Design for Soft Error Mitigation

Michael Nicolaidis, *Member, IEEE*

Invited Paper

***Abstract*—In nanometric technologies, circuits are increasingly sensitive to various kinds of perturbations. Soft errors, a concern for space applications in the past, became a reliability issue at ground level. Alpha particles and atmospheric neutrons induce single-event upsets (SEU), affecting memory cells, latches, and flip-flops, and single-event transients (SET), initiated in the combinational logic and captured by the latches and flip-flops associated to the outputs of this logic. To face this challenge, a designer must dispose a variety of soft error mitigation schemes adapted to various circuit structures, design architectures, and design constraints. In this paper, we describe various SEU and SET mitigation schemes that could help the designer meet her or his goals.**

***Index Terms*—Alpha particles, atmospheric neutrons, design for reliability, design for soft error mitigation, fault tolerant design, nanometric technologies, single-event transients, single-event upsets, soft errors.**

I. Introduction

IN THE PAST, progress in very large scale integration (VLSI) technologies improved dramatically the reliability of electronic components, restricting the use of fault tolerance in a very restrictive number of application domains. We recently reached a point where these trends are reversed. Drastic device shrinking, increased complexity, power supply reduction, and increasing operating speeds that accompany the technological evolution to nanometric technologies have reduced dramatically the reliability of deep submicron ICs. A significant problem is related to soft errors induced on one hand by alpha particles produced by radioactive isotope traces found in packaging, bonding, and die materials and on the other hand by atmospheric neutrons created by the interaction of cosmic rays with the earth atmosphere [1]. While alpha particles are electrically charged and create a track of electron hole pairs when they pass through an IC, neutrons are electrically neutral and create soft errors in an indirect manner: a high energy neutron may interact with a silicon, oxygen, or other atom of the chip to create electrically charged secondary particles able to induce soft errors. It is to note that the interaction can create numerous secondary particles that could simultaneously affect several circuit nodes. Thermal neutrons may also represent a major source of soft errors, but this source is eliminated by borophosphosilicate glass (BPSG) removal in recent processes [2].

When a sensitive node, typically the drain of an off transistor, is in the proximity of the ionization track of an electrically charged particle, it collects a significant part of the generated charge carriers (holes or electrons), resulting in a transient current pulse on this node. The effect of this pulse depends on the type of the cell to which the struck node belongs. In a storage cell (e.g., a memory cell, a latch, or a flip-flop), a sufficiently strong pulse will reverse the cell state, resulting in a single-event upset (SEU). This is the most typical case of soft errors. When the collecting node belongs to a logic gate, the transient current pulse is transformed to a voltage pulse (single-event transient, SET) whose form and amplitude depend on the characteristics of the current pulse and of the electrical characteristics of the collecting node (load and strength of the transistors driving the node). The pulse can be propagated through one or more paths of the combinational logic to reach some latches or flip-flops. Such a path is composed of a succession of input–output pairs of a set of gates connected in series. Pulse propagation is conditioned by the state of the combinational logic since a controlling value (a 0 for a NAND or a 1 for a NOR gate) on one input of a gate will block the propagation of an error from another input of the gate to the gate output. Thus, a transient pulse can be propagated only through a sensitized path, that is, through a path such that all the inputs of the gates composing the path, excepting their inputs belonging to the path, have noncontrolling values. The pulse propagation is also conditioned by the pulse duration. With a good approximation, we can say that if the pulse duration is lower than the logic transition time of a gate, it cannot be propagated though it. If this duration exceeds the double of the logic transition time of a gate, the pulse is propagated without attenuation. The pulse is propagated with some attenuation if its duration is between these two values.

Finally, when the pulse reaches a latch or a flip-flop, it will be captured to produce a soft error only if it overlaps with the clock event. These conditions make soft errors induced by SETs more rare than SEUs induced by particles striking the nodes of a storage cell. However, the situation is changing for two reasons: as the logic transition time of logic gates becomes extremely short, the durations of transient pulses become quite larger than this time and they are no more filtered by the electrical characteristics of the gates [3]–[5]. In addition, as the clock frequencies become very high, the probability that the transient pulses overlap with the clock event also becomes

Manuscript received February 15, 2005; revised June 21, 2005.

The author is with iRoC Technologies, Santa Clara, CA 95054 USA (e-mail: michael.nicolaidis@iroctech.com).

Digital Object Identifier 10.1109/TDMR.2005.855790

high [3]–[5]. Finally, the path sensitization condition may not have a significant impact on reducing circuit sensitivity to SEUs since a node of a combinational circuit is usually connected to the circuit outputs through several paths. So, the probability that none of them is sensitized can be low. Indeed, the pulse will often be propagated through several paths, resulting on several pulses that reach the circuit outputs at different times. This increases the probability that some of these pulses overlap with the clock event when it reaches a latch/flip-flop.

The soft error rate (SER) produced by these effects may exceed the failure in time (FIT) specifications in various application domains. In such applications, soft error mitigation schemes should be employed for memories and eventually for logic. Soft error mitigation techniques include approaches using process modifications such as, for instance, the removal of BPSG for eliminating soft errors produced by thermal neutrons [1], [2], approaches adding extra process steps such as, for instance, the addition of DRAM-type capacitance to the nodes of memory or latch cells, and design-based approaches. In this paper, we only address design solutions. These solutions include electrical-level, gate-level, and architectural-level approaches.

Error-correcting code (ECC) is the conventional solution used to reduce SER in memories. It may incur a moderate area cost, acceptable by most designers. But in some designs, the speed and/or area penalty incurred by ECC may be undesirable. In addition, in some types of memories, ECC may be inapplicable or may incur a very high area cost.

Duplication and comparison or triple modular redundancy (TMR) and majority voting are the most commonly used solutions for reducing SER induced by SEUs and SETs in logic parts. But they incur excessive area and power penalty, making them unacceptable in most designs.

In this paper, we discuss various alternative designs for soft error mitigation solutions, aimed to cope with the shortcomings of conventional solutions.

II. Design for Soft Error Mitigation in Memories

Memories represent the largest parts of modern designs. In addition, they are more sensitive to ionizing particles than logic since they are designed to reach the highest possible density. As a matter of fact, the SER of modern designs is dominated by the SER of their memories. Therefore, when it is required to reduce the SER of a design, protecting memories is usually the first priority.

A. Error Detecting Codes and Error Correcting Codes

The cheapest solution for protecting a memory is to add a parity bit to each memory word. During each write operation, a parity generator computes the parity bit of the data to be written. The data together with the computed parity bit are written in the memory. If a particle strike alters the state of 1 bit of a memory word, the error is discovered by checking the parity code during each read operation. Because this scheme detects but does not correct the error, it must be combined with a system-level approach for error recovery. This reduces the interest of the scheme since it increases the complexity of the system design. However, in some situations, error recovery can be very simple. For instance, if the memory is an instruction cache or a write-through data cache, all the data in the cache can also be found in the main memory. Thus, the erroneous data can be recovered from this memory by simply activating the miss signal of the cache each time the parity checker detects a parity error. But in a situation where error recovery is more complex, it may be preferable to protect the memory by means of codes enabling error correction.

While numerous error correcting codes (ECCs) exist, the incurred area and the speed penalty for implementing them can be excessive, especially for the ones that enable correction of multiple errors. As a matter of fact, codes enabling single-error correction, like the Hamming code, are the most used in practice for protecting memories. Such codes will be sufficient in most practical situations since the probability that a single event flips several cells is low. However, as we move to higher device densities, the probability of this situation increases since memory cells are closer to each other and can be flipped by a single ionizing particle. In addition, as the reaction of a neutron with a silicon, oxygen, or another atom usually creates several secondary particles, the probability of multicell upsets is becoming nonnegligible. Thus, in designs targeting quite high levels of reliability, protection against multicell upsets has to be considered carefully. In such designs, we can still use ECCs correcting single errors, but we must select memories using large column multiplexing. In this case, the distance between cells belonging to the same memory word is high and the probability of an error affecting more than 1 bit of the same word vanishes.

Coming back to ECCs correcting single errors, in a memory that stores words of n bits, these codes add k check bits, where k verifies the relation $k = \lceil \log_2(n + k + 1) \rceil$. It is obvious that these codes will incur high extra area and power dissipation for memories with short words and moderate or low extra area and power dissipation for memories with larger words. For instance, for a 4-bit word length, the extra area and power dissipation is roughly 75%. This goes down to 50% for an 8-bit word length, 31% for a 16-bit word length, 19% for a 32-bit word length, and 11% for a 64-bit word length.

When a double error affects a memory word, these codes cannot correct it but they can detect it. However, they cannot signal if the error is single and thus correctable, or double and thus noncorrectable. By adding a bit that computes the parity of the data and check bits, it is possible to distinguish single from double errors affecting the data and/or the check bits since the parity bit will detect the single errors but not the double ones.

From the above discussion, we observe that ECCs correcting single errors are very convenient for modern designs using large data widths since their cost for large word widths becomes moderate. However, some other shortcomings may make their use problematic. The first one is the speed penalty introduced by the circuitry performing check bit computation during a write operation and error correction during a read operation. This penalty increases as the word width increases and can become a major problem especially to those memories where ECC is most suitable since it introduces low area and power

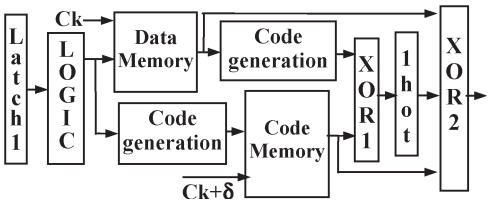


Fig. 1. Elimination of extra delay in the path of write data.

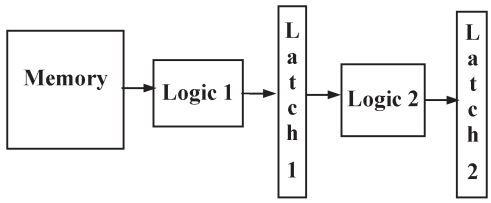


Fig. 2. System with an unprotected memory.

penalty. Solutions allowing to cope with this problem are therefore mandatory.

B. Removing Speed Penalty in Memories Protected by ECC

ECC introduces extra delay in the path of write data due to the circuitry used to compute the check bits of the write data and in the path of read data due to the circuitry used to compute the check bits and correct the error in the read data. It may be suitable in a design to remove the extra delay from the one or both of these paths.

Fig. 1 depicts the block diagram of an implementation eliminating extra delay in the path of the write data [6]. As shown in this figure, the data bits and the check bits are stored in two separate memories (data memory and code memory). We can see in the figure the block generating the check bits from the data bits to be written in the data memory as well as the block generating the check bits from the data bits read from the data memory. The later check bits are bit-wise XORed with the check bits read from the code memory by a block of XOR gates (XOR1), which produce the error syndrome. This syndrome indicates the position of the erroneous bit in a binary code form. A combinational block transforms this form into a 1-hot code (1-hot block). Finally, a second block of XOR gates (XOR2) bit-wise XORs the 1-hot form of the syndrome with the data bits read from the data memory and the check bits read from the code memory to correct eventual errors affecting these bits.

In this scheme, the data bits are written in the data memory as soon as they are ready. Thus, the frequency of the clock signal Ck is the same as in the case of a system that does not use ECC. This way, the speed penalty on the write operations is eliminated. However, an extra delay is added on the inputs of the code memory by the code generation block. To manage this delay, the code memory uses a clock signal $\mathrm{Ck} + \delta$, which has the same frequency as the signal Ck, but it is delayed with respect to this signal by a delay δ equal to the delay of the code generation block. As a matter of fact, the check bits are ready when the signal $\mathrm{Ck} + \delta$ enables writing these check bits in the code memory. But since the signal $\mathrm{Ck} + \delta$ enables both the write and the read operations from the code memory, then, during a read operation, the check bits will be read with a delay δ with respect to the data bits read from the data memory. However, this delay does not increase the delay of the error correction process because the check bits read from the code memory are applied directly to the XOR1 block while the data bits read from the data memory have to traverse the code generation block, which has a delay equal to δ.

The delayed clock signal $\mathrm{Ck} + \delta$ will be generated locally by adding a delay element on the Ck signal of the data memory. This local generation minimizes skews between the clock signals of the data memory and the code memory. Careful implementation of the delay element will consider worst-case delay of the code generation block as well as best-case (minimum) delay of this element. Another possibility is to use a single clock signal for both memories, but use the one edge of this signal (e.g., the rising edge) as the active edge for the data memory and its second edge (e.g., the falling edge) as the active edge for the code memory. This will work if the delay of the code generation block does not exceed the time interval that separates these edges. From the synchronization point of view, this approach is similar to the time redundancy approach using duplicated flip-flops with shifted clocks (see Section III-B-3, Fig. 8). For the same reasons as the ones discussed in Section III-B-3, the code generation block and/or the logic block may by padded to guarantee that no path between Latch1 and data memory has delay lower than δ.

Another implementation consists of adding a pipeline stage in the code generation block. With this solution, the operations in the check memory will be performed one clock cycle later than in the data memory. This solution will work if the delay of the code generation block does not exceed one clock cycle. If this delay is larger than the clock cycle, more pipeline stages are added. This means that a write in the code memory will be performed more than one clock cycle after a write in the data memory. This large delay is balanced by the delay of the code generation block placed on the read data since, in this block, a similar number of pipeline stages are added. Long delays in the later block will be handled by the scheme that removes speed penalty on the read data, as described below.

Fig. 2 depicts the block diagram of a system where the data read from a memory pass through a combinational logic block (Logic1) and are stored in a stage of latches/flip-flops (Latch1). The block Logic1 can eventually be empty. This corresponds to the case where the data read from the memory are stored directly to the Latch1 stage of latches. In this figure, the memory is not protected by an ECC. If ECC is used, the error detection and correction circuitry will be placed between the memory and the combinational block Logic1, increasing significantly the signal delay and decreasing accordingly the clock frequency.

In many designs, it may be required to maintain the clock frequency of the initial (unprotected) design. For doing so, one solution consists of adding a pipeline stage (i.e., a stage of latches) between the memory and the latch-stage Latch1 in Fig. 2. This solution may reach the desirable clock frequency if the delay of the detection and correction logic does not

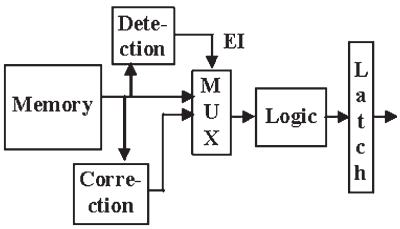


Fig. 3. Elimination of detection and correction delays.

exceed the clock cycle. But the system performance will still be impacted as the data read from the memory will reach the latch stage Latch1 one clock cycle later than in the unprotected design. This may not be desirable in many designs.

A second solution is based on the observation that generating an error detection signal is faster than correcting the error. Based on this observation, the clock frequency can be determined to account only for the delay of the error detection signal. The data read from the memory are supplied to the system through a MUX. The first inputs of the MUX come directly from the memory and the second inputs from the error correction block. In error-free operation, the first inputs of the MUX are supplied to the system. When the error detection signal detects an error, the system is halted for one clock cycle to provide some extra time for performing the error correction. In the next cycle, the second inputs of the MUX are supplied to the system. While this solution reduces the speed penalty, the designer may yet not achieve its target speed due to the delay added by the error detection signal.

Fig. 3 depicts a scheme that eliminates both the error detection and the error correction delays [6]. This figure corresponds to the design of Fig. 2 but considers that the memory is protected by an ECC. For simplicity, the detection and correction blocks are shown to be distinct, but in reality they share the code generation block and the XOR1 block. The idea here is to supply through a MUX the data to the system as soon as they are read from the memory. Thus, we can use a clock period that does not account for the error detection and error correction delays. In this case, the system will work properly as far as no errors are present in the data read from the memory. During this time, the MUX supplies to the system the data coming directly from the memory. But when an error is present in the read data, it is propagated and contaminates the system before the error detection signal indicates it. In this case, when error detection signal (EI) is activated, it activates during the next clock cycle the hold signal of all the latches of the system but the latch stage Latch1. It also enables the MUX to supply to the system the data coming from the error correction block. Thus, during one clock cycle, all the latches but the latch stage Latch1 are hold, maintaining their previous state, while the latch stage Latch1 is decontaminated during this cycle since it receives the corrected data. The system restarts on the next clock cycle from the correct state.

In the above discussion, we consider that the erroneous data propagate to a single stage of latches (Latch1). But if the clock is very fast and/or the data width is large (large delay of the detection block) and/or the system is large (long interconnections for distributing the hold signal), it may take several clock cycles for detecting the error and holding the system latches. In such cases, the erroneous data will be propagated and contaminate more than one latch stages before the system latches are hold and the decontamination process is activated. To handle this case, the detection and correction blocks are pipelined to make them compatible with the clock cycle. The number of pipeline stages used in the detection block also accounts for the delay required for EI signal to activate the hold signal along the circuit (delay of interconnections, etc.). Then, as described in [6], the decontamination process will take several clock cycles (as many as the number of contaminated latch stages). In addition, the number of cycles during which the latch stages are hold is variable, depending to the number of pipeline stages that separate each latch stage from the memory.

The case where the logic block placed between the memory and Latch1 receives inputs from other circuits in addition to the read data is also treated in [6].

C. ECC Versus Nonstandard Memories

The above discussions consider memories performing read and write operations in the conventional manner. Implementing ECC in some memories performing read and write operations in the nonconventional manner, such as memories with maskable operations and content-addressable memories (CAMs), can be more difficult.

In a memory allowing maskable operations, some writes will write data on all the bits of the word selected by the current memory address and some other writes will write data only on a subset of bits of the memory word. This subset of bits is determined by setting the bits of a mask. When a subset of the bits of a word is written, the contents of the remaining bits are unknown. Thus, it is not possible to compute the new check bits for this word [7]. To cope with this problem, several solutions are possible [7].

One consists of coding separately several subsets of the bits of each word. This is possible because usually the maskable writes cannot be performed over any random collection of the bits of a word. For instance, for a memory using 64-bit words, the word can be divided into eight fields of 8 bits and the maskable writes are performed over one of these fields. In this example, instead of implementing a single code for the whole word, we will implement one code for each one of the above eight fields. Because during a maskable or a nonmaskable write all the bits of each written field are known, the check bits for each written field can be computed. While this solution enables implementing the code without ambiguity, the area cost can be significant because the size of the encoded data is small. For instance, in the above example, we will need to add 50% extra bits to store the check bits in the memory (four check bits for every eight data bits). The worst case corresponds to memories where maskable writes are performed over single bits. In this case, the above solution leads on triplicating the memory bits.

Another solution, which allows eliminating this high cost, consists of performing a read before each maskable write, correct the error in the read data, if any, and combine the read and the written data to compute the new check bits. This solution

reduces the area cost for protecting memories with maskable operations to the cost required for regular memories. However, performing an extra read implies a performance penalty that may not be acceptable in many systems.

A third solution consists of exploiting the following: when a maskable write is performed, the bit lines of the bits that are not affected by this write are precharged and the amplifiers of these bits are in the read mode. In this situation, a simple modification of the memory allows reading these bits simultaneously with the write operation performed over the bits selected by the maskable write. In this manner, we dispose the values of all the bits of the memory word (i.e., the values written in the bits selected by the maskable write and the values of the nonselected bits). Thus, we can compute the check bits for the new data stored in this word and write these check bits in a separate memory (code memory). Because the write of certain bits of the word and the read of certain other bits are performed in parallel, there is no performance penalty. However, we need to cope with a major difficulty: the read bits may include errors. These errors must be corrected before computing the new check bits, otherwise the check bits computation will mask the errors. For detecting and correcting these errors, we need to know the values stored in every bit of the word before performing the maskable write. This means that we need to read all the bits of the word, but as described above we only read the bits not affected by the maskable write. So, detecting and correcting an error on these bits is not possible by using conventional coding schemes. A nonconventional solution is proposed in [7]. The simplest implementation corresponds to memories that perform maskable writes over fields composed of a single bit. In this case, using a single error correcting/double error detecting codes (e.g., the Hamming code and a parity bit) and a special error correction circuitry allows correcting any single error in the read bits. So, in this case, the cost of the check bits is the same as for the conventional codes. When maskable operations use larger bit fields, a more complex code is required. This code adds $k + 2$ extra check bits, where k is the size of the field on which a maskable operation is performed. The details can be found in [7].

This approach can also be used in memories where maskable writes can be performed over random collections of bits, in which case the scheme implementing one code per bit field is not applicable. For such memories, the use of the approach performing parallel reads and writes of the bits of a word requires implementing the memory in a manner that several words can be read in parallel. This can be done by selecting an appropriate memory size. For instance, for a memory with 16-bit words, we can use a memory with 64-bit words but having a number of words four times less than the initial memory. Operations in the new memory are performed in a tricky manner that makes them equivalent to the operations performed in the initial 16-bit memory, and at the same time, it allows proper code computation for maskable and nonmaskable operations. The scheme allows a drastic reduction of the check bits. For instance, for a memory using 16-bit words and performing maskable writes over random bits, the only alternative solution to the new codes is triplication (i.e., 200% of extra bits). If we implement the new code by using a memory that allows reading in parallel 64 bits (four 16-bit words), the extra bits represent 39% of the bits of the unprotected memory. But this number has to be considered carefully since the area overhead will be higher than 39%. In fact, with this scheme, in order to perform the code computation process properly, the check bits have to be stored in a separate dual-port memory allowing simultaneous read and write of two words. The higher area occupied by such memories will lead for this example to a total extra area of 60% to 70%. But this area remains still much lower than the 200% extra area required if we use the only other alternative solution.

Another kind of difficult to protect memories are CAMs. Each word in such memories comprises a data field and an address field (or tag field). The address field of each word includes a comparator that allows comparing in parallel all the addresses stored in the CAM against an address applied on the address inputs of the CAM (associative search). When the address stored in one of the CAM words matches the externally applied address, a hit is activated and the data stored in the data field of this word are read or new data are written. While many varieties of CAMs exist, including ternary CAMs or TCAMs that allow masking a selected set of bits of the address field of a word or of the externally applied address, all varieties share a common characteristic that makes difficult the complete protection of a CAM. When an error affects the address stored in a CAM word, an associative search of the correct address value will produce an incorrect miss. This error cannot be detected since the word containing the erroneous address is not read to check for its integrity. As a matter of fact, we can use parity or ECC codes to protect the data field and the address field of CAMs. With this solution, the data field will be completely protected. However, the address field can be protected only when the error leads to an incorrect hit since in this case the hit word can be read and checked, but it cannot be protected when the error leads to an incorrect miss. An incorrect miss will not be a problem in systems where CAM is used as a cache memory since, in this case, the data will be retrieved from the main memory. But in numerous systems employing CAMs, this is not the case.

The above discussions illustrate that for some memory types, it may be quite expensive or even impossible to achieve protection against soft errors by means of ECC. In addition, ECC may introduce a nonnegligible area cost and a significant speed penalty even for memories performing conventional read and write operations. In this context, alternative approaches using built-in current sensors (BICS) or hardened storage cells gain interest.

D. Memory Protection Based on BICS

A memory cell being in the steady state drives a very small current. But when its state is reversed due to the impact of an ionized particle, abnormal current flows through the Vdd and Gnd lines of the cell. One component of this current dissipates the charge collected by the struck node of one of the cross-coupled inverters composing the cell. The second component corresponds to the switching current flowing through the second inverter during the transition period of the cell. It is then possible to detect the occurrence of an SEU by implementing

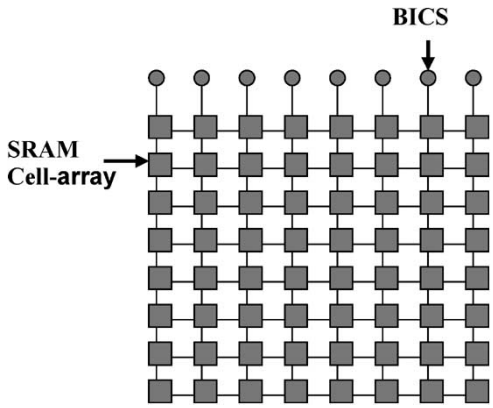


Fig. 4. Memory protection using BICS.

one current sensor (BICS) on each vertical power line of the memory cell array [8], as shown in Fig. 4.

Because BICSs monitor the vertical power lines of the memory, the detection of an SEU also indicates the position of the affected bit. This is also true when two neighbor columns share the same power line since memories usually use column multiplexing. In such a case, neighbor cells belong to two different words and an error signaled by a BICS determines which bit is possibly affected in each of these words. It is then possible to add a parity bit to each memory word and use it to locate the faulty word. When the faulty word is located, the error is corrected by reversing the value of the bit indicated by the active BICS.

Various approaches can be used to locate the faulty word. The simplest one is to wait until the faulty word is read by a regular read operation of the system. By checking the parity of the read data during each read operation, the error is detected and corrected as described above. The problem of this approach is that it may elapse a long time between the occurrence of an error and the instant that the erroneous word is read by a regular read operation. If in the mean time a second SEU occurs in the memory and is detected by a different BICS, then, when an erroneous word is located, it will not be possible to determine if the erroneous bit is in the position of the first BICS or the second BICS. This situation will cause practical problems only in radiation hostile environments where SER is very high. Of course, this is not the case of a terrestrial environment.

A second approach consists of reading periodically the memory in order to bound the time during which the memory contains a latent error. This approach requires a more complex implementation, but as said earlier it is not really necessary for the terrestrial environment.

The BICS approach was validated by an early implementation in a 0.25-μm process [9]. Simulation results show that the technique still works in advanced nanometric technologies [10], but silicon prototypes are required for further validation.

The technique is promising since it requires a very low area cost of 4% to 7% according to each specific implementation and is very suitable to protect memories that are difficult to protect by ECC, such as memories using maskable operations or CAMs. In the case of CAMs, during associative searches, the address field is neither read nor written. Thus, after the activation of a BICS allocated to the address field of a CAM, the cycles of associative searches can be used to read the words of the address field, locate the error, and correct it. In large CAMs composed of several smaller CAM blocks, the activation of BICs will also indicate the affected CAM block, allowing for a faster location of the erroneous word.

III. Design for Soft Error Mitigation in Logic

As stated earlier, logic is affected by SEU-related and SET-related soft errors. SEUs occur when an ionizing particle striking a sensitive node of a latch or flip-flop cell flips the cell state. SET-related soft errors occur when a transient pulse, initiated by an ionizing particle striking a sensitive node of a logic gate, is propagated through the subsequent gates of the combinational logic and captured by a latch or flop-flop. The largest part of soft errors affecting logic designs is coming from SEUs, although the part coming from SETs is increasing as we move to higher integration densities. As a matter of fact, when the SER of logic parts of a design exceeds the specified FIT for these parts, protecting latches or flip-flops may be sufficient to achieve the specified FIT. This can be done by using hardened latches/flip-flops. But if mitigating SEUs affecting the latches/flip-flops do not allow meeting the target FIT, a more complete soft error mitigation approach has to be used to cope with SET-related soft errors. This approach can be based on hardware (space) redundancy and/or time redundancy.

Note also that SEUs have a particular impact on designs implemented in SRAM-based field-programmable gate arrays (FPGAs). In this case, the function of the design is determined by the values stored in the configuration SRAM cells of the FPGA. Thus, an SEU affecting such a cell has a much more severe impact than SEUs affecting memories, latches, and flip-flops as it modifies the very function of the design. An approach to fix this problem consists of reading periodically the configuration memory, checking its contents and reprogramming the FPGA when an error is detected. This technique is acceptable only if the application tolerates long error latencies. If this is not acceptable, TMR is necessary. TMR can also be combined with the read reprogram scheme to avoid error accumulation in the configuration memory. The interest of the TMR approach is its flexibility, as every designer can employ it. But its major drawback is an excessive hardware overhead, which exceeds 200%. Using hardened SRAM cells to implement the configuration memory of FPGAs is the less area consuming solution for solving this problem. Its major drawback is a low flexibility since a designer that needs to improve the reliability of its design cannot employ it. It has to be done by the FPGA provider. However, for economical reasons, the later will not provide hardened FPGA families unless it becomes mandatory for a significant part of its market segment.

A. Hardened Storage Cells

Hardened static storage cells (SRAM cells, latches, flip-flops) preserve their state even if the electrical state of some of their nodes is altered by an ionizing particle strike. Various hardened storage cells have been proposed in the literature

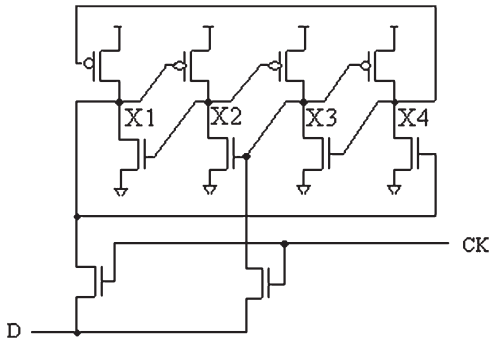


Fig. 5. Hardened SRAM cell.

[11]–[13]. The basic drawbacks of these cells are their hardware cost, which is generally close to duplication and the fact that their hardening principle requires specific transistor sizing. As a result, these cells do not scale easily as the device size is shrinking. The principle used in dual interlocked storage cell (DICE) [14] is of different nature. It achieves immunity against transients affecting any single node without requiring any particular transistor sizing. Therefore, it scales for any process generation and can be implemented by using minimum transistor size. This makes the DICE cell suitable for protecting storage cells against SEUs in nanometric technologies.

The transistor diagram of the DICE cell is shown in Fig. 5. The cell has two states, the 0 state ($X1 = 0$, $X2 = 1$, $X3 = 0$, $X4 = 1$) and the 1 state ($X1 = 1$, $X2 = 0$, $X3 = 1$, $X4 = 0$). In any of these two states, and whichever is the node affected by a transient pulse, we can always find among the remaining three nodes two consecutive nodes having the values 1 and 0. We call these two nodes the hold nodes and the two other nodes the affected nodes. For instance, in the 0 state, if the node struck by a particle is X2, the hold nodes are the nodes X4 and X1. In the same state, if the struck node is X3, the hold nodes are again X4 and X1, while if the struck node is X1 or X4 the hold nodes are X2 and X3. We can find similarly the hold nodes for the 1 state and for any struck node. In any of these situations, we can check easily that the states of the hold nodes either are not changed by the effects of the particle strike or, in the worst case, they are driven in the high-impedance state. Thus, the hold nodes preserve their correct values. The state of the other two nodes can be modified by the particle strike, but one transistor of each inverter driving one of these nodes is driven by one hold node. As a consequence, currents flowing through the transistors of these inverters quickly restore the correct values in the affected nodes.

By using the DICE cell, it is easy to built hardened SRAM cells, latches, and flip-flops. So, the DICE cell can be used to protect SRAMs, register files, CAMs, the configuration memory of FPGAs, and the latches or flip-flops of logic designs.

Comparison between industrial flip-flop and DICE-based flip-flop performed for a 90-nm process node shows 81% area overhead, identical rise time (0.13 ns in both case), and a 20% fall time increase (0.12 ns versus 0.10 ns). The 20-ps increase of the fall time will represent a very small amount of the delay of a pipeline stage and will have a very small effect on the clock frequency.

During static operation, the cell presents complete immunity for any transient affecting a single node of the cell and regardless to the transient strength. During dynamic operation (which is a concern only for latches and flip-flops), a transient affecting a single node of the cell could induce an erroneous state, but this can only happen if the cell is operating with very stringent timing conditions and even in this case the particle strike should happen at specific time frames. As a matter of fact, the cell presents excellent hardening characteristics. Its practical use can however be impacted by a significant extra area with respect to standard storage cells. This is in particular true if it has to be used to harden the configuration memory of FPGAs or to harden memories difficult to protect by other schemes (e.g., CAMs). This high area is also a drawback for the other known hardened cells. Therefore, to make hardened cells more attractive, new low-area cells are needed. Such cells are currently under validation and will be reported in further communications.

B. Set Mitigation

Protecting combinational logic is more complex than protecting memories or latches and flip-flops and may involve quite high hardware cost. It is therefore mandatory to select properly the protection scheme in order to meet design requirements in terms of area, speed, power, and reliability. The good news is that we could use a single scheme for mitigating both the soft errors coming from SETs as well as those coming from SEUs in the latches/flip-flops of logic parts.

Basically, there are two approaches for protecting logic: error masking and error detection. A typical example of error masking is triple modular redundancy (TMR), where the circuit under protection is triplicated and a majority voter determines the circuit output. A typical example of error detection is duplication and comparison. Excepting some systems in which the fail stop concept can be used (the system operation is stopped upon error detection), in the majority of systems, error detection should be combined with a recovery scheme that brings the system into a correct state from which operation can be resumed. Since soft errors are transient faults, the recovery could consist of reexecuting the latest operation. Another possibility consists of using checkpoints, that is, particular points of the system operation in which the system state is saved in a mass storage media. When an error is detected, the system operation resumes from the state saved during the last checkpoint (rollback recovery) [15], [16].

Because massive redundancy schemes such as TMR and duplication involve excessive hardware cost and power dissipation, in the following we discuss alternative schemes aimed at reducing this cost.

1) Time Redundancy Implemented in Software: Though this paper is concerned about hardware-based soft error mitigation approaches, for completeness we rapidly refer two software-based schemes that could be of interest to the reader. The first software-based soft error mitigation technique executes duplicated instructions over duplicated data and compares the results to check their integrity [17], [18]. This protection does not detect control flow errors, which are often the most

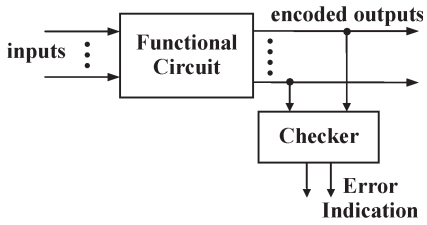


Fig. 6. General structure of self-checking circuits.

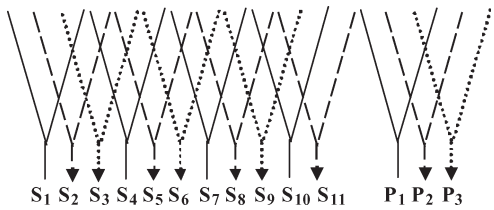


Fig. 7. Group parity coding.

undesirable. So, extra instructions are added in strategic points of the program to check for such faults [19]. Software rollback can also be implemented to achieve error recovery [20]. The interest of the software-implemented approach is its flexibility since it does not require any hardware modification. The counterpart is a high memory overhead (5× for error detection, 6× for error detection and recovery) and speed penalty (3× for error detection, 4× for error detection and recovery) [21].

Of course, the cost is reduced if control flow checking [19], [22], [23] alone is used, but the protection is incomplete. Evaluation of the cost of these approaches reported in [19] for four different benchmark programs shows a speed penalty varying from 107% to 185% for the scheme in [23], from 120% to 426% for the scheme in [22], and from 110% to 354% for the scheme in [19]. The memory overhead varies, respectively, from 124% to 338% for [23], from 153% to 630% for [22], and from 129% to 496% for [19]. The significant difference between the costs incurred by these methods is reflected to a significant difference between their error detection efficiencies.

Multithreading is another software-based transient fault detection and recovery approach [24], [25] that reduces memory and speed cost by exploiting the capabilities of modern processors to execute multiple threads of computation. The speed penalty of this approach is only 40% [26], but it is less flexible than the first approach as it requires some hardware modifications.

The applicability of both approaches is limited to processor cores, so, for designs of different nature, only hardware-based approaches can be used.

2) Error Detection Based on Space (Hardware) Redundancy: To avoid the large area cost incurred by duplication, self-checking design can be used. In this design, the protected circuit generates outputs encoded into an error detecting code (parity, m-out-of-n, Berger, or arithmetic code) and the circuit outputs are checked by the checker of this code, as shown in Fig. 6.

There is a rich literature on self-checking circuit design. Most of the publications consider fault-secure implementations. This property means that the self-checking scheme detects any error produced by a modeled fault. But usually these publications consider the stuck-at fault model. They also address the self-testing property, which guarantees that each modeled fault is manifested by at least one input vector. This property precludes the existence of faults that can stay undetectable forever and eventually become dangerous if combined with another fault occurring later. Obviously, the self-testing property is not of interest if we restrict our fault model to transient faults.

Fault-secure design using the parity code: This code detects all single errors and more generally all errors of odd multiplicity. So, to achieve the fault-secure property, it is usually used in circuits where each internal line is connected with a single output (circuits with maximum divergent degree equal to 1). To make this design practical, the outputs of logic blocks can be partitioned into several groups such that any two outputs sharing some logic belong to different groups [27]. This is illustrated in Fig. 7, where each cone represents the combinational logic that generates a circuit output. In this figure, plain line cones belong to one group, doted line cones to a second group, and dashed line cones to a third group. We observe that the cones belonging to the same group do not have common parts. Then, by using a parity bit for each group, it is guaranteed that a fault could not affect more than one output of a given group and the errors are always detected by checking the parities. The parity outputs can be implemented either without sharing logic gates with the regular outputs or by sharing such gates. In the later case, attention is paid so that a parity output shares gates only with regular outputs not belonging to the group protected by this parity output. Some logic replication can also be used to reduce the number of parity groups [27], [28]. This is done when a small portion of logic is shared by several outputs. In this case, by replicating this portion of logic, we can obtain a lower number of parity groups resulting in a reduced total cost (cost of the replicated logic parts and of the logic generating the parity bits). Note that some modern synthesis tools often perform this type of replication in order to reduce interconnection cost by maintaining a local connectivity between inputs and outputs of logic gates. Experiments presented in [28] over 14 Microelectronics Center of North Carolina (MCNC) combinational benchmark circuits show an area overhead for the technique varying from 35% to 171%. Thus, this approach can be efficient for some blocks of a design and inefficient for some other blocks. Thus, the designer may select another scheme for protecting the later blocks. In particular, the scheme using parity groups may be inefficient for arithmetic circuits since in such circuits the cones of any two outputs are overlapping.

Finally, for some standard building blocks of VLSI circuits, such as adders, multipliers, dividers, and shifters, a thorough analysis may lead to specific fault-secure solutions. For instance, such solutions may constrain errors to always propagate on an odd number of outputs, it may use specific detection means for some difficult faults, etc. Such techniques were used in [29]–[34] to protect various types of adders, arithmetic logic units (ALUs), multipliers, and shifters. These solutions often achieve a low or moderate area cost (e.g., 15% to 24%

for adders, 16% to 20% for shifters, and 40% to 50% for multipliers).

Fault-secure design using unordered codes: In these codes, there are no two different code words x and z such that x covers z (to be written $x \supseteq z$), where x covers z means that x has a 1 in each bit position z has a 1. For instance, if $x = 10\,010$ and $z = 10\,001$, then neither x covers z nor z covers x, and this pair can belong to an unordered code. From this property, if an error that includes only bit errors of the type $1 \rightarrow 0$ transforms a code word x into an erroneous word z, then z is covered by x and cannot belong to the unordered code. Thus, the error is detected. Similarly, any error that includes only bit errors of the type $0 \rightarrow 1$ is detectable. Thus, unordered codes detect all unidirectional errors. The most used unordered codes are the m-out-of-n code and the Berger code.

An m-out-of-n code [35] is an unordered code in which information and check bits are merged. It is composed of code words that have exactly m 1s (e.g., 2-out-of-4 code: 1100, 1010, 1001, 0110, 0101, 0011).

The Berger code [36] is an unordered code in which information bits are separated from check bits. In this code, the binary value of the check bits represents the number of 0s in the information bits.

To guarantee the fault-secure property, unordered codes are used in circuit structures in which, for any node of the circuit, all the paths between the node and the circuit outputs have the same inversion parity [37]. In this case, when a fault affecting a circuit node is propagated to the circuit outputs, it is always inverted (if the inversion parity of the paths is 1) or always not inverted (inversion parity 0), resulting on a unidirectional error. As a matter of fact, unordered codes can guarantee the fault-secure property for a limited class of functional blocks. To cope with this problem, one can move all the inversions to the inputs of the block and implement the rest of the circuit as an inversion-free circuit. Thus, any fault in a circuit node excepting the input nodes produces detectable errors. The synthesis tool for fault-secure finite-state machines (FSMs) presented in [38] uses this idea. However, unordered codes lead to higher area cost than the parity codes principally due to the significantly higher complexity of their checkers.

Fault-secure design using arithmetic codes: Arithmetic codes [39]–[41] are divided into separable and nonseparable codes. The separable codes are the most practical. They are obtained by associating a check part X' to an information part X, where X' is equal to the modulo A of the information part X, that is, $X' = |X|_A$. The number A used in this operation is referred as the check base. The check bases that allow the simplest arithmetic code generators and arithmetic code checkers are of the form $2^m - 1$ (low-cost arithmetic codes [42]). Practical implementations for any other odd check base were also proposed [43], but they remain more complex with respect to the ones corresponding to low-cost codes. Arithmetic codes are interesting for checking arithmetic operations because they are preserved under such operations. To achieve minimal cost, the check base $A = 3$ is often used. For this check base, the code includes only two check bits. This check base detects any single arithmetic error (i.e., when the arithmetic difference between the correct and the erroneous values is a power of 2). From this property, the check base $A = 3$ achieves the fault-secure property in all modular adder and multiplier implementations (i.e., circuits built by using full and half adder cells). In arithmetic operators using nonregular structures (e.g., using a carry lookahead adder at their last stage), a thorough analysis is required to determine the check base able to ensure the fault-secure property [44], [45]. In this context, the arithmetic code generators and arithmetic code checkers presented in [43] gain interest [46] as the selected base is not necessary a low-cost one. Arithmetic codes result on significant area overhead for protecting adders but a low overhead for protecting parallel multipliers, especially if the operant size is large. This is because the area of the arithmetic code generators and the arithmetic code checkers is proportional to the operand size while the area of the multipliers is proportional to the square of the operand size. Consequently, the area overhead becomes low as the operand size increases. Experiments using the automation tool presented in [46] show a 10% area overhead for a 32×32 multiplier and a 6% area overhead for a 64×64 multiplier.

Fault-secure designs versus transient faults: The designs described above detect both permanent and transient faults. Thus, they can be used to perform error detection for both fault types. However, these designs were proposed to cover stuck-at faults. Thus, the fault-secure property is proven for such faults, but what about transient faults? Stuck-at faults modify the value of the affected node during the whole duration of the clock cycle. Thus, all the outputs connected to the affected node through sensitized paths will be erroneous. On the other hand, an SET will produce an error on an output only if the transient pulse overlaps with the clock event when it reaches this output. Thus, an SET initiated from a circuit node may affect only a subset of the outputs affected by a stuck-at fault initiated from the same node. Therefore, the errors produced by transients are different than the errors produced by stuck-at faults. As a matter of fact, the detection of the later does not guarantee the detection of the former. Indeed, the fault-secure property for transient faults is achieved only for a particular class of circuits designed to be fault secure for stuck-at faults [47]. These circuits are referred as path-secure circuits. They achieve the fault-secure property by means of structural constraints: the circuit structure is such that the propagation of an error from an internal node to the circuit outputs through any collection of paths gives errors detected by the output code. Among the fault-secure designs described above, those checked by the parity code and using parity groups and those checked by unordered codes and using inversion-free functions are path secure. The other designs, that is, those using parity codes to implement adders, ALUs, multipliers, and shifters [29]–[34], and those using arithmetic codes to implement adders and multipliers [42], [44]–[46], are not path secure. As the fault-secure property for transient faults is not achieved for these designs, before selecting any of them, its efficiency should be evaluated by means of a fault injection tool, like ROBAN [48].

3) Error Detection Based on Time Redundancy: A hardware-based scheme performing error detection of transient faults by means of time redundancy was proposed in [49]. As shown in Fig. 8, this scheme checks the content of the

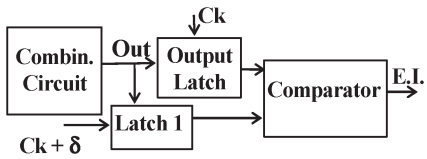


Fig. 8. Error detection based on time redundancy.

functional latch/flip-flop (Output Latch in the figure) by means of an extra latch/flip-flop (Latch1) and a comparator. The extra latch uses a clock delayed by a delay equal to δ with respect to the clock of the functional latch. Thus, the two latches capture the output of the circuit at two instances that differ by a delay δ. As a matter of fact, this scheme detects every transient pulse on the circuit output that has a duration lower than or equal to δ-Dsetup-Dhold (Dsetup and Dhold being, respectively, the set-up time and the hold time of the latches). The scheme will also detect pulses of larger duration affecting its outputs, but not all of them. In fact, if a pulse of larger duration reaches the circuit output at a time that is well adjusted with respect to the clock events of both latches, it could overlap with the set-up time or the hold time of both of them and alter both their states. On the other hand, the scheme does not guarantee detection of all errors induced by SETs initiated in internal nodes of the circuit that have duration lower than δ-Dsetup-Dhold. Indeed, after propagation through the circuit paths, the duration of the transient pulses can be increased when they reach the circuit outputs. This is particularly true when the circuit contains reconvergent paths (multiple paths starting from a node and reconverging to another node). They can propagate the original pulse through several paths and concatenate several pulses into a single one [50]. This pulse can be larger than the original one due to the difference of delays of the propagating paths. Due to the same mechanism, multiple pulses can also appear in the reconvergence node. As a matter of fact, the efficiency of the scheme can vary from one design to another. Thus, to determine its efficiency for a given design, the scheme should be evaluated by a fault injection tool like ROBAN [48].

Note finally that the scheme also detects the SEUs produced when a particle strikes a node of the functional latch (Output Latch) since in this case the content of Latch1 is correct.

The scheme results in a moderate hardware cost corresponding to an extra latch per circuit output plus a comparator. However, it may increase the clock cycle by a time interval equal to $\delta = \mathrm{Dtr} + \mathrm{Dsetup} + \mathrm{Dhold}$. In fact, for $\delta < 0$ ($\mathrm{Ck} + \delta$ precedes Ck), Latch1 captures the output of the combinational circuit at an instant that precede by δ the latching instant of the functional latches (Output Latch). Thus, data on the combinational circuit outputs must be ready at a time interval δ before the end of the clock cycle. Hence, the clock cycle must be increased by δ.

However, for $\delta > 0$, this speed penalty can be eliminated, as shown in [49]. In fact, since no extra delay is added in the signal captured by the output latch, the clock signal of this latch (Ck) can be as fast as in an unprotected design. However, Latch1 captures its input at a time interval δ after the latching edge of Ck. If the minimum delay of combinational circuit is less than δ (more precisely less than $\delta +$ Dhold), the output of combinational circuit may change before it is captured by Latch1. To avoid this problem, the delay of such paths is augmented (using slower gates, adding buffers, etc.).

Another variation of the scheme allowing to reduce its cost is presented in [50]. The scheme removes the extra latch (Latch1 in Fig. 8) and compares the output of the functional latch against its input. The area cost reduction obtained by this scheme is significant as we only have to add a comparator to the original circuit. Its efficiency with respect to SETs is the same as for the scheme of Fig. 8. However, its efficiency with respect to SEUs is reduced. The reason is that the comparison cannot be done at the end of the clock cycle because the input value of the output latch changes. Thus, the content of the output latch cannot be checked until the end of the clock cycle to account for all SEUs that could produce functional errors.

Previous work [49] proposes to use the scheme with $\delta > 0$ for also detecting timing faults (which become predominant in nanometric technologies) and eventually combine the scheme with a retry approach for correcting the errors.

Recently, [51] and [52] proposed to reduce the power dissipation of a circuit by reducing the power supply at a level lower than the one required to guarantee correct operation under worst-case conditions. Since this reduction may induce timing errors under worst-case conditions, an error detection and correction scheme is proposed to cope with. It turns out that the proposed error detection principle is the same as the one shown in Fig. 8. In fact, this scheme uses a "shadow" latch (Latch1 in Fig. 8) clocked by a delayed clock and a comparator that compares the output of the shadow latch against the output of the functional latch. After error detection, the content of the "shadow" latch is used to restore the state of the erroneous functional latch. Thus, the shadow latch is used for both error detection and retry, resulting on hardware cost reduction. In addition, thanks to a smart implementation, the impact of the extra hardware on power and speed becomes negligible and a drastic reduction of the total circuit power dissipation is obtained. The proposed retry scheme works for timing errors as they do not affect the shadow latches (Latch1 in Fig. 8). This is because the shadow latches capture their inputs much later than the functional latches. Unfortunately, it does not work for SEUs and SETs since they may induce errors also on the shadow latches.

The authors experimented with this scheme in a 64-bit single-issue in-order processor using the Alpha instruction set, implemented in a Taiwan Semiconductor Manufacturing Company (TSMC) 0.18-μm process. By protecting 8% of flip-flops against timing errors, hence at a very low area cost, the authors obtained a power dissipation reduction ranging from 25% to 35%.

Another technique allowing detection of errors induced by transient faults is presented in [53]. It uses a transition detector to monitor the outputs of the combinational circuit. Faults in the transition detector are also detected. The scheme is efficient against SETs. It does not detect SEUs as the outputs of the latches are not monitored.

The time redundancy principle using duplicated latches/flip-flops was employed to implement two processors. The first processor ROCS81 [54] implements the time redundancy scheme in LEON [55], a 32-bit RISC processor developed by

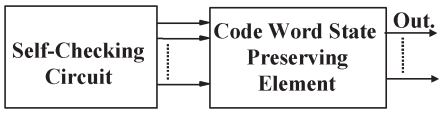


Fig. 9. SET-tolerant combinational circuit.

the European Space Agency (ESA). Radiation test results show [54] that the scheme allows complete protection of the logic against both SEUs and SETs.

A second design implements the time redundancy scheme together with an error recovery approach in an 8-bit low-power processor. The result is a fault-tolerant design able to detect errors and recover from them. The area overhead for the combined approach is 96%, while radiation tests show a complete protection against SETs and SEUs. A significant part of this overhead corresponds to circuitry added to allow error recovery for SEUs occurring in the latches during their eventual hold cycles. If we disregard hold cycles, detection and correction of timing faults and SETs as well as of SEUs affecting the latches at any other clock cycle can be achieved by less than 60% of the silicon area. This area overhead is expected to be lower for larger processors. Also, combinational blocks in processors often have a low ratio (total area)/#outputs since they use aggressive pipeline to achieve high speed. This is not advantageous for the time-redundancy approach of Fig. 8, which adds a flip-flop per combinational output. Thus, the area overhead for processor designs represents the worst case. It is therefore expected to be lower for other circuits.

4) Error Masking Combining Space and Time Redundancy: A scheme merging space and time redundancy to mask errors induced by SETs is presented in [49]. The idea is to combine self-checking design with time redundancy. Since self-checking design distinguish correct output values (code words) from incorrect output values (noncode words), we can use this capability to extract the correct values from the perturbed signals. For doing so, we can exploit the fact that SETs affect the outputs of the circuit only for short laps of time, resulting on a temporal occurrence of a noncode word. For the rest of the time, the outputs are correct (code words). In this situation, in order to extract the correct values from the perturbed output signals, the outputs of the self-checking circuit are monitored by an asynchronous sequential element (see Fig. 9) having the following properties.

1) The element produces on its output a determined value for each code word input.
2) The element preserves its previous output value for each noncode word input. Thus, if an error changes an input code word into a noncode word, the output value already produced by the code word input is preserved. Such an element will be referred as code word state preserving element.

Thanks to property 2), errors occurring on the self-checking circuit outputs after the output of the code word state preserving element reaches its correct value are masked, but errors occurring before this time may not be masked. As we discuss later, a careful implementation may mask SETs occurring at any time.

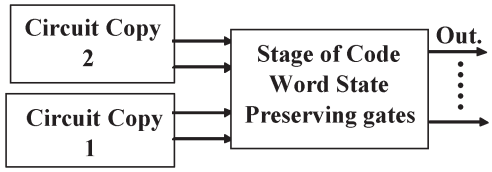


Fig. 10. Perturbation-tolerant circuit based on duplication.

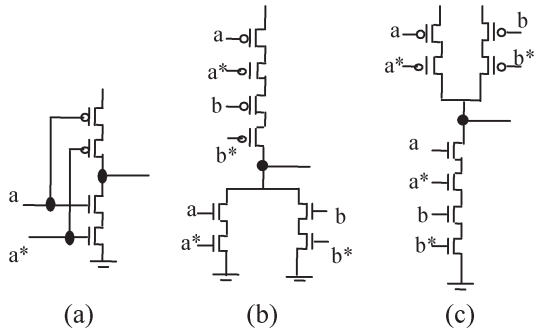


Fig. 11. Code word state preserving gates for the duplication code, realizing the inverter, NOR, and NAND functions. (a) Inverter. (b) NOR. (c) NAND.

The simplest implementation of this scheme uses a duplicated circuit and a code word state preserving element for the duplication code, as shown in Fig. 10. The code word state preserving element for this code can be implemented by the gate referred as "Inverter" in Fig. 11(a). This gate is also known as C-element. When the inputs of this gate are equal (code word input), the gate operates as an inverter. When these inputs are not equal (noncode word input), the output of the gate is in the high impedance state and preserves its previous value. Thus, the gate exhibits the properties of a code word state preserving element for the duplication code and the scheme of Fig. 10 tolerates transient faults.

Code word state preserving elements for the duplication code, realizing the function of any other gate, can be obtained by duplicating the transistors and the inputs of the gate, as illustrated in Fig. 11. These elements can be used to replace the latest stage of gates of the two circuit copies.

Let Dcw be the logic transition time of the code word state preserving gate. When a transient pulse affects the outputs of one circuit copy, the transient will be masked under one of the following conditions.

— The pulse disappears at an instant which precedes the end of the clock cycle by a time interval larger than Dcw. This time interval is sufficient to drive the output of the code word state preserving gate to its correct value before the end of the clock cycle. Thus, when the latching edge of the clock occurs, the circuit outputs are correct.
— Before the occurrence of the transient pulse, the outputs of the two circuit copies have correct values for a time period larger than Dcw. In this case, the outputs of the code word state preserving gates will be driven to their correct values. Then, if a transient pulse appears on the outputs of one of the two circuit copies, the correct output value of the code word state preserving gates will be preserved due to the property 2).

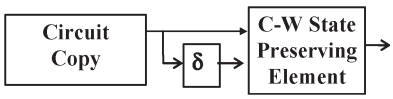


Fig. 12. SET masking using time redundancy.

Therefore, to guarantee that any transient pulse of duration Dtr is masked, we can allocate enough time to the circuit operation so that one of the above two cases always occurs. This will be guaranteed if after their transition within a clock cycle the outputs of the two circuit copies remain in their steady state for a time equal to $\mathrm{Dtr} + 2\mathrm{Dcw}$. This time interval determines the increase of the clock period that we have to pay in order to mask any transient of duration Dtr. In such a case, a latch/flip-flop placed on the output of the code word state preserving gate will never be impacted by SETs.

A finer analysis shows that only an increase of the clock cycle by $\mathrm{Dtr} + \mathrm{Dcw}$ is necessary. Indeed, if the duration of the transient pulse does not exceed Dtr, if Dcw^- is the time during which the outputs of the two circuit copies remain in their steady state before the occurrence of the transient pulse, and if Dcw^+ is the time during which the outputs of the two circuit copies remain in their steady state after the disappearance of the transient pulse, then we will have $\mathrm{Dcw}^- + \mathrm{Dcw}^+ \geq \mathrm{Dcw}$. In this case, the output of the code word state preserving gate will start its transition towards its new correct value, but because $\mathrm{Dcw}^- < \mathrm{Dcw}$, the transient appears before the transition is completed and the gate is driven in the high impedance state. Thus, its output preserves its latest electrical state. The transition continues for a time period Dcw^+ after the disappearance of the pulse. Since $\mathrm{Dcw}^- + \mathrm{Dcw}^+ \geq \mathrm{Dcw}$, the output of the code word state preserving gate will be driven to its new correct value.

The above designs do not offer protection against SEUs. To cope with these faults too, we can use the DICE cell (Fig. 5). This cell has the properties of the code word state preserving elements. Thus, if in Fig. 10 we use as code word state preserving element a latch/flip-flop designed according to the principle of the DICE cell, the design will be protected against both SETs and SEUs [49].

The scheme of Fig. 10 involves a high area overhead due to circuit duplication. To reduce this cost, we can trade hardware penalty with speed penalty [49], as shown in Fig. 12. In this figure, each output of the unique circuit copy feeds a pair of inputs of a code word state preserving element. The one input of this pair is driven by one output of the circuit and the other input of the pair is driven by a delayed copy of the same output. Thus, a transient pulse reaching an output of the circuit will reach the one input of the code word state preserving element immediately and the other input with a delay δ. If the duration of the transient pulse does not exceed δ, the transient pulse could not affect simultaneously both inputs of this element. Thus, the inputs of this element will be either correct or they will not belong to the duplication code. In the later case, the code word state preserving element will be in the high impedance state. The maximum time duration that it could be in this state is 2δ. Thus, if we increase the clock cycle by δ with respect to the clock cycle of the circuit of Fig. 10, we can mask any transient pulse whose duration does not exceed δ.

IV. Area Cost and Circuit Reuse

Standard duplication and TMR schemes involve a very high area overhead (more than 100% for the duplication and more than 200% for TMR). The various schemes presented in the previous sections require significantly lower hardware overhead. However, we cannot provide a precise value for this overhead as we can do for standard duplication or TMR since this overhead can vary significantly from one circuit to another. For instance, the time redundancy scheme of Fig. 8 induces a much smaller area overhead in the case of a multiplier than in the case of an adder (duplicating the output latches of a multiplier will represent a small percentage of the multiplier area). It is also much smaller in the case of conservative designs than in the case of fast processors using an aggressive pipeline. Indeed, in designs with aggressive pipeline, the flip-flops may occupy a larger area than the combinational logic, resulting in more than 50% extra area for the scheme of Fig. 8. It is also difficult to provide precise values for the area overhead related to the schemes of Figs. 5, 7, and 9 as it varies significantly from one circuit to another. In many cases, this overhead may exceed the 50% of the area of the unprotected design and may become a major obstacle to the adoption of these schemes. Circuit reuse may alleviate this obstacle. Some reuse examples are presented below.

A. Reusing Scan–Path Resources

Successful reuse examples of scan–path resources are presented in [56]. Some of scan–path schemes used by Intel designers associate a scan flip-flop (scan portion) to each functional flip-flop (system portion). During the test, both flip-flops are used, but during system operation only the functional flip-flop is used. This provides the opportunity to use the scan flip-flop for protecting the functional flip-flops against SEUs. For doing so, [56] uses during system operation the scan flip-flop as a shadow flip-flop, which uses the same clock and receives the same inputs as the functional flip-flop.

The first scheme detects SEUs affecting any flip-flop by using an XOR gate to compare the output of the system flip-flop against the output of the shadow flip-flop. The outputs of the XOR gates are compacted by an OR tree into a single error detection signal (self-checking design). To avoid using OR trees for compacting the outputs of the XOR gates into a single error indication signal, an error signaled on the output of the XOR gate is trapped by the shadow flip-flop by using a second XOR gate (error-trapping design). At prespecified recovery points, the shadow flip-flops are connected into a scan chain and their contents are shifted out. An error recovery is initiated if any trapped errors are discovered.

The third scheme adds a C-element [Fig. 10(a)] to the outputs of the system flip-flop and the shadow flip-flop. During system operation, SEUs affecting the system or the shadow flip-flop are masked by the C-element. This protection is not perfect since, from the discussion in Section III-B4 in relation with the

scheme of Figs. 10 and 11(a), we observe that errors affecting one or the other flip-flop before the C-element output reaches its correct value are not masked. However, according to [56], circuit simulations show a reduction of the SEU rate by a factor of 20. During the test phase, the C-element disturbs the proper use of the system and the scan flip-flops. To fix this problem, two transistors of the C-element are bypassed to convert it into an inverter. Thus, it provides on its output the inverse of the value stored in the system flip-flop.

Due to the reuse of the scan flip-flop as a shadow flip-flop, the area overhead of these solutions is very low. But this is only true if the designer decides to associate a distinct scan flip-flop to each system flip-flop for improving test and debug quality. But the common industrial practice today is to use a single flip-flop for both scan operation and system operation. However, the advantages of the combined scheme in terms of test quality, debug, and reliability may motivate a more frequent use in the future. For a design where 25% of flip-flops are protected by using one of the above schemes, [56] reports a less than 5% power overhead, 0.3% area overhead, and insignificant speed penalty.

The schemes in [56] consider flip-flop-based designs. Latch-based designs use transparent latches instead of flip-flops. These latches are rated by two non-overlapping clocks [57]. Transparent latches cannot be connected into scan paths since they do not dispose a master and a slave portion. Thus, for test purposes, a second transparent latch must be associated to each functional transparent latch in order to convert it during the test phase into a muster–slave scan latch. As the second latch is not used during system operation, it can be used as a shadow latch. A C-element added on the outputs of these latches allows masking SEUs. In this case, the output of the C-element reaches its correct value during the transparent phase of the latches, hence before the beginning of their blocked phase. Thus, the protection against SEUs is complete since an SEU is masked even if it occurs at the very beginning of the blocked phase of the latches.

The above schemes do not cover SETs. To cover both SETs and SEUs, we can convert during system operation the scan flip-flop into a shadow flip-flop, as in the scheme proposed in [56]. But in the present case, this flip-flop is rated by a delayed clock $(\mathrm{Ck} + \delta)$ to implement the time redundancy scheme of Fig. 8. Then, SETs and SEUs are detected by using a comparator to compare the outputs of the system flip-flops against the outputs of the shadow flip-flops. The error trapping design used in [56] can also be adopted here to avoid the use of the comparator.

V. Conclusion

As soft errors become a stringent reliability problem in nanometric technologies, design for soft error mitigation is gaining importance. As any fault tolerant approach, soft error mitigation may impact significantly the area, speed, and power, so the selection of the best approach is a complex tradeoff between these parameters and the target level of reliability. It is therefore suitable to dispose a variety of solutions that can help achieve various constraints of the design, such as minimal area, minimum power, minimum speed, maximum reliability, or a tradeoff between these parameters. We presented various schemes that exhibit such characteristics and could help the designer meet product requirements. Active work in this area in various research centers should further enrich the space of solutions and improve our ability to face the soft error threat.

References

[1] R. C. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test. Comput.*, vol. 22, no. 3, pp. 258–266, May/Jun. 2005.

[2] R. C. Baumann, T. Z. Hossain, S. Murata, and H. Kitagwava, "Boron compounds as a dominant source of alpha particles in semiconductor devices," in *Proc. 33rd Int. Reliability Physics Symp.*, Las Vegas, NV, 1995, pp. 297–302.

[3] M. Baze and S. Buchner, "Attenuation of single event induced pulses in CMOS combinational logic," *IEEE Trans. Nucl. Sci.*, vol. 44, no. 6, pp. 2217–2223, Dec. 1997.

[4] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," *IEEE Trans. Nucl. Sci.*, vol. 44, no. 6, pp. 2209–2216, Dec. 1997.

[5] J. Benedetto, P. Eaton, K. Avery, D. Mavis, T. Turflinger, P. Dodd, and G. Vizkelethyd, "Heavy ion induced single event transients in deep submicron processes," *IEEE Trans. Nucl. Sci.*, vol. 51, no. 6, pp. 3480–3485, Dec. 2004.

[6] M. Nicolaidis, "Electronic circuit assembly comprising at least one memory with error correcting means," French and U.S. Patents pending, filed July 2001.

[7] ——, "Data storage method with error correction," French and U.S. Patents pending, filed July 2002.

[8] F. Vargas and M. Nicolaidis, "SEU tolerant SRAM design based on current monitoring," in *Proc. 24th IEEE Int. Symp. Fault Tolerant Computing*, Austin, TX, Jun. 1994, pp. 106–115.

[9] T. Calin, F. Vargas, M. Nicolaidis, and R. Velazco, "A low-cost, highly reliable SEU-tolerant SRAM: Prototype and test results," *IEEE Trans. Nucl. Sci.*, vol. 42, no. 6, pp. 1592–1598, Dec. 1995.

[10] B. Gilles, M. Nicolaidis, F. G. Wolff, C. A Papachristou, and S. L. Garverick, "An efficient BICS design for SEUS detection and correction in semiconductor memories," in *Proc. Design Automation and Test Europe (DATE)*, Munich, Germany, Mar. 2004, pp. 592–597.

[11] L. Rockett, "An SEU hardened CMOS data latch design," *IEEE Trans. Nucl. Sci.*, vol. 35, no. 6, pp. 1682–1687, Dec. 1988.

[12] S. Whitaker, J. Canaris, and K. Liu, "SEU hardened memory cells for a CCSDS reed Solomon encoder," *IEEE Trans. Nucl. Sci.*, vol. NS-38, no. 6, pp. 1471–1477, Dec. 1991.

[13] D. Bessot and R. Velazco, "Design of SEU-hardened CMOS memory cells: The hit cell," in *Proc. 2nd European Conf. Radiation and Its Effects Components and Systems (RADECS)*, St. Malo, France, 1993, pp. 563–570.

[14] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron CMOS technology," *IEEE Trans. Nucl. Sci.*, vol. 43, no. 6, pp. 2874–2878, Dec. 1996.

[15] D. P. Siewiorek and R. S. Swwarz, *Reliable Computer Design and Evaluation*. Newton, MA: Digital, 1992.

[16] D. K. Pradhan, *Fault-Tolerant Computing System Design*. Upper Saddle River, NJ: Prentice-Hall, 1996.

[17] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. M. Sonza Reorda, and M. Violante, "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors," *IEEE Trans. Nucl. Sci.*, vol. 47, no. 6, pp. 2231–2236, Dec. 2000.

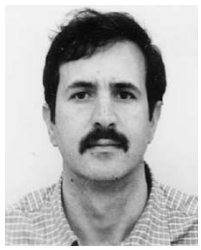
[18] N. Oh, S. Mitra, and E. J. McCluskey, "ED4I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, Feb. 2002.

[19] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Proc. 18th Int. Symp. Defect and Fault Tolerance VLSI Systems*, Cambridge, MA, Nov. 3–5, 2003, pp. 581–588.

[20] M. Rebaudengo, M. S. Reorda, and M. Violante, "A new approach to software-implemented fault tolerance," *J. Electron. Test., Theory Appl.*, no. 20, pp. 433–437, Aug. 2004, Kluwer Academic Publishers.

[21] M. Rebaudengo, M. S. Reorda, M. Violante, B. Nicolescu, and R. Velazco, "Coping with SEUs/SETs in microprocessors by means of low-cost solutions: A comparative study," *IEEE Trans. Nucl. Sci.*, vol. 49, no. 3, pp. 1491–1495, Jun. 2002.

[22] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 627–641, Jun. 1999.
[23] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Reliab.*, vol. 51, no. 2, pp. 111–122, Mar. 2002.
[24] S. Mukherjee and S. Reinhardt, "Transient fault detection via simultaneous multithreading," in *Proc. Int. Symp. Computer Architecture*, Vancouver, BC, Canada, Jun. 2000, pp. 25–36.
[25] P. N. Vijaykumar and I. Pomerantz *et al.*, "Transient fault recovery using simultaneous multithreading," in *Proc. Int. Symp. Computer Architecture*, Anchorage, AK, May 2002, pp. 87–98.
[26] S. Mukherjee, S. Kontz, and S. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. Int. Symp. Computer Architecture*, Alaska, AK, May 2002, pp. 99–110.
[27] K. De, C. Natarajan, D. Nair, and P. Banerjee, "RSYN: A system for automated synthesis of reliable multilevel circuits," *IEEE Trans. Very Large Scale (VLSI) Integr. Syst.*, vol. 2, no. 2, pp. 186–195, Jun. 1994.
[28] N. A. Touba and E. J. McCluskey, "Logic synthesis techniques for reduced area implementation of multilevel circuits with concurrent error detection," in *Proc. Int. Conf. Computer Aided Design*, San Jose, CA, 1994, pp. 651–654.
[29] M. Nicolaidis, R. O. Duarte, S. Manich, and J. Figueras, "Achieving fault secureness in parity prediction arithmetic operators," *IEEE Des. Test. Comput.*, vol. 14, no. 3, pp. 60–71, Apr.–Jun. 1997.
[30] M. Nicolaidis and R. O. Duarte, "Design of fault-secure parity-prediction booth multipliers," *IEEE Des. Test. Comput.*, vol. 16, no. 3, pp. 90–101, Aug. 1999.
[31] R. O. Duarte, M. Nicolaidis, H. Bederr, and Y. Zorian, "Efficient fault-secure shifter design," *J. Electron. Test., Theory Appl.*, vol. 12, no. 1–2, pp. 29–39, Feb. 1998.
[32] M. Nicolaidis, "Carry checking/parity prediction adders and ALUs," *IEEE Trans. Very Large Scale (VLSI) Integr. Syst.*, vol. 11, no. 1, pp. 121–128, Feb. 2003.
[33] V. Ocheretnij, M. Goessel, E. S. Sogomonyan, and D. Marienfeld, "A modulo p checked self-checking carry-select adder," in *Proc. 9th IEEE Int. On-line Testing Symp.*, Kos Island, Greece, Jul. 2003, pp. 25–29.
[34] V. Ocheretnij, D. Marienfeld, E. S. Sogomonyan, and M. Gossel, "Self-checking code-disjoint carry-select adder with low area overhead by use of add1-circuits," in *Proc. 10th IEEE Int. On-Line Testing Symp.*, Madeira, Portugal, Jul. 2004, pp. 31–36.
[35] C. V. Freiman, "Optimal error detection codes for completely asymmetric binary channels," *Inf. Control*, vol. 5, no. 1, pp. 64–71, Mar. 1962.
[36] J. M. Berger, "A note on error detection codes for asymmetric binary channels," *Inf. Control*, vol. 4, no. 1, pp. 68–73, Mar. 1961.
[37] J. E Smith, "The design of totally self-checking check circuits for a class of unordered codes," *J. Des. Autom. Fault-Toler. Comput.*, vol. 1, no. 4, pp. 321–343, Oct. 1977.
[38] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Trans. Comput.-Aided Des. Integr. Cicuits Syst.*, vol. 12, no. 6, pp. 878–887, Jun. 1993.
[39] W. W. Peterson, "On checking an adder," *IBM J. Res. Develop.*, vol. 2, no. 2, pp. 166–168, Apr. 1958.
[40] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*, 2nd ed. Cambridge, MA: MIT Press, 1972.
[41] A. Avizienis, "Arithmetic algorithms for error-coded operands," *IEEE Trans. Comput.*, vol. C-22, no. 6, pp. 567–572, Jun. 1973.
[42] T. R. N. Rao, *Error Coding for Arithmetic Processors*. New York: Academic, 1974.
[43] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Comput.*, vol. 423, no. 1, pp. 68–77, Jan. 1994.
[44] U. Sparmann, "On the check base selection problem for fast adders," in *Proc. 11th VLSI Test Symp.*, Atlantic City, NJ, Apr. 1993, pp. 62–65.
[45] U. Sparmann and S. M. Reddy, "On the effectiveness of residue code checking for parallel two's complement multipliers," in *Proc. 24th Fault Tolerant Computing Symp.*, Austin, TX, Jun. 1994, pp. 219–228.
[46] I. Alzaher Noufal and M. Nicolaidis, "A tool for automatic generation of self-checking multipliers based on residue arithmetic codes," in *Proc. Design, Automation and Test Europe Conf.*, Munich, Germany, Mar. 1999, p. 122.
[47] L. Anghel, M. Nicolaidis, and I. Alzaher Noufal, "Self-checking circuits versus realistic faults in very deep submicron," in *Proc. 18th IEEE VLSI Test Symp.*, Montreal, QC, Canada, Apr. 2000, pp. 55–66.
[48] D. Alexandrescu, L. Anghel, and M. Nicolaidis, "New methods for evaluating the impact of single event transients in VDSM ICs," in *Proc. IEEE Int. Symp. Defect and Fault Tolerance VLSI Systems*, Vancouver, BC, Canada, Nov. 2002, pp. 99–107.
[49] M. Nicolaidis, "Time redundancy based soft-error tolerant circuits to rescue very deep submicron," in *Proc. 17th IEEE VLSI Test Symp.*, Dana Point, CA, Apr. 1999, pp. 86–94.
[50] L. Anghel and M. Nicolaidis, "Cost reduction and evaluation of a temporary faults detecting technique," in *Proc. Design Automation and Test Europe Conf. (DATE)*, Paris, France, Mar. 2000, pp. 591–598.
[51] D. Ernst *et al.*, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Int. Symp. Microarchitecture*, San Diego, CA, Dec. 2003, pp. 7–18.
[52] ——, "Razor: Circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, Nov.–Dec. 2003.
[53] C. Metra, M. Favalli, and B. Ricco, "On-line detection of logic errors due to crosstalk, delay, and transient faults," in *Proc. Int. Test Conf.*, Washington, DC, Oct. 18–23, 1998, pp. 524–533.
[54] D. Chardonnereau, R. Keulen, M. Nicolaidis, E. Dupont, K. Torki, F. Faure, and R. Velazco, "Fault tolerant 32-bit RISC processor: Implementation and radiation test results," in *Single Event Effects Symp.*, Manhattan Beach, CA, Apr. 23–25, 2002.
[55] E.S.A.: European Space Agency. LEON designed by Giri Gaisler. [Online]. Available: http://www.gaisler.com
[56] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *Computer*, vol. 38, no. 2, pp. 43–52, Feb. 2005.
[57] M. Nicolaidis, "Ensemble de circuits électroniques protégé contre des perturbations transitoires," French patent application, filed Apr. 5, 2005.



Michael Nicolaidis (M'92) is a Research Director at the French National Research Center, Grenoble, France, and a Chief Technical Officer and Founder of iRoC Technologies, Santa Clara, CA, since January 2001. He was the Leader of the Reliable Integrated Systems Group at Techniques of Informatics and microelectronics for Computer Architecture (TIMA) Laboratory until December 31, 2000. His research interests include very large scale integration (VLSI) testing, discrete Fourier transform (DFT), on-line testing, fault tolerant design, reliability issues in very deep submicron technologies, and fault-tolerant approaches for nanotechnologies. He published more than 150 papers, edited one book and several journal special issues, and holds several patents.

Dr. Nicolaidis was the Program Chair or General Chair of the 1997, 1998, and 1999 IEEE VLSI Test Symposium, General Co-Chair of the IEEE International On-Line Testing Workshop from 1995 to 2002, General Co-Chair of the 2003, 2004, and 2005 IEEE International On-Line Testing Symposium, and Vice Chair of the IEEE Computer Society Test Technology Technical Council (TTTC). He received twice the Best Paper Award of the Design and Test in Europe Conference, the Best Paper Award of the IEEE VLSI Test Symposium, and the Meritorious Service Award of the IEEE Computer Society. He is a Golden Core Member of the IEEE Computer Society.