System-level, Cross-layer Cooperation to Achieve Predictable Systems from Unpredictable Components

In coming years, a number of factors will lead to a significant shift in the way computer systems manage reliability, variation, and fabrication. Currently, computer systems assume perfect device fabrication and operation. For high-reliability systems, the usual methods of increasing system reliability involve ECC coding on memories and triple-modular-redundancy (TMR) of critical components. These brute force methods are able to increase system reliability when silicon fabrication processes are able to deliver high individual device reliability and low variation. However, as the critical dimensions of devices, such as transistors and wires, used to implement computer systems shrink to only a few nanometers, rates of transient faults, permanent faults, and variation between devices on the same die are expected to increase to the point where this approach will no longer be practical.

Instead, computer systems will need to adopt a model in which each layer in the abstraction hierarchy — applications, O/S, architecture, circuits — is prepared for the layer below to transmit bad data and in which all of the layers in the hierarchy cooperate to deliver correct operation in spite of faults, variations, and other effects.

This shift to a multi-level approach to resilience is further motivated by trends in fabrication processes where device manufacturing is increasingly limited by power consumption instead of device density, and by trends in computer architecture where designs with large numbers of independent execution resources, such as cores and reconfigurable units, are becoming more common. The need to decrease system power consumption makes it critical that schemes to tolerate errors and variation consume as little power as possible during error-free computation cycles. Diagnostics resources, error-correction facilities, and spare resources can sit idle until needed. Repair software in the O/S can be triggered to manage the repair, perhaps using distinct execution resources from those where the error was detected.

Architectures where the size of an independent execution resource is small compared to the total chip size offer two distinct opportunities that do not exist in current architectures:

1. We can use a few independent execution resources on diagnostics and reliability management without significantly impacting performance or energy.
2. We can spare entire execution resources.

Moving to the multi-level model of resilience that we envision will require substantial changes in the way engineers design and build systems. These changes reformulate the guarantees and responsibilities of each layer in the system stack and redefine the mechanisms by which each layer in the stack can communicate with the other layers to (1) inform them of its capabilities and needs, (2) provide information about the state of the system, and (3) respond to changes in system state. In this proposal, we outline a program of workshops that will crystallize this vision and identify the new science and engineering disciplines needed to overcome these challenges. The output of these workshops will be a draft call for proposals for a government-funded research program in the area of multi-level resilience, which we will present to program managers at the NSF and other agencies to solicit support for this critical research area.

1 Problem Statement

Our modern world is increasingly mediated and controlled by computing devices, including our critical infrastructure, our communications, our automobiles, our buildings, and our medical devices. This situation has lead to a global dependence on global positioning satellite systems to provide precision-navigation for commercial and military aircrafts, on high-availability, high-reliability computing systems for the financial and utility infrastructure, and on drive-by-wire technology to increase the safety of our transit systems.

Ubiquitous, inexpensive computation enables these high levels of sophistication and automation that enhance our lives and productivity. Much of the increasing computational power and decreasing costs that facilitate ubiquitous computation is driven by Moore's law, which leads manufacturers to aggressively shrink the feature size of transistors, storage nodes, and wires (collectively termed as *devices*). When these devices were composed from very large numbers of atoms and charge was stored in large numbers of electrons, the statistical effects governing the placement of individual atoms and electrons would normalize, making device behavior predictable and stable. However, as feature sizes scale to tens of atoms across, the statistical nature of individual atoms, dopants, and electrons is both sparse and unnormalized, leading to elements that vary widely across an entire integrated circuit and that are more susceptible to contaminants. These situations have given rise to several distinct effects including:

- transient errors from radiation [6, 4, 16] and noise fluctuations [7, 14, 13, 8],
- process variation and permanent defects from manufacturing [3, 1],
- compound reliability problems caused by the presence of temperature and power fluctuations on top of other reliability problems, and
- fragile devices with shortened lifetimes [3, 1].

Many of these problems are already affecting critical infrastructure, as observed in neutron upsets to avionics [10] and upsets in unprotected caches. Since the reliability of many of these computing systems is already below an acceptable level, further decreases in the reliability due to scaling issues cannot be feasibly withstood.

Because we have traditionally been able to address these problems at the manufacturing and device level, microarchitecture, architecture, system and application designers have been able to reasonably expect components that are free of defects and transient failures. As a result, only systems for harsh environments and life-critical systems have required higher-level designers to pay attention to failure. Since these applications have traditionally been associated with small markets where the computational component only makes up a small portion of the system cost, these systems have been able to employ brute-force, high-overhead reliability techniques, such as triple modular redundancy.

However, as computational control moves into the mainstream, significant fractions of the cost of products migrate into computerized automation. Combined with the fact that manufacturing can no longer hide device-level variation and noise effects, efficient mitigation of these effects requires attention from layers at the circuit, microarchitecture, architecture, system, and application levels of the system stack. Designers at these higher levels must expect to operate in a new world where:

- Fabrication will be imperfect and some portion of the computational and memory devices on a chip will be unusable.
- Lifetime variability of parameters and usability will cause individual devices to wear out in less time than the desired operational lifetime of the component.

- Conservative margining against variation will be too costly in terms of delay or energy.
- Transient errors in computational and memory devices will make reliable computation impossible for **all** ICs and systems without mitigation, even high-volume consumer products.

With these expectations, large components and systems will need to tolerate defective or widely varying devices and faults while maintaining high-reliability guarantees. Furthermore, all designs must be jointly optimized in an energy-area-delay-reliability space so that the overhead to mitigate the increased noise effects does not negate the improvements in density, delay, and energy offered by scaling.

2 Motivational multi-level example: DRAM

Parts of today's computing systems do tolerate imperfect manufacturing and transient errors. Notably, both our communication links and our bulk storage (*e.g.*, DRAM, Flash, hard disks, compact discs) operate despite the fact that individual bits may be corrupted during transmission or storage. This mitigation does not incur integer factors of overhead through coordination and cooperation across different levels of the memory or communication system design.

For example, we have long accepted that DRAM cells may not yield and that even good DRAM cells could change their value (*e.g.* be upset by ionizing radiation). Consequently, modern memory systems employ a combination of the following techniques:

- differential reliability—to achieve the high density of tiny memory cells while maintaining high reliability and yield for non-repairable control and error-correction circuitry
- device level hardening of memory bits—to contain the failure rate of individual memory bits
- ability to map out bad memory regions—to avoid defective or unreliable regions of the chip (*e.g.* row and column sparing for defects)
- redundancy in the form of ECC—to allow efficient error detection and correction
- periodic scrubbing—to prevent error accumulation

If we insisted that we solve the memory reliability problem at the device level so that memory bits never fail, we would be forced to build much larger memory cells by employing larger feature sizes, more transistors, or dual interlocked storage cells that store millions of electrons to guard against almost all ionizing particles. Instead, we code the data at the next level in the hierarchy so we can detect and correct failures. A modern Hamming ECC for memory adds 8 bits to each 64-bit word. For an increased overhead of 12.5%, any two errors in a word are detected and any single error corrected. The result is a DRAM device that can provide a net reduction in area per application data bit.

Furthermore, memory errors accumulate over time. If we demanded that our error correction code handle all errors accumulated in each memory word over the months or years that an application is in operation, we would need a code with greater redundancy and greater overhead. Instead, we scrub the data periodically at higher levels in the system to prevent the accumulation of errors [11, 9]. Even if we had to perform one scrub cycle every 1000 real memory cycles, we would only spend 0.1% of our memory bandwidth on overhead for scrubbing. This is less overhead than changing to a larger ECC code (*e.g.* 82-bits to correct 2 errors in a 64-bit word). Optimization across layers allows a more efficient solution than solving the problem of memory error rates at any single level.

An important phenomenon this memory example illustrates is time scales and filtering. While device hardening and ECC effectively filter out errors at one rate, the system can afford longer

time operations, such as scrubbing, at higher levels. If the lower levels filter errors sufficiently, the higher level operations are invoked less frequently.

3 Vision

We can no longer assume computational elements will be perfectly and identically fabricated and operate without transient upsets. While, in the previous example, we were able to show how multi-level solutions have been useful in protected DRAMs, similar solutions for computation currently do not exist. In part, the heterogeneous design of computing systems and their ability to transform data makes posing simple solutions that do not rely on brute-force replication difficult. Nonetheless, there are numerous hints that many of the high-level ideas can transfer.

Hardware organizations must be prepared for repair. Both RAM and hard disks expect errors and employ microarchitectures and abstractions that allow repair. While RAM repair, such as row and column sparing, occurs below the architectural level and is invisible to the software, bad disk sectors in hard drives are visible to the operating system. In a similar manner, our computational organizations must be prepared for errors. Mitigation of the errors will likely require cooperation across the microarchitecture, architecture, and operating system.

Errors must be filtered at multiple levels. To use small devices, memory systems allow individual memory bits to fail. The microarchitecture assists by correcting errors during memory access. The operating system collaborates by scrubbing memory. To use small devices and low energy for computation, we must similarly expect occasional errors in the computation. These errors will need to be caught and corrected at higher levels in the system stack.

Multilevel trade-offs provide efficient solutions, generalizing the idea of hardware-software trade-offs. With errors slipping through devices, higher levels must be prepared to detect and correct them. Similar to the way we distribute the function of a memory system across hardware (*e.g.* TLB) and software (*e.g.* TLB and page miss handling), efficient solutions will carefully divide functionality between the microarchitecture and system software. Suitable architectural interfaces will be required and will benefit from compiler and application support.

Strategic redundancy improves solution efficiency. Information theory tells us how to provide shared redundancy across large blocks of data to avoid brute-force replication. Efficient computational solutions will similarly avoid brute-force replication. For example, invariants and end-to-end consistency checks on the computation may allow lightweight checks for errors.

Differential reliability enables more efficient solutions. DRAMs with ECC and row sparing carefully exploit the fact that the ECC allows the core of the memory to be less reliable than the periphery. This solution also exploits the ability to fabricate devices with different feature sizes to assure stronger reliability. Computations can similarly employ a mix of larger, more-reliable devices and smaller, less-reliable devices. Similarly, we can use higher voltages and currents to make some circuits more reliable than others. Thereby, computations that have efficient checks or are less sensitive to errors can be run on smaller, lower-energy devices. In this manner, high-level information about application invariants or requirements drives microarchitectural decisions around the deployment of circuits and devices with different characteristics.

Scalable solutions should allow adaptation to error rates and reliability. Scalability to different error rates and different levels of protection is not demonstrated by the simple, traditional DRAM example. However, information theory does tell us how to develop codes of different rates to handle different needs, and it is easy to see how to add adaptability for memory systems (See

Section 4.) With growing error rates and applications with differing needs for protection, we need the engineering understanding of how to best provide that protection across the design space as well as architectures and components that can be tuned in-system to varying environmental conditions. Device wear suggests error

rates will change over time in a single component, further driving the need for in-system adaptation

To sum up, just as a multi-level cache memory system attempts to provide the density of a large memory with the speed of small memory:

- A traditional, ECC-protected memory provides the reliability of large feature sizes with the density of small memory cells.
- Multi-level computational designs can provide the reliability of large-feature and large-energy devices with the density and energy consumption of small-feature, low-energy devices.

4 Adaptable Memory Systems

Multi-level solutions could be extended to make memory systems adaptive to changing environments and aging. High-level, periodic monitoring software can use low-level information, such as ECC correction rates, to assess the system's health in order to estimate upset rate

and system vulnerability. These checks may discover that (i) the system's mitigation methods are either too aggressive or not aggressive enough for the actual upset rate, (ii) specific modules or pages have excessively high upset rates, or (iii) specific words have new permanent failures. Based on this discovery, the high-level software can direct changes to the lower-level system configuration to reduce vulnerability. For example, the system can (i) change the scrubbing rate or change the ECC code rate to adapt to the upset rate, (ii) map out a particular memory page if a region of memory is experiencing an excessively high upset rate, or (iii) replace a newly defective row with a spare.

5 Resilient Computing Example

As an example of how multi-level reliability might be implemented in a computing device, consider the CPU of a 2018-era computer that contains a memory hierarchy using the approaches described earlier and a number of I/O devices. Assuming current CMOS trends continue, such a CPU might have a die size of 1–2 cm^2 and contain between sixteen and sixty-four cores of similar complexity to current CPUs, a larger number of finer-grained execution units (*e.g.* GPUs, FPGA-like cells), or a mix of execution unit granularities. This CPU might be designed with a peak clock rate of 6-8 GHz, although power and thermal limitations would prevent it from operating all of its cores at their peak clock rates except in short bursts. Instead, the operating system will dynamically adjust the supply voltages and clock frequencies of the cores, tuning their throughput to meet the demands of the operating environment and the characteristics of the applications being executed without overheating the CPU.

Such a chip would **prepare for repair** by incorporating hardware structures that allow cores, ALUs, memory blocks, or other functional units to be disabled and isolated from the rest of the chip if they develop permanent faults. Multi-level rollback support would be possible through instruction squashing at the microarchitectural level and hardware support for low-overhead checkpointing

for larger rollback windows at the OS- and application-level.

The CPU would also support **strategic redundancy** in critical modules. Register files and on-chip memories would employ ECC. The CPU would use **light-weight detection mechanisms** to catch errors in computations, such as residue arithmetic, parity bits on instruction words, operation sequence signatures, and a heartbeat timer to detect software "hangs." These mechanisms will detect and **filter out** the vast majority of transient errors and, in combination with test routines, will be used to diagnose permanent errors.

Further, the CPU will **cooperate across multiple levels** with the OS and applications to exploit **differential reliability** to make reliability/performance/power consumption trade-offs according to the needs of the application. By default, the OS might enable the CPU's error detection and checkpointing hardware, assuming that the application does not contain any self checks, but is not critical enough to merit redundant execution of each operation. Applications that can tolerate some errors in their outputs, such as video playback, could inform the OS of this fact, allowing it to disable some of the reliability mechanisms to reduce power consumption. Applications that embed their own **lightweight checks**, such as ILP solvers that check the validity of the solution found, could disable most or all of the error-checking hardware except during consistency checks, while functional applications that do not modify their inputs or external state might choose to disable checkpointing and re-execute from the beginning if an error occurs. Conversely, applications that contain critical regions could request that the OS turn on redundant execution during those regions. For example, an on-line banking application might run with "normal" reliability settings while the user is checking balances and reviewing past statements, but turn on redundant execution during a balance transfer transaction.

When the processor, OS, or application detect an error in a computation, they will use **multi-level schemes** to diagnose and correct the error. The hardware records that an error has occurred, either by incrementing a hardware counter or signaling an exception. The faulty operation is reissued. If the operation completes correctly the second time, the faulty operation is logged as being an unrepeatable error. If the operation fails twice in a row or too many times in a particular window of time, the processor assumes that a permanent error has occurred and signals an exception to the operating system. Finally, the operating system responds by performing a local diagnosis and repair, migrating the computation to another core, reverting to its last checkpointed state, and/or terminating the application. At higher levels, the application can report suspicion of an error to the OS and request rollback and re-execution. The OS notices when the application is making re-execution requests too frequently. This, too, can trigger diagnosis, repair, and migration.

In addition to these reactive diagnoses and migration, the OS will periodically migrate computations off of each core and invoke test routines on the core to detect permanent errors. During these tests, the OS may deliberately overstress the core by running it at accelerated clock rates in order to determine whether aging effects are decreasing the core's performance to the point where it can no longer run safely at a reasonable level of performance or energy consumption. The results of these tests, as well as tests performed when the processor is fabricated, will be stored by the OS and used when making decisions about the operational clock frequency for each core and the assignment of tasks to cores.

The hardware and the OS will **adapt** to the error rates seen in the system. The CPU might include selectable ECC logic, allowing the system to configure different amounts of protection against soft errors depending on the needs of the application. Similarly, the CPU might add logic that allows groups of cores to check each other's results when running critical computations that

demand particularly high reliability. The operating system will monitor the error rate logs, calculate the reliability of the system, and reconfigure the hardware to achieve a desired level of resilience. A system located at sea level might well be able to operate with fewer reliability mechanisms enabled than one located in Denver, Colorado, due to the greater rates of radiation-induced soft errors at Denver's altitude, for example. Systems that experience substantial variation in error rates, such as airplane or spacecraft control systems, could extend this capability by incorporating radiation detectors that allow them to respond more quickly to changes in soft error rates, avoiding the need to operate under worst-case assumptions at all times.

In-field **adaptation** can also efficiently accommodate aging. As a part ages, it may exhibit higher fault rates demanding more frequent checkpoints and more heavy use of resilience mechanisms; since each component ages differently, we do not penalize the robust components for a fraction that age more quickly.

Overall, distributing reliability across the system stack provides both dependability and flexibility by allowing the system to tune itself based on its needs.

6 Open Questions

An important goal of the study is to develop community consensus on the key questions that need to be addressed in this area (*i.e.*, can we agree upon the set of questions that, if answered, would make a significant difference?). These questions should provide guidance for research and priorities for funding. Following are our initial attempts to formulate the questions. The broad workshops and discussions in the study will allow us to socialize, refine, focus, formulate, and prioritize these questions as well as identify any additional, high-priority questions.

1. How do we best accommodate repair? (granularity, division of responsibility among layers, visible interfaces)

2. What is the right level of filtering at each level of the hierarchy? (how do we characterize and assess? how do we tune and validate?)

3. How do we organize, manage, and analyze layering for cooperative fault mitigation?

   (a) What should new contracts and interfaces look like?
   (b) What information is useful to reflect up the stack?
   (c) What controls on lower levels should be exposed and how?
   (d) What information is it useful for higher-levels to pass down?
   (e) How do we evaluate and compose techniques across levels?
   (f) How do we engineer and analyze adaptation and repair control loops across layers?

4. Can we establish a useful theory and collection of design patterns for lightweight checking?

   (a) What computational classes are lightweight checkable (*i.e.*, for what class of computations is it provably (asymptotically or a constant-factor) less expensive to check (deterministically or probabilistically) the solution than to compute it? can we enlarge this class by *extending* the output of the base computation so its results include a certificate to assist checking? (*e.g.* Extended GCD [2]))

(b) How do we express checks in computation and optimize their use?

5. What would a theory and framework for expressing and reasoning about differential reliability look like?

   (a) How do we express (identify, analyze) and exploit allowable noise (error rates) from substrate for computation or piece of a computation?
   (b) What is the value of reflecting/exposing errors to the application level and what is the proper way to do so?
   (c) How do we express the reliability needs of an application, or how can we analyze the needs of subcomponents based on structure of the application?

6. Can a scalable theory and architectures that will allow adaptation to various upset rates and system reliability targets be developed? The theory side might identify the minimum achievable energy-delay-area-reliability surface as a function of upset rate; the architecture side would develop systems that are efficiently scalable to different upset rates and reliability targets. Assessment of an architecture may include: (i) How close can it come to the theoretical bounds? and (ii) Over what range of upsets and reliability targets does it scale?

7 Societal Impact

Economic: The growth of our economy and well being has been fueled by continually cheaper and more powerful computations that enable greater automation and new services and products. This, in turn, has been fueled by Moore's Law scaling. Integrating reliability and variation management into designs is essential to allowing us to continue to extract size, cost, and energy benefits from scaled computations.

Energy: Energy consumption promises to be a limitation to our capabilities and our economy. Operating in a regime where small numbers of failures can be tolerated can allow significant reductions in the amount of energy consumed by a computation.

Ultra-reliable Systems: Computation increases our infrastructural capabilities and our efficient use of scarce resources. Unless we systematically address reliability issues, these systems will be hit by the double-whammy of increasing device count and decreasing device reliability.

Harsh Environments: Automated computations expand our reach and survivability into harsh environments, such as space, high altitude, or extreme temperatures. However, these environments increase the upset and wear rates for devices, an effect which is further magnified as devices scale down in size. To provide reliable computations in these critical roles in harsh environments, we must be able to scale our reliability solutions to these more extreme environmental characteristics. We must further be able to do so with modest incremental effort on top of mainstream designs.

Security: As more of our interactions are managed and enhanced by computer mediation, it becomes increasingly critical that these systems be robust against deliberate subversion attempts. It is a difficult task to guarantee that a system cannot be penetrated even when we assume the devices and components work perfectly. Misbehaving devices violate key assumptions and create a myriad of new attack vectors against our systems. For example, researchers have already identified ways in which soft errors can be used to defeat cryptographic systems [15, 12] and software isolation layers [5].

Education: Flexibility and adaptability are one of the strengths of computing systems. Successful solutions are regularly deployed into uses beyond those originally envisioned by the designer. Between both the large fraction of computing systems that are employed in critical roles and the potential for almost any system to be deployed into such a role, noise-tolerance and reliability management is a concern for all computer engineers; we must make sure that computer engineering education evolves rapidly to prepare engineers for the new reality.

8 Proposed Activities

The goal of this proposal is to gather a community of widely-varied computer researchers to nurture a vision for a multi-level approach to reliability. We will facilitate community crystallization and refinement of the vision, generating a clear picture of the challenges and opportunities offered in multi-level reliability approaches. To foster this community we pursue a two-tiered approach:

1. Three focused workshops of experts aimed at discussing and revising the questions that we have presented and initiating cross-layer discussions.

2. A wiki to allow broad participation in between workshops for attendees and for the larger computer research field.

By the end of the proposed effort we believe the community will sufficiently converge on the vision that we can draft a call for proposals for a government-funded research program in the area of multi-level resilience that we will present to program managers at the NSF and other agencies to solicit support for this critical research area.

The first of the three workshops will focus on a discussion of the open questions as presented in Section 6. During this workshop, we will present the vision for the multi-layer approach to reliability and identify the key open questions, using our list as a starting point for discussion and organization. We expect the broad participation and discussion to generate additional candidate questions and refine the existing questions.

The breadth of topics makes it difficult to do justice to all of these issues in a single workshop. To that end, the second and third workshops will be organized around distinct themes. The working themes are:

- **Workshop 2:** exposure and cooperation at the level of the application, algorithm, programming language, and compiler (Q4, Q5)

- **Workshop 3:** cross-layer information sharing and cross-layer optimization (Q1, Q2, Q3)

The core working group will tune the theme and content of these later workshops as necessary to properly address emergent issues, concerns, and priorities that become apparent in the first workshop.

Three meetings also provide location and time diversity. All of the likely participants have busy schedules and it will certainly be impossible to pick a single date when everyone can attend.

After the final workshop, we will draft the call for proposal with our core working group. The primary output for this project is a draft solicitation for a program including:

- Identification of key questions to be addressed.

- Guidance for key characteristics of research in this program (*e.g.* program-specific review criteria)

Additional outputs will include Wikis from the visioning process and digest summaries of the workshops.

9 Participation

We will use the Wiki to promote broader participation. To make sure we get the word out to the right people in each community and know whom to invite, we are assembling a core working group of 8–10 individuals with representation across the communities and industries. The core working group will be involved in suggesting and recruiting participants for the early workshops, reviewing Wiki submissions, selecting invited speakers for the workshops that come out of the Wiki discussion, and reviewing and refining the output reports.

Person	**Affiliation**	**Expertise**
Sarita Adve	UIUC	Architectures and System Softwware
Todd Austin	University of Michigan	Architecture
Andrew Huang	Chumby Industries, Inc.	System Engineer
Ravi Iyer	UIUC	Reliable Distributed Systems
Subhasish Mitra	Stanford University	Fault Tolerance, Testing, Circuits, Architecture
Sani Nassif	IBM	CAD and Technology
John Savage	Brown University	Information and Computational Theory
David Walker	Princeton University	Programming Languages
Gary Swift	Xilinx	Device Physics, Radiation Effects

10 Timeline

Date	**Event**
Nov. 2008	Core working group assembled
Jan. 2009	Wiki up, start soliciting contributions
Feb. 2009	Vision/Questions workshop (now: post-SELSE March 26-27)
Apr. 2009	Second workshop (high-level exposure) (perhaps move to June)
Jul. 2009	Third workshop (cross-layer)
Oct. 2009	Final report

11 Organizers

Person	**Affiliation**	**Expertise**
André DeHon	University of Pennsylvania	Architecture, CAD, Technology Impact
Heather Quinn	Los Alamos National Laboratory	Radiation Effects, Fault-Tolerance, Runtime
Nicholas Carter	Intel Corporation	Architecture, Technology Impact

References

[1] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4/5):433–449, July/September 2006.

[2] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[3] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November–December 2005.

[4] O. Flament, J. Baggio, C. D'hose, G. Gasiot, and J.L. Leray. 14 MeV neutron-induced SEU in SRAM devices. *IEEE transactions on nuclear science*, 51(5):2908 – 2911, 2004.

[5] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[6] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *Proceedings of Symposium on VLSI*, pages 73–74, 2001.

[7] J. Kim and L. Kish. Error rate in current-controlled logic processors with shot noise. *Fluctuation and Noise Letters*, 4(1):83–86, 2004.

[8] Laszlo B. Kish. End of Moore's law: Thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305:144–149, 2002.

[9] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing*, pages 37–42, 2004.

[10] E Normand. Single event effects in avionics. *Transactions on Nuclear Science*, 43(2):461–474, 1996.

[11] Abdallah Saleh, Juan Serrano, and Janak Patel. Reliability of scrubbing recovery-techniques for memory systems. *IEEE Transactions on Reliability*, 39(1):114–122, 1990.

[12] Adi Shamir. Research announcement: Microprocessor bugs can be security disasters. Available online at `http://cryptome.org/bug-attack.htm`, November 2007.

[13] Karl-Ulrich Stein. Noise-induced error rates as limiting factor for energy per operation in digital ic's. *IEEE Journal of Solid State Circuits*, 12(5):527–530, October 1977.

[14] J. A. Swanson. Physical versus logical coupling in memory systems. *IBM Journal of Research and Development*, 4(3):305–310, July 1960.

[15] Jun Xu, Shuo Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 421–432, 2001.

[16] James Ziegler and Helmut Puchner. *SER – History, Trends and Challenges: A guide for designing with Memory ICs*. Cypress, 2004.