault is injected into the application. Afterwards the application is monitored to determine if the fault is activated within a window of certain instructions (1,600 in their experiments). The fraction of injection runs where the fault is not activated shows the resilience of the application. Using this approach, they characterize error resilience of several GPU applications.

6 Future Challenges and Conclusion

With recent growth in the use of internet, several services are now provided online in a 24×7 manner to users situated across the globe. These services include e-learning, e-commerce, entertainment, and social-networking etc. The sheer volume and scale of infrastructure (e.g. data-centers, networks, storage etc.) required to support these services, along with the social and financial consequence of a failure, indicate that resiliency will become even more important in the design of next-generation computing systems.

The ever-increasing demands of computational capability has motivated the researchers towards design of extreme-end supercomputers. These systems will deal with hundreds of exabytes of data and will use aggressive power-management techniques. Under such constraints, approaches such as redundant execution and frequent data-backup will incur prohibitive overhead. Under high error rates, these systems may only run a few days before some part requires rebooting. To ensure reliability at such extreme-scale, highly effective and low-overhead solutions will be required. At the same time, these solutions need to be economical to justify their use in mainstream computing market.

We believe that in near future, the challenges of reliability need to addressed simultaneously at different abstraction levels. At circuit level, novel fault models need to be developed based on a rigorous understanding of modern fabrication technologies. At architecture and application level, efficient error correction and mitigation mechanisms need to be designed while also leveraging the error-resiliency of underlying algorithms. Finally, at system-node and cluster level, intelligent assignment of "reliability quota" across different cores or servers need to be done to balance reliability, performance, power and financial cost. Ensuring synergistic design and operation of techniques operating at different levels is a major research challenge.

In this paper, we presented a survey of techniques for improving the reliability of computing systems. We underscored the crucial importance of reliability in future systems and identified research directions that merit further exploration. We also classified the works based on several key parameters to highlight their similarities and differences. We hope that this survey will spark interesting future work in the area of design of resilient computing systems.

ACKNOWLEDGMENTS

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

REFERENCES

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov./Dec. 2005.

[2] J. F. Ziegler, and H. Puchner, *SER—History, Trends and Challenges: A Guide for Designing with Memory ICs*. San Jose, CA, USA: Cypress, 2004.

[3] 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 *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 29–40.

[4] E. Normand, "Single-event effects in avionics," *Trans. Nucl. Sci.*, vol. 43, no. 2, pp. 461–474, Apr. 1996.

[5] W.-C. Feng, "Making a case for efficient supercomputing," *Queue*, vol. 1, no. 7, p. 54, 2003.

[6] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *Int. J. High Performance Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014, DOI. 10.1177/1094342014522573.

[7] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Soft error and energy consumption interactions: A data cache perspective," in *Proc. Int. Symp. Low Power Electron. Design*, 2004, pp. 132–137.

[8] D. H. Yoon and M. Erez, "Memory mapped ECC: Low-cost error protection for last level caches," in *Proc. Int. Symp. Comput. Archit.*, 2009, pp. 116–127.

[9] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2010, pp. 477–486.

[10] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Elsevier Sustainable Comput.: Informat. Systems*, vol. 4, no. 1, pp. 33–43, 2014.

[11] R. Riedlinger, R. Arnold, L. Biro, B. Bowhill, J. Crop, K. Duda, E. S. Fetzer, O. Franza, T. Grutkowski, C. Little, C. Morganti, G. Moyer, A. Munch, M. Nagarajan, C. Parks, C. Poirier, B. Repasky, E. Roytman, T. Singh, and M. W. Stefaniw, "A 32 nm, 3.1 billion transistor, 12 wide issue itanium processor for mission-critical servers," *J. Solid-State Circuits*, vol. 47, no. 1, pp. 177–193, Jan. 2012.

[12] S. Rusu, H. Muljono, D. Ayers, S. Tam, W. Chen, A. Martin, S. Li, S. Vora, R. Varada, and E. Wang, "Ivytown: A 22nm 15-core enterprise Xeon® processor family," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2014, pp. 102–103.

[13] V. Zyuban, S. Taylor, B. Christensen, A. Hall, C. Gonzalez, J. Friedrich, F. Clougherty, J. Tetzloff, and R. Rao, "IBM POWER7+ design for higher frequency at fixed power," *IBM J. Res. Develop.*, vol. 57, no. 6, pp. 1:1–1:18, 2013.

[14] E. J. Fluhr, J. Friedrich, D. Dreps, V. Zyuban, G. Still, C. Gonzalez, A. Hall, D. Hogenmiller, F. Malgioglio, R. Nett, J. Paredes, J. Pille, D. Plass, R. Puri, P. Restle, D. Shan, K. Stawiasz, Z. T. Deniz, D. Wendel, and M. Ziegler, "5.1 POWER8™: A 12-core server-class processor in 22nm SOI with 7.6 Tb/s off-chip bandwidth," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2014, pp. 96–97.

[15] S. Mittal, "A survey of techniques for managing and leveraging caches in GPUs," *J. Circuits, Syst., Comput.*, vol. 23, no. 8, 2014, http://www.worldscientific.com/doi/abs/10.1142/S0218126614300025.

[16] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al,-Hashimi, and S. M. Reddy, "Cache size selection for performance, energy and reliability of time-constrained systems," in *Proc. Asia South Pacific Design Autom. Conf.*, 2006, pp. 923–928.

[17] A. Biswas, C. Recchia, S. S. Mukherjee, V. Ambrose, L. Chan, A. Jaleel, A. E. Papathanasiou, M. Plaster, and N. Seifert, "Explaining cache SER anomaly using DUE AVF measurement," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2010, pp. 1–12.

[18] G.-H. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli, "Balancing performance and reliability in the memory hierarchy," in *Proc. Int. Symp. Performance Anal. Syst. Softw.*, 2005, pp. 269–279.

[19] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: Myth or necessity?" in *Proc. IEEE Pacific Rim Int. Symp. Dependable Comput.*, 2004, pp. 37–42.

[20] S. Mittal, J. S. Vetter, and D. Li, "Improving energy efficiency of embedded DRAM caches for high-end computing systems," in *Proc. 23rd Int. ACM Symp. High Performance Parallel Distrib. Comput.*, 2014, pp. 99–110.

[21] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, R. Balasubramonian, and V. Srinivasan, "Efficient scrub mechanisms for error-prone emerging memories," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2012, pp. 1–12.

[22] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches," *Trans. Parallel Distrib. Syst.*, 2014, DOI. 10.1109/TPDS.2014.2324563.

[23] H. Sun, C. Liu, W. Xu, J. Zhao, N. Zheng, and T. Zhang, "Using magnetic RAM to build low-power and soft error-resilient L1 cache," *Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 1, pp. 19–28, Jan. 2012.

[24] H. Naeimi, C. Augustine, A. Raychowdhury, S.-L. Lu, and J. Tschanz, "STTRAM scaling and retention failure," *Intel Technol. J.*, vol. 17, no. 1, pp. 54–75, 2013.

[25] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," in *Proc. Int. Symp. Comput. Archit.*, 2013, pp. 440–451.

[26] V. Sridharan, D. A. Liberty, and D. R. Kaeli, "A taxonomy to enable error recovery and correction in software," in *Proc. Workshop Quality-Aware Design*, 2008, http://research.ihost.com/quad/program.html

[27] W. Zhang, S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam, "ICR: In-cache replication for enhancing data cache reliability," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2003, pp. 291–300.

[28] W. Zhang, "Replication cache: A small fully associative cache to improve data cache reliability," *Trans. Comput.*, vol. 54, no. 12, pp. 1547–1555, Dec. 2005.

[29] S. Kim, "Area-efficient error protection for caches," in *Proc. Design, Autom. Test Eur.*, 2006, pp. 1282–1287.

[30] B. T. Gold, M. Ferdman, B. Falsafi, and K. Mai, "Mitigating multi-bit soft errors in L1 caches using last-store prediction," in *Proc. Workshop Archit. Support Gigascale Integr.*, 2007.

[31] S. Kim and A. K. Somani, "Area efficient architectures for information integrity in cache memories," *ACM SIGARCH Comput. Archit. News*, vol. 27, no. 2, pp. 246–255, 1999.

[32] K. Bhattacharya, N. Ranganathan, and S. Kim, "A framework for correction of multi-bit soft errors in L2 caches based on redundancy," *Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 2, pp. 194–206, Feb. 2009.

[33] V. Sridharan, H. Asadi, M. B. Tahoori, and D. Kaeli, "Reducing data cache susceptibility to soft errors," *Trans. Dependable Secure Comput.*, vol. 3, no. 4, pp. 353–364, Oct.–Dec. 2006.

[34] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proc. Int. Symp. Microarchit.*, 2007, pp. 197–209.

[35] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," in *Proc. Int. Symp. Comput. Archit.*, 2005, pp. 532–543.

[36] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 264–275, 2004.

[37] W. Zhang, "Computing cache vulnerability to transient errors and its implication," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, 2005, pp. 427–435.

[38] A. Shrivastava, J. Lee, and R. Jeyapaul, "Cache vulnerability equations for protecting data in embedded processor caches from soft errors," in *Proc. ACM Sigplan Notices*, vol. 45, no. 4, pp. 143–152, 2010.

[39] J. Yan and W. Zhang, "Evaluating instruction cache vulnerability to transient errors," in *Proc. Workshop Memory Performance: Dealing Appl., Syst. Archit.*, 2006, pp. 21–28.

[40] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's L1 data cache," in *Proc. Int. Conf. Comput. Design*, 2007, pp. 499–505.

[41] S. Wang, J. Hu, and S. G. Ziavras, "On the characterization and optimization of on-chip cache reliability against soft errors," *Trans. Comput.*, vol. 58, no. 9, pp. 1171–1184, Sep. 2009.

[42] J. Suh, M. Annavaram, and M. Dubois, "MACAU: A Markov model for reliability evaluations of caches under Single-bit and Multi-bit Upsets," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2012, pp. 1–12.

[43] S. Tavarageri, S. Krishnamoorthy, and P. Sadayappan, "Compiler-assisted detection of transient memory errors," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design. Implementation*, 2014, pp. 204–215.

[44] M. Sugihara, T. Ishihara, and K. Murakami, "Task scheduling for reliable cache architectures of multiprocessor systems," in *Proc. Design, Autom. Test Eur.*, 2007, pp. 1490–1495.

[45] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2006, pp. 411–420.

[46] J. Xu, R. Shen, and Q. Tan, "PRASE: An approach for program reliability analysis with soft errors," in *Proc. IEEE Pacific Rim Int. Symp. Dependable Comput.*, 2008, pp. 240–247.

[47] H. Wang, S. Baldawa, and R. Sangireddy, "Dynamic error detection for dependable cache coherency in multicore architectures," in *Proc. Int. Conf. VLSI Design*, 2008, pp. 279–285.

[48] I. Oz, H. R. Topcuoglu, M. Kandemir, and O. Tosun, "Thread vulnerability in parallel applications," *J. Parallel Distrib. Comput.*, vol. 72, no. 10, pp. 1171–1185, 2012.

[49] R. Jeyapaul and A. Shrivastava, "Enabling energy efficient reliability in embedded systems through smart cache cleaning," *ACM Trans. Design Autom. Electron. Syst.*, vol. 18, no. 4, pp. 53:1–53:25, 2013.

[50] H. Zhao, A. Sharifi, S. Srikantaiah, and M. Kandemir, "Feedback control based cache reliability enhancement for emerging multicores," in *Proc. Int. Conf. Comput.-Aided Design*, 2011, pp. 56–62.

[51] A. Chakraborty, H. Homayoun, A. Khajeh, N. Dutt, A. Eltawil, and F. Kurdahi, "E < MC2: less energy through multi-copy cache," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2010, pp. 237–246.

[52] O. Ergin, O. S. Unsal, X. Vera, and A. Gonzalez, "Exploiting narrow values for soft error tolerance," *Comput. Archit. Lett.*, vol. 5, no. 2, p. 12, 2006.

[53] I. Kadayif and M. Kandemir, "Modeling and improving data cache reliability," *ACM SIGMETRICS Performance Evaluation Rev.*, vol. 35, no. 1, p. 12, 2007.

[54] S. Z. Can, G. Yalcin, O. Ergin, O. Unsal, and A. Cristal, "Bit impact factor: Towards making fair vulnerability comparison," *Microprocessors Microsyst.*, vol. 38, pp. 598–604, 2014.

[55] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked DRAM caches," in *Proc. Int. Symp. Comput. Archit.*, 2013, pp. 416–427.

[56] H. Sun, P. Ren, N. Zheng, T. Zhang, and T. Li, "Architecting high-performance energy-efficient soft error resilient cache under 3D integration technology," *Microprocessors Microsyst.*, vol. 35, no. 4, pp. 371–381, 2011.

[57] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: positional effects in DRAM and SRAM faults," in *Proc. Int. Confe. High Performance Comput., Netw., Storage Anal.*, 2013, pp. 22:1–22:11.

[58] W. Zhang and T. Li, "Managing multi-core soft-error reliability through utility-driven cross domain optimization," in *Proc. Int. Conf. Appl.-Specific Syst., Archit. Processors*, 2008, pp. 132–137.

[59] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding software approaches for GPGPU reliability," in *Proc. Workshop General Purpose Process. Graphics Process. Units*, 2009, pp. 94–104.

[60] J. Lee and A. Shrivastava, "A compiler optimization to reduce soft errors in register files," in *Proc. ACM Sigplan Notices*, vol. 44, no. 7, pp. 41–49, 2009.

[61] T. M. Jones, M. F. O'boyle, and O. Ergin, "Evaluating the effects of compiler optimisations on AVF," in *Proc. Workshop Interaction Compilers Comput. Archit.*, 2008.

[62] J. Yan and W. Zhang, "Compiler-guided register reliability improvement against soft errors," in *Proc. ACM Int. Conf. Embedded Softw.*, 2005, pp. 203–209.

[63] H. Amrouch and J. Henkel, "Self-immunity technique to improve register file integrity against soft errors," in *Proc. Int. Conf. VLSI Design*, 2011, pp. 189–194.

[64] H. Tabkhi and G. Schirner, "Application-specific power-efficient approach for reducing register file vulnerability," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2012, pp. 574–577.

[65] J. Xu, Q. Tan, and H. Zhou, "Scheduling instructions for soft errors in register files," in *Proc. Int. Conf. Dependable, Auton. Secure Comput.*, 2011, pp. 305–312.

[66] N. Farazmand, R. Ubal, and D. Kaeli, "Statistical fault injection-based AVF analysis of a GPU architecture," in *Proc. 8th IEEE Workshop Silicon Errors Logic Syst. Effects*, 2012.

[67] 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," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2012, pp. 123–134.

[68] J. Hu, S. Wang, and S. G. Ziavras, "On the exploitation of narrow-width values for improving register file reliability," *Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 7, pp. 953–963, Jul. 2009.

[69] J. Tan, Y. Yi, F. Shen, and X. Fu, "Modeling and characterizing GPGPU reliability in the presence of soft errors," *Parallel Comput.*, vol. 39, no. 9, pp. 520–532, 2013.

[70] D. Palframan, N. S. Kim, and M. Lipasti, "Precision-aware soft error protection for GPUs," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2014, pp. 49–59.

[71] G. Memik, M. T. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," in *Proc. Design, Autom. Test Eur.*, 2005, pp. 586–591.

[72] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," in *Proc. Int. Symp. Comput. Archit.*, 2008, pp. 341–352.

[73] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 516–527, 2007.

[74] W. Zhang and T. Li, "Microarchitecture soft error vulnerability characterization and mitigation under 3D integration technology," in *Proc. Int. Symp. Microarchit.*, 2008, pp. 435–446.

[75] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *Proc. Int. Symp. Comput. Archit.*, pp. 460–469, 2007.

[76] S. Hong and S. Kim, "TEPS: Transient error protection utilizing sub-word parallelism," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2009, pp. 286–291.

[77] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient fault tolerance in multi-media applications through selective instruction replication," in *Proc. Workshop Radiation Effects Fault Tolerance Nanometer Technol.*, 2008, pp. 339–346.

[78] K. R. Gandhi and N. R. Mahapatra, "Energy-efficient soft-error protection using operand encoding and operation bypass," in *Proc. Int. Conf. VLSI Design*, 2008, pp. 45–51.

[79] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2009, pp. 117–128.

[80] L. Yu, D. Li, S. Mittal, and J. S. Vetter, "Quantitatively modeling application resilience with the data vulnerability factor," in *Proc. ACM/IEEE Int. Conf. High Performance Comput., Netw., Storage, Anal.*, 2014, pp. 695–706.

[81] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *Proc. 44th Annu. /IFIP Int. Conf. Dependable Syst. Netw.*, 2014, pp. 467–478.

[82] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2011, pp. 466–477.

[83] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2012, pp. 1–11.
[84] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proc. 15th ed. ASPLOS Archit. Support Program. Lang. Oper. Syst.*, 2010, pp. 397–408.
[85] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," *Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 337–350, Oct.-Dec. 2010.
[86] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2012, pp. 111–122.
[87] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, 2014, pp. 221–230.
[88] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John, "A first-order mechanistic model for architectural vulnerability factor," in *Proc. Int. Symp. Comput. Archit.*, 2012, pp. 273–284.
[89] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in *Proc. Int. Symp. Workload Characterization*, 2011, pp. 184–193.
[90] K. Swaminathan, R. Mukundrajan, N. Soundararajan, and V. Narayanan, "Towards resilient micro-architectures: Datapath reliability enhancement using STT-MRAM," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2011, pp. 236–241.
[91] N. Soundararajan, N. Vijaykrishnan, and A. Sivasubramaniam, "Impact of dynamic voltage and frequency scaling on the architectural vulnerability of GALS architectures," in *Proc. Int. Symp. Low Power Electron. Design*, 2008, pp. 351–356.
[92] X. Fu, W. Zhang, T. Li, and J. Fortes, "Optimizing issue queue reliability to soft errors on simultaneous multithreaded architectures," in *Proc. Int. Conf. Parallel Process.*, 2008, pp. 190–197.
[93] X. Fu, J. Poe, T. Li, and J. A. Fortes, "Characterizing microarchitecture soft error vulnerability phase behavior," in *Proc. Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst.*, 2006, pp. 147–155.
[94] N. Seifert and N. Tam, "Timing vulnerability factors of sequentials," *Trans. Device Mater. Rel.*, vol. 4, no. 3, pp. 516–522, Sep. 2004.
[95] V. Sridharan and D. R. Kaeli, "Using hardware vulnerability factors to enhance AVF analysis," in *Proc. Int. Symp. Comput. Archit.*, 2010, pp. 461–472.
[96] D. Borodin and B. H. Juurlink, "Protective redundancy overhead reduction using instruction vulnerability factor," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2010, pp. 319–326.
[97] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: embedded code generation aiming at reliability," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, 2011, pp. 237–246.
[98] L. Duan, B. Li, and L. Peng, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics," in *Proc. Int. Symp. High Performance Comput. Archit.*, 2009, pp. 129–140.
[99] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 385–396, 2010.
[100] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam, "Mechanisms for bounding vulnerabilities of processor structures," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 506–515, 2007.
[101] J. Tan and X. Fu, "RISE: Improving the streaming processors reliability against soft errors in GPGPUs," in *Proc. 21st Int. Conf. Parallel Archit. Compilation Tech.*, 2012, pp. 191–200.
[102] X. Vera, J. Abella, J. Carretero, and A. González, "Selective replication: A lightweight technique for soft errors," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 8:1–8:30, 2009.
[103] N. Madan and R. Balasubramonian, "Leveraging 3D technology for improved reliability," in *Proc. Int. Symp. Microarchit.*, 2007, pp. 223–235.
[104] S. Rehman, M. Shafique, P. V. Aceituno, F. Kriebel, J.-J. Chen, and J. Henkel, "Leveraging variable function resilience for selective software reliability on unreliable hardware," in *Proc. Conf. Design, Autom. Test Eur.*, 2013, pp. 1759–1764.
[105] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Mixed-mode multicore reliability," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 169–180, 2009.
[106] G. Sun, E. Kursun, J. Rivers, and Y. Xie, "Exploring the vulnerability of CMPs to soft errors with 3D stacked non-volatile memory," in *Proc. Int. Conf. Comput. Design*, 2011, pp. 366–372.
[107] S. Hari, R. Venkatagiri, S. Adve, and H. Naeimi, "GangES: Gang error simulation for hardware resiliency evaluation," in *Proc. Int. Symp. Comput. Archit.*, 2014, pp. 61–72.
[108] A. Savino, S. Carlo, G. Politano, A. Benso, A. Bosio, and G. Di Natale, "Statistical reliability estimation of microprocessor-based systems," *Trans. Comput.*, vol. 61, no. 11, pp. 1521–1534, Nov. 2012.
[109] A. K. Coskun, T. S. Rosing, Y. Leblebici, and G. De Micheli, "A simulation methodology for reliability analysis in multi-core SoCs," in *Proc. ACM Great Lakes Symp. VLSI*, 2006, pp. 95–99.
[110] I. S. Haque and V. S. Pande, "Hard data on soft errors: A large-scale assessment of real-world error rates in GPGPU," in *Proc. IEEE/ACM Int. Conf. Cluster, Cloud Grid Comput.*, 2010, pp. 691–696.
[111] S. Mittal, Y. Cao, and Z. Zhang, "MASTER: A multicore cache energy saving technique using dynamic cache reconfiguration," *Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 22, no. 8, pp. 1653–1665, Aug. 2014.
[112] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proc. Int. Symp. Comput. Archit.*, 2001, pp. 240–251.
[113] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proc. Int. Symp. Comput. Archit.*, 2002, pp. 148–157.
[114] M. Poremba, S. Mittal, D. Li, J. S. Vetter, and Y. Xie, "DESTINY: A tool for modeling emerging 3D NVM and eDRAM caches," in *Proc. Design Autom. Test Eur.*, 2015, pp. 1543–1546, ISBN 978-3-9815-3704-8.

Sparsh Mittal received the BTech. degree in electronics and communications engineering from IIT, Roorkee, India and the PhD. degree in computer engineering from Iowa State University, USA. He is currently working as a Postdoctoral research associate at ORNL. His research interests include resilience, non-volatile memory, memory system power efficiency, cache and GPU architectures. He is a member of the IEEE.

Jeffrey S. Vetter received the PhD degree from the Georgia Institute of Technology (GT), Atlanta, GA, USA. He holds a joint appointment between ORNL and Georgia Institute of Technology (GT). At ORNL, he is a Distinguished R&D Staff Member, and the founding group leader of the Future Technologies Group. At GT, he is a Joint Professor in the Computational Science and Engineering School, the Project Director for the NSF Track 2D Experimental Computing Facility for large scale heterogeneous computing using graphics processors, and the Director of the NVIDIA CUDA Center of Excellence. His research interests include massively multithreaded processors, non-volatile memory and heterogeneous multicore processors. He is a senior member of the IEEE.