Online Error Detection and Correction of Erratic Bits in Register Files

X. Vera, J. Abella, J. Carretero, P. Chaparro, A. González
Intel Barcelona Research Center
Intel Labs - Universitat Politècnica de Catalunya
Barcelona, Spain
{xavier.vera, jaume.abella, javierx.carretero.casado, pedro.chaparro.monferrer,antonio.gonzalez}@intel.com

***Abstract*— Aggressive voltage scaling needed for low power in each new process generation causes large deviations in the threshold voltage of minimally sized devices of the 6T SRAM cell. Gate oxide scaling can cause large transient gate leakage (a trap in the gate oxide), which is known as the *erratic bits* phenomena.**

Register file protection is necessary to prevent errors from quickly spreading to different parts of the system, which may cause applications to crash or silent data corruption. This paper proposes a simple and cost-effective mechanism that increases the resiliency of the register files to *erratic bits*. Our mechanism detects those registers that have *erratic bits*, recovers from the error and quarantines the faulty register. After the quarantine period, it is able to detect whether they are fully operational with low overhead.

I. Introduction

Scaling of CMOS minimum feature size continues to enable Moore's law trend in integration densities [1]. With every new process technology, random dopant fluctuations phenomena may result in larger deviations in the threshold voltage of minimally sized devices of SRAM cells. This mismatch may reduce available cell noise margin, which results in unstable cells. Erratic bit phenomena have been already reported in advanced flash memories [2], and have been attributed to trapping/detrapping effects that modify the threshold voltage. Recently, it has been reported the observance of erratic behavior in SRAM for 90nm [3].

Maintaining adequate cell noise margin is becoming even more challenging for modern processors, which require lower operating voltage (*Vcc*) due to electric field density and power limitations. Although technology scaling reduces the power consumption per transistor, it increases the power density per area unit. Therefore, assuming constant chip area [1], the power demand of a processor increases much faster than the power envelop, which is not expected to grow substantially due to cooling solution costs. Similarly, scaling transistors and keeping *Vcc* constant increases the electric field density that transistors experience, leading to higher stress and faster degradation. In consequence, *Vcc* must be decreased in each new process generation. Some important market segments such as embedded processors and laptops have further *Vcc* constraints to extend battery life. Those processors make an aggressive use of Dynamic Voltage and Frequency Scaling (DVFS) techniques to adapt their *Vcc* and frequency to the current workload and battery state [4], [5].

Whereas the erratic *erratic bits* phenomenon can be eliminated for 90nm SRAMs by process optimization, *erratic bits* behavior gets worse with smaller cell sizes. Making things worse, dimensions and operating voltages of transistors have been shrinking constantly, which has increased their sensitivity against radiation phenomena. Therefore, a single radiation such as alpha particles released by radioactive impurities and neutrons coming from outer space can cause a transient error [6].

In this work we will focus on protecting the register file. Register files are accessed very frequently, which increases the probability of errors to propagate to the output of the program. Thus, protecting them is critically important.

Some general methods to tolerate transient and permanent faults due to soft errors or fabrication defects have been deployed in the past [7]–[9]. Although they are not especially suited for low *Vcc* related failures, they may help to tolerate them to some extent. Some of such methods are based on deactivating faulty blocks [8], which could disable too many registers in the register file. Some other methods are based on the concept of redundancy such as triple modular redundancy (TMR) [9]. TMR cost is unaffordable for most of the scenarios because tripling the register file requires roughly triple area and power. Some information redundancy is also provided with error detection and/or correction schemes such as parity and ECC [7]. Parity or ECC suffice to detect and/or correct infrequent errors, but they are not suitable for a large number of permanent errors. In general, registers with faulty bits must be disabled permanently, which may decrease the number of available registers drastically. Therefore, register file yield rapidly decreases. A low register file yield translates into significant performance loss or even a useless core because it has fewer registers than required to operate. Thus, new techniques to tolerate moderate and high error rates due to permanent faulty bits at the expense of low overhead are required.

Low-latency storage structures such as register file, latches and some first level caches (DL0, IL0, DTLB, ITLB) do not employ ECC due to its huge cost in area and delay, and use parity instead (like Intel® Montecito™ [10]). Parity allows detecting a single error, but once the error is detected, correction is only possible if the instruction that produced the corrupted value has not left the pipeline. Over-estimation of the *erratic bits* problem can result in over-design of the protection mechanisms, which will eventually increase the reliability cost. On the other hand, insufficient protection of register files will make the system unreliable and therefore useless. Hence, a tradeoff between reliability and cost must be achieved.

This paper proposes a simple and cost-effective mechanism that increases the resiliency of the register files to the *erratic bits* problem. Our proposal recovers from *erratic bits* in the register file; whenever an error is detected, our mechanism figures out whether it is an *erratic bits* problem or a soft error, identifies the erratic bit and recovers from the error without requiring further information. Later, the entry affected by the *erratic bits* problem can be disabled to prevent future errors.

As a result, register files (or other low-latency storage structures) protected only with error detection techniques can recover from the *erratic bits* problem and thus, higher reliability is achieved. Moreover, significant energy savings can be achieved because the *Vcc* can be further reduced, even if intermittent errors appear more often, because our mechanism allows full recovery.

The rest of the paper is organized as follows. Section II reviews the effect of low *Vcc* on the faulty bit rates in register files. Section III introduces our mechanism to detect and correct *erratic bits* caused by low *Vcc*. Section IV highlights some related work. Finally, Section V summarizes the main conclusions of this work.

II. *Vcc* Impact on the Register File

Faults experienced by semiconductor devices are likely to increase due to shrinking geometries and lower power voltages. Intermittent faults occur due to unstable or marginal hardware, and are usually observed under some particular environmental conditions (e.g., high temperatures). In this section, we give an overview of the *erratic bits* problem.

For conventional register files, the *Vcc* allowed during active operation is dictated primarily by read/write margins of the worst SRAM cells. *Vcc* in standby modes needing fast reactivation is set mainly by the minimum voltage required for data retention and soft error rate. In order to meet energy efficiency goals, designers have to use a relatively large SRAM cells to achieve the low *Vcc* values.

Gate oxide scaling can cause large transient gate leakage, which is inevitable due to physics and is likely to worsen with technology shrinking and voltage scaling. Published data [2], [3] shows no correlation between pre and post burn-in units. Recent studies have shown that the faultiness of each bit is independent of its location and the faultiness of its neighbor bits [3], [11]. Thus, the exact number of faulty bits and their location is highly unpredictable given that the fault probability of each bit is roughly the same.

As a consequence, the faulty bit distribution is basically random. Similar observations apply to those bits that become faulty in the field due to degradation, since their location is also random. The minimum *Vcc* at array/chip level is determined to tail the bits that display an erratic behavior; however, since BI/Stress causes the *redistribution* of erratic bits, the conventional guardbanding strategy do not longer work.

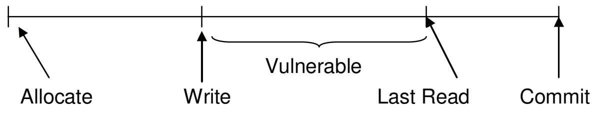

Fig. 1. Example of situation where parity detects and corrects an error by flushing the pipeline

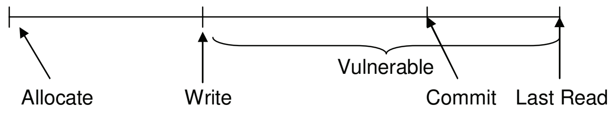

Fig. 2. Example of situation where parity can detects but not correct an error

Erratic bits are also *intermittent* errors, which manifest as stuck-at bits during a period in the order of *micro-* or *mili-*seconds. Normal bits can become erratic, while previously stable bits can show fluctuations. The data obtained from faulty bits is basically random. A faulty bitcell may be unable to flip its contents fast enough at write time. This way, its contents are not updated properly sometimes. Similarly, some bitcells may fail to provide their contents in time when they are read because they cannot feed bitlines properly. Finally, a faulty bitcell may flip its contents if process variations spoil its data retention capabilities. In any case, the value obtained from such a faulty bitcell can be different from what was written in the cell and can change easily in subsequent read operations.

Whereas permanent faulty bits can be easily identified by means of March tests [12] at boot time or during operation, similarly to the case of Intel®Pellston™technology [10], such approaches are of little help to solve the *erratic bits* problem due to their intermittent nature. Therefore, an online mechanism that detects faulty bits is required.

A. *Vulnerability of the Register File*

Parity in the register file allows detecting a single bit error, but once the error is detected, correction is only possible if the instruction producing the value corrupted has not left the pipeline. Let us consider the situation shown in Fig. 1. If the error is detected when the value is read, flushing the pipeline and re-executing from the oldest instruction would correct the error, since we would be able to re-generate the wrong value. Notice that this situation is more likely in out-of-order processors, where there is usually an important slack between the value is written back to the register file, and the instruction commits. For in-order processors, this slack is very small, and in many designs, when instructions write-back, they leave the pipeline.

However, for those cases when a register is read *after* the producer leaves the pipeline (see Fig. 2), the re-execution of the instruction that generated the value is not possible, and

TABLE I

BASELINE PROCESSOR

Parameter	Value
Memory	45ns latency
UL2	4 MB, 16-way, 12 cycle hit, 1 R/W port
DL1	32KB, 8-way, 3 cycle hit, 1 read + 1 write port
I-Cache	32KB, 8-way, 3 cycle hit, 1 read + 1 write port
DTLB/ITLB	128 entries, 8-way
ROB/MOB	128/30 loads, 22 stores
Register File	128 Int, 128 FP
Issue Queue (IQ)	32 entries, 6 issue, up to 3 Ints, up to 2 FP
Integer Units	3 ALUs, 2 AGU, 1 multiplier
FP Units	1 adder, 1 multiplier

TABLE II

WORKLOADS

Benchmark suite	#traces	Desc./Examples
Encoder	62	Audio/video encoding
SPECfp	41	Spec Fp 2K
SPECint	35	Spec Int 2K
Kernels	52	VectorAdd, FIRs
Multimedia	85	WMedia, photoshop
Office	75	Excel, word, powerpoint
Productivity	45	Internet contents creation
Server	53	TPC-C
Workstation	49	CAD, rendering

therefore, recovery is not possible unless we have some kind of checkpoint which is expensive in general.

B. *Parity Protection*

We have evaluated how often parity allows us detecting and recovering from errors in the register file. We have measured the coverage it provides for single-bit upsets for a processor that resembles an Intel® Core^{TM} Micro-Architecture (see Table I). This is an advanced out-of-order processor that was designed for efficiency and optimized performance across different market segments. Since the vulnerability of a processor depends on the dynamic behavior of the processor (and thus, the benchmarks run), we opt to run more than 500 traces representing all kind of benchmarks. They are detailed in Table II, and include kernels, Office, multimedia applications and SPEC programs. The experiment consists in measuring the AVF [13] of all the different registers: for each register, we measure how many values are consumed before the producer leaves the pipeline. We plot in Fig. 3 the s-curve. Results show that employing only parity makes 90% of the benchmarks have less than 50% possibilities to recover from a single-bit upset. Moreover, the average coverage (i.e., errors detected and corrected) for parity is $\mu = 28.6\%$ ($\sigma = 15.1\%$). Therefore, we need a simple mechanism to identify and recover from intermittent errors caused by *erratic bits* reducing those cases where errors detected by parity could not be recovered.

III. PROTECTING THE REGISTER FILE

Next, we describe a simple and cost-effective mechanism that increases the resistance of the register file to *erratic bits*.

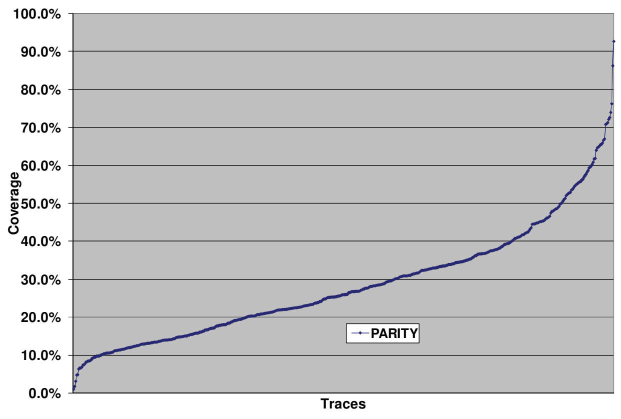

Fig. 3. Coverage for parity

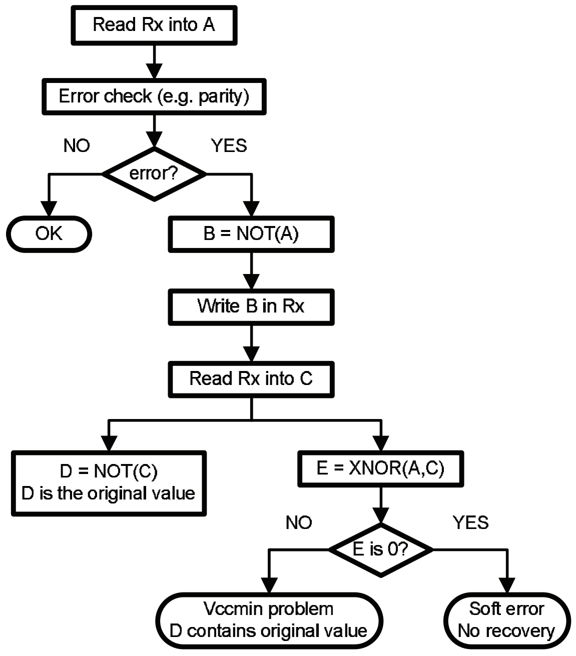

Fig. 4. Flow diagram to detect and recover from the *erratic bits* problem

A. *Identifying and Recovering from* erratic bits

In the rest of the explanation we will consider a register file protected with parity (or any other error detection mechanism) to explain how our mechanism is implemented. Notice that our methodology can also be applied to other structures protected with other error detection techniques. For instance, we can use our mechanism to identify and recover from intermittent errors in copy-back caches.

Our mechanism exploits the fact that intermittent errors due to the *erratic bits* problem make some bits to stick at "0" or "1" for some time: it does not matter which value we write into that bit, we will always read the same value.

Fig. 4 outlines the algorithm employed. All values are stored with their corresponding error detection codes (i.e., parity bit),

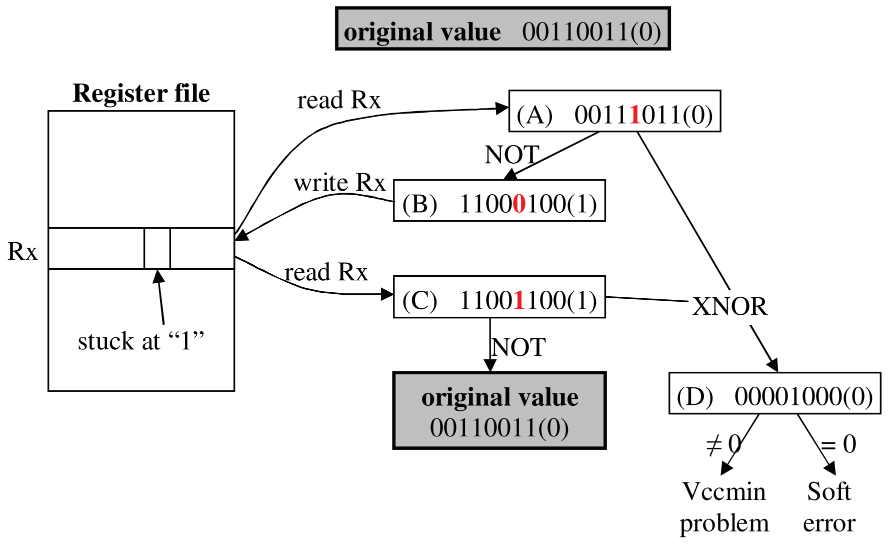

Fig. 5. Example of recovery from an intermittent error due to the Vccmin problem using only parity to detect the error

which are treated as part of the value (they are not recomputed at any point). The error correction starts when parity detects an error. In that case, we need to find out if the error is caused by the *erratic bits* problem, and correct it. This is very easy to implement if we make use of the property that *erratic bits* behave like stuck-at bits. If we invert the read register (which is faulty), and write it back again, we would have the original (and correct value) inverted. Therefore, we have corrected the error and can continue with the execution.

Example. For illustration purposes, we run the example in Fig. 5 to explain in detail our solution:

1) Let us assume that the original value is `00110011` and its parity bit is `0`. Parity checking detects an error when reading a register (Rx), which has the fifth bit stuck at "1" due to the *erratic bits* problem. We start reading the defective value and its parity bit, `00111011(0)`, and store it in latch A. We want to highlight that A differs from the original (correct) value in a single bit (the one stuck at "0" or "1"). Such bit can be even the parity bit.
2) We invert A and store it in B: as a result, we obtain `11000100(1)`. Notice that B is the inverse of the original value in all bits but one (the one with the *erratic bits* problem).
3) The inverted value (B) is stored in Rx. Since the faulty bit behaves like a stuck-at bit, when we write-back, the only bit that was not the inverse of the original value in B is flipped because such bit is stuck at "1" in Rx. Thus, Rx contains the original value inverted, `11001100(1)`.
4) Rx is read and stored in C.
5) If the error detected by parity were caused by the *erratic bits* problem, inverting C would give us the correct value. We make sure that the error is an *erratic bit* by XNORing A and C and storing the result in D. The result obtained in our example is `00001000(0)`. Notice that the faulty bit is exactly in the position where the bit is "1".
6) If it is an intermittent error caused by *erratic bits*, we only need to invert C to recover.

However, soft errors may also flip some bits. Let us see how our mechanism would work:

1) Let us assume that the original value is `00110011` and its parity bit is `0`. Parity checking detects an error when reading a register (Rx), which has the fifth bit flipped to "1" due to a particle strike. We start reading the defective value and its parity bit, `00111011(0)`, and store it in latch A.
2) We invert A and store it in B: as a result, we obtain `11000100(1)`. Similar to the previous case, B is the inverse of the original value in all bits but one (the one with the particle strike problem).
3) The inverted value (B) is stored in Rx. Now, Rx works fine; therefore, when we write-back, we do not obtain the original value inverted, rather `1100`**0**`100(1)`.
4) Rx is read and stored in C.
5) XNORing A and C yields a `0`. Therefore, we conclude that the error was caused by a soft error.

The *erratic bits* problem is very unlikely to affect several bits (orders of magnitude lower probability than single bit). In any case, the problem resides in the detection part, since our mechanism would be able to recover the original value.

Notice that if the erratic behavior disappears before we store the inverted value, our mechanism will consider the error as a soft error, and it will not be able to correct it.

B. Unprotected Register Files

For those implementations where the register file is unprotected, our mechanism still works, although at a higher cost in terms of performance and with some extra hardware changes. Basically, for every register access, we would need to make sure that the operation is correct. If we do not want to stall the processor on every read access, we would require specific hardware for inverting, latching and XNORing for *every* potential read (i.e., every read port), and either some extra write ports, or some logic to share the write ports, giving priority to the validation. Notice that this would cause some performance loss due to stalled instructions when write ports are not enough.

C. Disabling Entries Affected by Vccmin

The entry with the *erratic bits* problem (register Rx in our example described in Fig. 5) must be disabled temporally to prevent further errors. We have two different options, depending on the processor architecture.

1) Physical Register File: Once a register is detected to suffer the *erratic bits* problem, we take it out from the pool of available registers. This is achieved as follows:

1) We have a list (*vccmin_list*) similar to the regular free list where we place registers affected by the *erratic bits* problem.
2) The register that has the *erratic bits* problem is remapped (only if needed) and transferred to the *vccmin_list*. The easiest way of remapping would be: (i) flush, (ii) change the RAT if necessary with a new allocated register and move the contents of the register, and (iii) re-executing from the oldest instruction.

One of the main properties of the *erratic bits* problem is that it is intermittent; this is, after some time (order of *micro-* or *mili*-seconds) the problem may disappear and the register may be fully operational again. Therefore, we do not want to remove the registers as if they had a permanent error. Rather, we need to check whether those registers in the *vccmin_list* are free of faulty bits and bring them back to the free list. We propose a simple option which consists in every `T` cycles (in the order of millions), move all registers in the *vccmin_list* to the free list. If the *erratic bits* problem still exists, faulty registers will be removed again.

2) Architectural Register File: Organizations like in-order cores or out-of-order cores that keep speculative values in the reorder buffer, use an architectural register file. In those cases, we would need some spare registers, in such a way that the malfunctioning one are replaced by new ones.

The spare registers can be in a different register file, and will be accessed by the lower bits of the tag (like a cache). Each register will be only in one of the register files; in order to know where, we employ:

- A bit vector that indicates for each register whether it is in the normal register file or the spare register file
- A bit added to each register tag indicating in which bank it is stored so reads/writes can be done fast.

When a malfunctioning register is detected, we first identify which spare register would use; if it is empty, it starts using it and sets the corresponding bit in the bit vector. If it is in use, the value is "evicted" to the normal register file, it starts using it and the bit vector is updated accordingly. Notice that detecting whether the spare register is in use is as simple as ORing those entries of the bit vector corresponding to registers that would map in such spare register.

Similar to the case of the physical register file, in order to find out if registers work, we enable them, and check for potential errors at runtime.

D. Overhead

The cost of the mechanism is very low in terms of performance and hardware. Whenever the error is detected, which will happen rarely, the processor is stalled and few simple operations are required. Such operations can be implemented either as microcode to use existing hardware or with specific hardware (i.e., logic for inverting and XNORing, and few latches). Restoring the value and writing it back to an entry requires few steps. For instance, in the case of the register file the commit stage must stall and wait for all instructions in the pipeline to stop. Then, the value recovered must be stored in a free register, the rename table updated and the pipeline flushed to resume execution.

IV. Related Work

Redundancy is a widely used technique to recover from transient faults in a processor. Replicating register values into unused registers to recover from transient faults and soft errors was proposed in [14] where if ECC signals an error, correct value is taken from the uncorrupted register that holds the copy. Recently Reis et al. proposed using hardware-software hybrid schemes which achieves fault tolerance by replicating instructions at compiler level and using hardware fault detectors that make use of this redundancy [15], [16].

Replicating parts of the core has also been explored. DIVA [17] uses a simple in-order core as a checker for an out-of-order core. It has to design the checker from scratch since the objective is to catch design errors of a complex design that cannot be verified at design time. The IBM G5 [18] replicates the frontend and the execution engine, and all instructions are executed twice in parallel. By comparing the output of the instructions, it detects errors. In order to recover from errors, it keeps a copy of the register file.

Multithreading is used for error detection and recovery [19]–[21]. The general idea is to use the multithreading capabilities existing in modern SMT processors to run two copies of the same thread and after execution check the outcome of the instructions to detect the errors and recover from them if it is possible.

The inherent hardware redundancy in CMPs has been also used for error detection and correction [22]. A detailed efficient implementation of CMPs executing redundant threads with recovery capabilities has been described in [23].

Recently, Montesinos et al. [24] have proposed to use a small ECC table to protect the most vulnerable register versions in the register file against soft errors. Compared to our work, we do not limit the total number of registers to be protected. Moreover, they do not deal with the *erratic bits* problem.

V. Conclusions

Technology scaling and aggressive voltage scaling results in large deviations of the voltage threshold for SRAM. The observable impact is the erratic behavior of many bits. Moreover, due to their intermittent nature, classic offline testing techniques do not longer work.

ECC is currently used for caches. However, low-latency storage structures such as register file, latches and some first level caches (DL0, IL0, DTLB, ITLB) do not employ ECC due to its huge cost in area and delay, and use parity instead. In order to solve this problem, this paper proposes a set of mechanisms that increases the reliability of the register file against the *erratic bits* problem. Our mechanism detects those registers that have *erratic bits*, recovers from the error and quarantines the faulty register. After the quarantine period, it is able to detect whether they are fully operational with low overhead. As a result, we can detect and recover from all faults due to *erratic bits*.

Since our mechanism enables full recovery for the *erratic bits* problem, structures protected only with error detection techniques (e.g., parity) can operate at lower operating voltages to save power even if the *erratic bits* problem is exacerbated, because we can recover from such errors.

Acknowledgments

The Spanish Ministry of Education and Science under grant TIN2007-61763 partially supported this work.

References

[1] S. I. Association, *International Technology Roadmap for Semiconductors (2005 Edition)*, http://public.itrs.net ed.

[2] T. Ong, A. Fazio, N. Mielke, S. Pan, N. Righos, G. Atwood, and S. Lai, "Erratic erase in etox/sup tm/ flash memory array," in *Proceedings of the Symposium on VLSI Technology (VLSI). Digest of Technical Papers.*, 1993, pp. 83–84.

[3] M. Agostinelli, J. Hicks, J. Xu, B. Woolery, K. Mistry, K. Zhang, S. Jacobs, J. Jopling, W. Yang, B. Lee, T. Raz, M. Mehalel, P. Kolar, Y. Wang, J. Sandford, D. Pivin, C. Peterson, M. DiBattista, S. Pae, M. Jones, S. Johnson, and G. Subramanian, "Erratic fluctuations of SRAM cache vmin at the 90nm process technology node," in *Technical digest of IEEE International Electron Devices Meeting (IEDM)*, December 2005, pp. 655–658.

[4] G. Semeraro, D. Albonesi, S. Dropsho, G.Magklis, S. Dwarkadas, and M. Scott, "Dynamic frequency and voltage control for a multiple clock domain microarchitecture," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO 35)*, 2002, pp. 356–367.

[5] A. Iyer and D. Marculescu, "Power efficiency of voltage scaling in multiple clock, multiple voltage cores," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design (ICCAD02)*, 2002, pp. 379–386.

[6] R. Baumann, "Soft errors in advanced computer systems," in *Proceedings of IEEE Design and Test of Computers*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 258–266.

[7] C. Chen and M. Hsiao, "Error-correcting codes for semiconductor memory applications: A state of the art review," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124–134, 1984.

[8] I. Koren and Z. Koren, "Defect tolerance in vlsi circuits: techniques and yield analysis," *Proceedings of the IEEE*, vol. 86, no. 9, pp. 1819–1838, 1998.

[9] 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.

[10] E. Fetzer, D. Dahle, C. Little, and K. Safford, "The parity protected, multithreaded register files on the 90-nm Itanium microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, January 2006.

[11] J. Kulkarni, K. Kim, and K. Roy, "A 160 mv, fully differential, robust schmitt trigger based sub-threshold SRAM," in *Proceedings of the 2007 International Symposium on Low Power Electronics and Design (ISLPED07)*, 2007, pp. 171–176.

[12] R. Rajsuman, "Design and test of large embedded memories: An overview," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 16–27, 2001.

[13] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM Press, 2003.

[14] G. Memik, M. Kandemir, and O. Ozturk, "Increasing register file immunity to transient errors," in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2005.

[15] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2005.

[16] G. Reis, J. Chang, N. Vachharajani, R. Rangan, D. August, and S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[17] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1999.

[18] L. Spainhower and T. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," *IBM Journal of Research and Development*, vol. 43, no. 5/6, pp. 863–873, 1999.

[19] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th annual International Symposium on Computer Architecture (ISCA)*, 2002.

[20] E. Rotenberg, "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the Annual International Symposium on Fault-Tolerant Computing (FTC)*, 1999, p. 84.

[21] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002.

[22] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance," in *Proceedings of the 33th International Symposium on Microarchitecture (MICRO)*, 2000.

[23] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, 2003.

[24] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN07)*, 2007, pp. 286–296.