Design Techniques for Cross-Layer Resilience

(Invited Paper)

Nicholas P. Carter, Helia Naeimi, Donald S. Gardner
Intel Corporation
2200 Mission College Blvd., RNB6-61
Santa Clara, California 95054
Email: nicholas.p.carter@intel.com, helia.naeimi@intel.com, d.s.gardner@intel.com

***Abstract*—Current electronic systems implement reliability using only a few layers of the system stack, which simplifies the design of other layers but is becoming increasingly expensive over time. In contrast, cross-layer resilient systems, which distribute the responsibility for tolerating errors, device variation, and aging across the system stack, have the potential to provide the resilience required to implement reliable, high-performance, low-power systems in future fabrication processes at significantly lower cost. These systems can implement less-frequent resilience tasks in software to save power and chip area, can tune their reliability guarantees to the needs of applications, and can use the information available at each level in the system stack to optimize performance and power consumption. In this paper, we outline an approach to cross-layer system design that describes resilience as a set of tasks that systems must perform in order to detect and tolerate errors and variation. We then present strawman examples of how this task-based design process could be used to implement general-purpose computing and SoC systems, drawing on previous work and identifying key areas for future research.**

I. Introduction

As fabrication technologies advance, increasing rates of errors [1], device variation [2], and aging [3] motivate the design of systems in which all of the layers in the system stack assume that devices and circuits will not always perform as designed. When compared to *brittle* systems, which assume perfect device fabrication and operation or *single-layer* approaches to reliability, these *cross-layer* resilient systems have the potential to deliver more-reliable operation, higher performance, lower cost, and/or lower power consumption by taking advantage of the information and capabilities available at each layer in the system stack.

Previous work on resilience has focused on phenomena-based approaches, in which designers identify the physical effects (soft errors, NBTI, etc.) that their design needs to tolerate and develop techniques to address each of those effects. In this paper, we propose an alternate approach that divides resilience into a set of five *tasks*: detection, diagnosis, reconfiguration, recovery, and adaptation, which may be implemented using hardware or software *mechanisms* at different levels of the system stack. While the use of these terms to describe aspects of resilience and reliability is not new, thinking about resilience as a set of tasks rather than a set of relatively-independent mechanisms makes cross-layer designs easier to envision and describe.

Cross-layer resilience is still is still in its infancy, and was the subject of a 2009 study (www.relxlayer.org) that was funded by the Computing Community Consortium. The goal of this paper is to inspire research into the techniques necessary to realize the study's vision [4] by presenting a structure for thinking about cross-layer resilience and two case studies that suggest possible approaches to cross-layer resilient computing systems and systems-on-chip (SoCs).

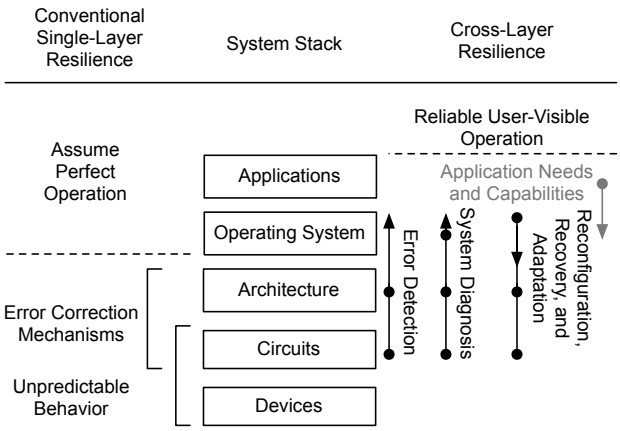

Fig. 1. System Stack and Resilience

II. Cross-Layer Resilient Design

Current systems concentrate mechanisms for resilience in the architecture and/or circuit levels of the system stack, as shown on the left side of Figure 1. While these single-layer (or few-layer) approaches to resilience simplify the design of the upper layers in the system stack by allowing programmers to ignore reliability, they also have significant drawbacks. To guarantee high reliability, single-level schemes must typically replicate computations, either across multiple functional units [5][6] or by time-multiplexing a single functional unit [7][8][9][10]. This replication imposes high overheads in power, performance, and/or chip area, although these costs can be reduced somewhat by only replicating the portions of the computation necessary to guarantee correctness [11][12][13].

More cost-conscious systems incorporate mechanisms that address the most common causes of errors, such as error correcting codes (ECC) in memory or residual arithmetic to protect datapaths [14]. This approach can be very cost-effective when system reliability is dominated by a small

number of error sources. However, as fabrication technologies advance, the total cost of the mechanisms required to handle increasing rates of multi-bit errors [15][16], temporal performance variation [17][18], aging mechanisms such as NBTI [19][20], and other effects is increasing rapidly.

In contrast, *cross-layer* resilience schemes divide error and variation tolerance into a set of *tasks*, which can be implemented by hardware or software *mechanisms* at different levels in the system stack. These resilience tasks can be thought of as steps that the system follows to handle a particular error or variation, although they may not occur sequentially. Resilience tasks are independent of the physical effects that cause errors and variation, while the mechanisms used to implement a given task may be specific to individual physical effects (e.g., implementing the detection task using separate mechanisms to detect soft errors, NBTI-induced delay faults, and voltage droops). To be as consistent as possible with previous work, we define five resilience tasks, although other task sets are certainly possible:

1) **Detection:** Determining that an error has occurred (i.e., that some fault has caused one or more bits in the computation to differ from their correct value).
2) **Diagnosis:** Characterizing the system's state to locate the causes of errors, determine how the system is changing over time, and predict errors before they occur.
3) **Reconfiguration:** Changing the state of the system to prevent an error from recurring and/or to prevent variation from causing errors.
4) **Recovery:** Ensuring that an error does not propagate to user-visible results, for example by rolling back an application and re-trying one or more failed operations.
5) **Adaptation:** Re-optimizing the system to provide the best possible performance/power given the changes to the system state made by the reconfiguration task.

The right side of Figure 1 illustrates how these tasks might be distributed across the system stack. In this figure, circles indicate levels of the stack that participate in each task, while arrows indicate the direction of information flow. Errors are detected in the lower levels of the stack, while the operating system, architecture, and circuits handle diagnosis through a combination of hardware and software. When an error occurs, the mechanisms in the circuits and architecture signal the operating system, which uses the system diagnosis and information about the application's error-handling capabilities to determine how to best respond to the error.

This approach has a number of advantages over systems that concentrate reliability in only a few layers in the stack. Cross-layer resilient systems can move less-frequent resilience tasks into software to reduce area and power overhead, either during system design or at run-time. They can also provide configurable reliability to efficiently match the needs of a given system or application by activating or de-activating reliability mechanisms based on the observed error rate, the reliability required by the application, and the sensitivity of a computation to errors. Finally, cross-layer resilient systems can take advantage of a wider scope of information when deciding how to handle an error or variation, allowing them to make more globally-optimal decisions than systems with less scope.

III. General-Purpose Computing Systems

Figure 2 illustrates an example of how a cross-layer resilient general-purpose computing system (laptop, workstation, etc.) might be implemented. For the purposes of this discussion, we assume that the system is required to be backward-compatible with older software, and thus discuss scenarios where applications both are and are not involved in resilience. As shown in the figure, the computing system's hardware consists of a many-core CPU and off-chip DRAM. The operating system incorporates four software sub-systems that contribute to resiliency: an error handler routine, a resource map, which describes the current state of the system, a hardware configuration routine that controls the system's hardware, and a task scheduler, which takes the information in the resource map into account when scheduling tasks.

A. Detection

When applications are not involved in resilience, errors are detected at run-time by a set of low-cost hardware mechanisms, such as ECC codes on memories and parity/residue codes on computations. When an error is detected, the hardware signals an error handler in the operating system, which directs the recovery, reconfiguration, and adaptation tasks.

Resilience-aware software can significantly improve error detection rates and/or reduce the amount of hardware required to detect errors. Algorithmically, it is possible to check the results of many computations in significantly less time than is required for the computation itself, although exploiting this behavior typically requires programmers to invest significant effort. Alternatively, some algorithms, such as matrix operations, can be modified to operate on data structures that incorporate checksums or other redundancy, allowing them to detect and/or correct errors [21].

Compiler-based techniques can detect many errors in software by observing common error symptoms or violations of invariants [22][23], and can insert redundant instructions to detect errors [24][25][26]. Combining a resilience-aware compiler with appropriate hardware support can also effectively detect control-flow errors, such as branches taking the wrong path, which can be hard to detect in hardware [27][28][29][30][31].

B. Diagnosis

In a cross-layer resilient system, error and system diagnosis are performed by a combination of hardware and software mechanisms. Temperature and supply voltage sensors can provide valuable information about the short-term state of the system, while delay sensors on logic paths [32][17][18], diagnostic circuits [20], and periodic hardware or software self-test [33][34][35] can help to diagnose longer-term variation and aging. Regardless of the set of diagnosis mechanisms a system implements, their outputs are sent to the OS and used to update

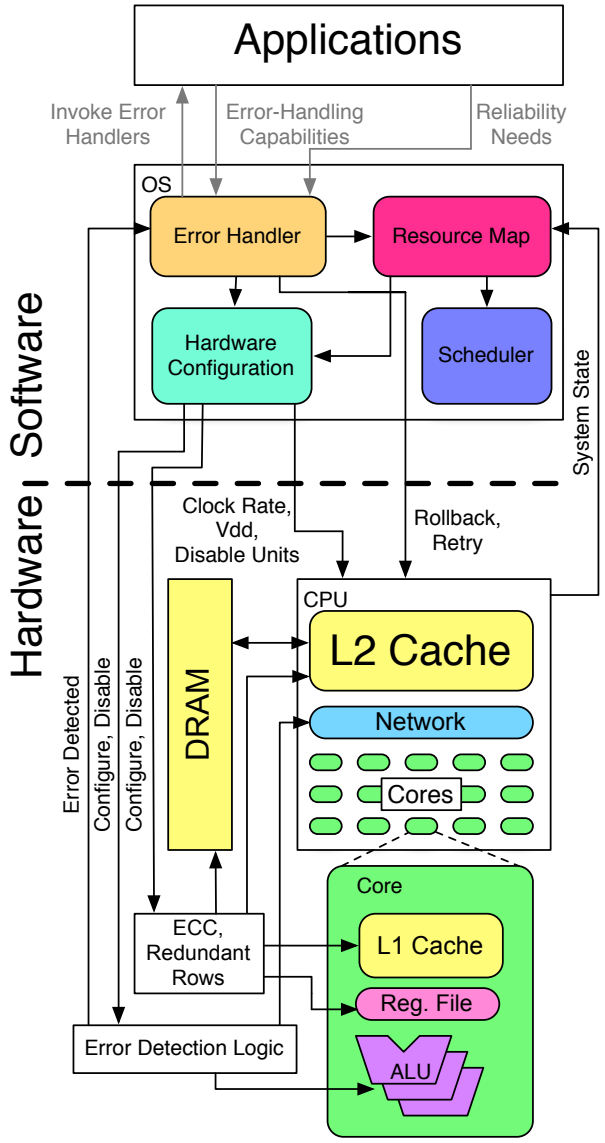

Fig. 2. Cross-Layer Resilient Computer System

its resource map with information on which units in the system are operating correctly and allowable clock rate/supply voltage combinations for each unit. System diagnosis can also help predict permanent errors before they occur by noting changes in transient fault rates and/or transistor behavior [36].

Because diagnosis is intimately tied to the system's hardware, it is difficult for application-level software to contribute significantly to this task in general-purpose computing systems. Single-purpose or embedded systems might choose to integrate the software-based test techniques mentioned above into applications to reduce the amount of system software they require, and applications that incorporate error detection techniques can inform the OS of the number of errors they detect to help it diagnose slow-onset errors.

C. Reconfiguration

There are two aspects to reconfiguration in a cross-layer resilient system: selecting a set of resilience mechanisms to use that provide sufficient reliability while maximizing performance/Watt, and adjusting the operating points and capabilities of different hardware units in response to faults, aging, and variation. Resilience-aware applications can contribute significantly to the first aspect of reconfiguration by informing the operating system of their capabilities, allowing it to disable hardware mechanisms that are not needed. Even when executing resilience-unaware applications, there is potential to select resilience mechanisms based on the system's needs, for example by selecting the amount of error-correction used in the memory or registers based on the observed error rate [37].

Techniques to reconfigure a system to account for faulty or aging hardware vary significantly with the type of hardware being reconfigured. The regular structure of memory arrays makes it possible to disable small regions of an array in response to faults [38]. Network techniques to tolerate failures are relatively well-studied, and efforts are exploring ways to apply these techniques to on-chip networks [39].

Reconfiguring execution resources involves trade-offs between the overhead of the reconfiguration mechanisms and the amount of hardware a given fault/variation can affect. Given the consequences of disabling an entire core in current-generation chips, a number of efforts have explored techniques to tolerate faults in execution units by exploiting redundant hardware in superscalar cores [40][41], combining multiple faulty cores into a single "virtual" core [42] or by relocating tasks if they try to use a faulty unit on a given core [43]. As the number of cores per chip increases, it may become more attractive to treat cores as atomic units, adjusting their clock frequency and supply voltage and/or disabling them in response to faults as long as the CPU provides mechanisms to isolate faulty cores from the rest of the system [44] and migrate tasks to healthy cores [45].

Like diagnosis, reconfiguration is a sufficiently hardware-specific task that it is difficult for applications to contribute to reconfiguration, although compiler-based reconfiguration schemes that replace instructions that require faulty hardware with software emulation [46] have been proposed.

D. Recovery

Error recovery can benefit significantly from a multi-layer approach, as single-layer recovery techniques, such as triple modular redundancy or application checkpointing [47], have very high costs. Much of the difficulty in error recovery comes from the wide variance in the delay between the time when a fault occurs and the time that the resulting error(s) are detected. When errors are detected quickly, they have little ability to impact state, and low-overhead recovery techniques, such as squashing instructions in the pipeline, are possible.

However, some errors, such as network errors and multi-bit memory errors, are inherently difficult to detect quickly and therefore require checkpointing and rollback or other more-powerful error recovery techniques, which are still very

much an open area of research. A number of projects have investigated techniques that leverage the redundant storage of data in cache hierarchies to provide low-overhead checkpointing [48][49][50]. While they can significantly reduce checkpointing overhead, these techniques have limited ability to guarantee that a checkpoint will be kept alive for a specific number of cycles, because movement of data into and out of the caches is determined by the application's memory accesses.

Integrating applications into the recovery task has the potential to greatly reduce recovery costs by only checkpointing the data that is necessary to roll back the application. Simply having the application determine when checkpoints should be taken can greatly reduce checkpointing overhead [47], while applications that provide their own checkpointing code can see even greater improvements [51]. A more general approach to application-level redundancy might involve the use of side-effect-free programming styles, in which functions are not allowed to modify any data other than their return values, making it possible to re-execute a function if an error occurs.

E. Adaptation

The adaptation task is responsible for deciding how to allocate tasks and power to the different units in the system in order to maximize overall performance without exceeding the power budget or other constraints. In our strawman system, the operating system handles adaptation, using the output from the diagnosis task and its knowledge of which applications are running to set the clock rate and supply voltage of each unit in the CPU and to determine the mapping of tasks to units.

At the application level, many of the same techniques used in implementing high-performance parallel programs also increase an application's ability to adapt to changing hardware. For example, dynamic load-balancing techniques can tolerate variations in thread run-time caused by either varying amounts of work in each sub-task or by cores operating at different clock rates. Similarly, applications that can increase or decrease the number of threads they use can adapt to changes in the number of cores available to them, regardless of whether those changes are caused by hardware faults or by other applications starting or stopping.

One challenge in adaptation is that it is currently very difficult for the system to predict how an application's performance will scale with parallelism, making it hard to decide whether it would be better to run the application on a small number of high-frequency cores or on a larger number of lower-frequency cores. A cross-layer system that provided an API for applications to pass predictions about their performance scaling to the operating system could significantly improve the OS' ability to allocate resources.

IV. System-on-chip Design

As the complexity of integrated circuits increases, more and more designers are using system-on-chip approaches to reduce design effort by re-using intellectual property (IP) blocks across multiple designs. While the task-based approach to resilience we propose applies well to SoC design, SoC designs have additional challenges and opportunities that are not present in the more-custom design techniques used in microprocessors. In this section, we outline these additional challenges and opportunities, and suggest some approaches to leverage the opportunities to overcome the challenges.

A. SoC Design Challenges and Opportunities

1) Challenge: Validation and Test: Systems-on-chip pose significant validation and test challenges. While validating each IP block at its design time is similar to conventional chip design, validating the entire SoC system is more difficult because the designer does not (and should not) have access to the internals of each IP block. This makes it harder to validate all of the corner cases in a design, leading to increased bug escapes. Fabrication test poses similar difficulties due to limited accessibility and visibility. Cross-layer resilience can help systems tolerate bugs by providing mechanisms to detect errors at runtime, retry computations when an error occurs due to a rare sequence of events, and reconfigure themselves to avoid exercising faulty circuitry.

2) Challenge: Varying Reliability Requirements: A given IP block will be used in multiple systems, which may have different reliability requirements. When a single-level approach to resilience is used, block designers are forced to choose between over-designing blocks to meet the needs of the most demanding SoCs or designing for efficiency in the "average" case and potentially not meeting the needs of highly-reliable SoCs. A cross-layer approach to resilience can allow IP blocks to meet the needs of a wide range of SoCs. For example, an IP block might provide a minimal set of error detection and retry mechanisms. SoC designers could then implement mechanisms for diagnosis, recovery, reconfiguration, and adaptation at higher levels in the system stack in order to efficiently provide the reliability required by each system.

3) Opportunity: Standard Resilient IP interface: The SoC industry currently uses a number of standardized interfaces to IP blocks that reduce system design time by allowing designers to implement one protocol for communication between the IP blocks in their design. A standardized protocol for cross-layer resilient SoCs would greatly simplify design by identifying the key information and capabilities that need to cross the IP-system boundary in order to provide reliability. Such an interface would need to incorporate three key features: a mechanism to isolate defective blocks from the rest of the system, protocols for inter-block communication about resilience, and signals to make resilience visible to the upper layers in the system stack. In the next section, we discuss how such a protocol could be used to design SoCs in more detail.

B. Cross-Layer Resilience for SoCs

Figure 3 shows an example system stack for resilient SoC design. The key differences between this stack and the one shown in Figure 1 are the division of the architecture layer into SoC architecture and IP block architecture layers and the insertion of the resilient IP interface between them.

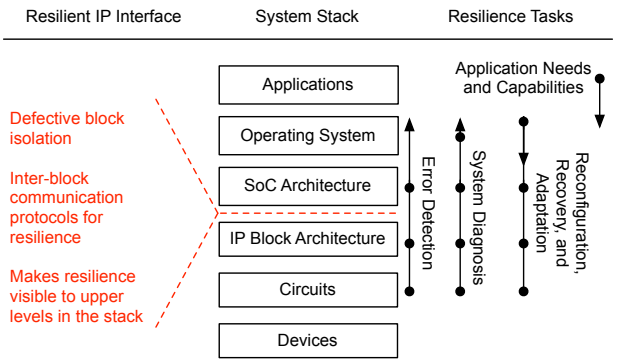

Fig. 3. Cross-Layer Resilient System-on-Chip

1) Detection: The standardized resilient IP interface will play two roles in error detection. First, it must provide mechanisms to detect and correct errors in inter-block communication, and these mechanisms should be configurable to allow different reliability/cost trade-offs, such as variable-strength ECC schemes, either by synthesizing different variants of the interface or perhaps by run-time configuration. Second, the IP interface should allow the system to configure or disable the error-detection mechanisms in each IP block if they are not required for a particular application.

2) Diagnosis: As with the general-purpose computing example, diagnosis in SoCs will be distributed across the system stack. The resilient IP interface will provide the cross-layer visibility required to allow higher layers in the stack to control and direct diagnosis.

3) Recovery: For recovery, the IP interface needs to guarantee data integrity between the IP blocks, as mentioned in the discussion of detection. It must also provide a protocol for retrying failed computations, such as a requirement that blocks buffer any requests they send to other blocks until completion of the request is acknowledged.

4) Reconfiguration and Adaptation: The resilient IP interface is also the key to reconfiguration and adaptation in our scheme. It should provide protocols to inform an IP block when a permanent error in the block has been detected and invoke any self-reconfiguration mechanisms the block may provide to handle the error. The interface must also provide mechanisms to disable and isolate defective IP blocks, allowing the system to continue (potentially with degraded performance) in spite of the defect, perhaps by performing the IP block's functions in software.

V. Conclusion

Increasing error, variation, and aging rates in semiconductor systems are making it more and more costly to tolerate all of the possible non-ideal device behaviors in one or two layers of the system stack. Distributing resilience and reliability across the system stack can improve performance and reduce power and area costs by taking advantage of the strengths of each layer and exploiting the characteristics of individual applications. We have described an approach to cross-layer resilient system design that divides resilience into five tasks: detection, diagnosis, reconfiguration, recovery, and adaptation.

To illustrate our model of cross-layer resilient design, we have presented a strawman design for a cross-layer resilient computing system and have shown how it could build on previous work. We have also presented a discussion of cross-layer resilient systems-on-chip, highlighting the key differences between general-purpose systems and SoCs as they affect resilience. It is our hope that these examples will both support our claims and motivate further research on cross-layer resilient design.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0637190 to the Computing Research Association. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Computing Research Association, the National Science Foundation, or the Intel Corporation. The authors would like to thank Subhasish Mitra for his many comments on early drafts of this paper.

References

[1] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2005, pp. 243–247.

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

[3] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, "Circuit failure prediction and its application to transistor aging," in *Proceedings of the 25th IEEE VLSI Test Symmposium*. IEEE Computer Society, 2007, pp. 277–286.

[4] A. DeHon, H. M. Quinn, and N. P. Carter, "Vision for cross-layer optimization to address the dual challenges of energy and reliability," in *Design and Test in Europe (DATE)*, 2010.

[5] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[6] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1, 1996, pp. 293–307 vol.1.

[7] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 25–36, 2000.

[8] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th annual ACM/IEEE International Symposium on Microarchitecture*. Austin, Texas: IEEE Computer Society, 2001, pp. 214–224.

[9] E. Rotenberg, "Ar-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. IEEE Computer Society, 1999.

[10] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 99–110, 2002.

[11] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of the 32nd annual ACM/IEEE International Symposium on Microarchitecture*. Haifa, Israel: IEEE Computer Society, 1999, pp. 196–207.

[12] N. Madan and R. Balasubramonian, "Power efficient approaches to redundant multithreading," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1066–1079, 2007.

[13] M. W. Rashid, E. J. Tan, M. C. Huang, and D. H. Albonesi, "Power-efficient error tolerance in chip multiprocessors," *Micro, IEEE*, vol. 25, no. 6, pp. 60–70, 2005.

[14] A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Trans. Comput.*, vol. 20, no. 11, pp. 1322–1331, 1971.

[15] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced CMOS bulk devices," in *Reliability Physics Symposium Proceedings, 2006. 44th Annual., IEEE International*, 2006, pp. 217–225.

[16] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 197–209.

[17] T. Austin, D. Blaauw, T. Mudge, and K. Flautner, "Making typical silicon matter with Razor," *Computer*, vol. 37, no. 3, pp. 57–65, 2004.

[18] K. Bowman, J. Tschanz, C. Wilkerson, S.-L. Lu, T. Karnik, V. De, and S. Borkar, "Circuit techniques for dynamic variation tolerance," in *Proceedings of the 46th Annual Design Automation Conference*. San Francisco, California: ACM, 2009, pp. 4–7.

[19] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The NBTI-aware processor," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 85–96.

[20] J. Blome, S. Feng, S. Gupta, and S. Mahlke, "Self-calibrating online wearout detection," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 109–122.

[21] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[22] N. Wang and S. Patel, "ReStore: Symptom based soft error detection in microprocessors," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2005, pp. 30–39.

[23] S. K. Sahoo, L. Man-Lap, P. Ramachandran, S. V. Adve, V. S. Adve, and Z. Yuanyuan, "Using likely program invariants to detect hardware errors," in *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN 2008)*, 2008, pp. 70–79.

[24] N. Oh, S. Mitra, and E. McCluskey, "ED^{4}I: error detection by diverse data and duplicated instructions," *Computers, IEEE Transactions on*, vol. 51, no. 2, pp. 180–199, Feb 2002.

[25] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 63–75, Mar 2002.

[26] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled fault tolerance," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 366–396, 2005.

[27] D. Lu, "Watchdog processors and structural integrity checking," *Computers, IEEE Transactions on*, vol. C-31, no. 7, pp. 681–685, July 1982.

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

[29] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, 2003, pp. 137–143.

[30] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, 2003, pp. 581–588.

[31] K. Pattabiraman, Z. Kalbarczyk, and R. K. Iyer, "Automated derivation of application-aware error detectors using static analysis," in *Proceedings of the 13th IEEE International On-Line Testing Symposium*. IEEE Computer Society, 2007, pp. 211–216.

[32] P. Franco and E. McCluskey, "On-line delay testing of digital circuits," in *VLSI Test Symposium, 1994. Proceedings., 12th IEEE*, Apr 1994, pp. 167–173.

[33] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, California, USA: ACM, 2006, pp. 73–82.

[34] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 97–108.

[35] S.-B. Park and S. Mitra, "IFRA: instruction footprint recording and analysis for post-silicon bug localization in processors," in *Proceedings of the 45th annual Design Automation Conference*. Anaheim, California: ACM, 2008, pp. 373–378.

[36] Y. Li, Y. M. Kim, E. Mintarno, D. S. Gardner, and S. Mitra, "Overcoming early-life failure and aging for robust systems," *Design & Test of Computers, IEEE*, vol. 26, no. 6, pp. 28–39, 2009.

[37] M. Zhang, S. Mitra, T. M. Mak, N. Seifert, N. J. Wang, Q. Shi, K. S. Kim, N. R. Shanbhag, and S. J. Patel, "Sequential element design with built-in soft error resilience," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 12, pp. 1368–1378, 2006.

[38] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," *SIGARCH Comput. Archit. News (ISCA 2008)*, vol. 36, no. 3, pp. 203–214, 2008.

[39] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester, "Vicis: a reliable network for unreliable silicon," in *Proceedings of the 46th Annual Design Automation Conference*. San Francisco, California: ACM, 2009, pp. 812–817.

[40] E. Schuchman and T. N. Vijaykumar, "Rescue: A microarchitecture for testability and defect tolerance," in *Proceedings of the 32nd annual International Symposium on Computer Architecture*. IEEE Computer Society, 2005, pp. 160–171.

[41] F. A. Bower, D. J. Sorin, and S. Ozev, "A mechanism for online diagnosis of hard faults in microprocessors," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Barcelona, Spain: IEEE Computer Society, 2005, pp. 197–208.

[42] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: improving lifetime chip performance for multicore processors in the presence of hard faults," in *Proceedings of the 2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2008, pp. 43–51.

[43] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee, "Architectural core salvaging in a multi-core processor for hard-error tolerance," in *Proceedings of the 36th annual International Symposium on Computer Architecture*. Austin, TX, USA: ACM, 2009, pp. 93–104.

[44] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith, "Configurable isolation: building high availability systems with commodity multi-core processors," *SIGARCH Comput. Archit. News (ISCA 2007)*, vol. 35, no. 2, pp. 470–481, 2007.

[45] K. Reick, P. Sanda, S. Swaney, J. Kellington, M. Mack, M. Floyd, and D. Henderson, "Fault-tolerant design of the IBM Power6 microprocessor," *Micro, IEEE*, vol. 28, no. 2, pp. 30–38, March-April 2008.

[46] A. Meixner and D. J. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN 2008)*, 2008, pp. 80–89.

[47] J. S. Planck, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under UNIX," in *Usenix*, New Orleans, LA, 1995, pp. 213–223.

[48] D. Hunt and P. Marinos, "General-purpose cache-aided rollback error recovery (CARER) technique," in *17th International Symposium on Fault-Tolerant Computing Systems*. IEEE CS Press, 1987, pp. 170–175.

[49] M. Prvulovic, Z. Zheng, and J. Torrellas, "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 111–122.

[50] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, 2002, pp. 123–134.

[51] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. Reno, Nevada: ACM, 2007, pp. 1–11.