

Self-Stabilizing Microprocessor^{*}

Analyzing and Overcoming Soft-Errors (Extended Abstract)

Shlomi Dolev and Yinnon A. Haviv

Department of Computer Science, Ben-Gurion University of the Negev, Israel.
{dolev, haviv}@cs.bgu.ac.il

Abstract. Soft-errors are changes in memory value caused by cosmic rays. Decrease in computing features size, decrease in power usage and shorting the micro-cycle period, enhances the influence of soft-errors. Self-stabilizing systems is designed to be started in an arbitrary, possibly corrupted state, due to, say, soft errors, and to converge to a desired behavior. Self-stabilization is defined by the state space of the components, and essentially is a well founded, clearly defined, form of the terms: self-healing, automatic-recovery, automatic-repair, and autonomic-computing. To implement a self-stabilizing system one needs to ensure that the micro-processor that executes the program is self-stabilizing. The self-stabilizing microprocessor copes with any combination of soft errors, converging to perform fetch-decode-execute in fault free periods. Still, it is important that the micro-processor will avoid convergence periods as possible, by masking the effect of soft errors immediately. In this work we present design schemes for self-stabilizing microprocessor, and a new technique for analyzing the effect of soft errors. Previous schemes for analyzing the effect of soft errors were based on simulations. In contrast, our scheme computes lower bound on the micro-processor reliability and enables the micro-processor designer to evaluate the reliability of the design, and to identify reliability bottlenecks.

1 Introduction

The interest in robust systems increased dramatically during the last years. New terms and research directions such as automatic-recovery, self-healing, self-repairing and self-stabilization [1,13,3] are extensively explored by academia and industry. This is no coincidence, the design of critical systems such as computer aircraft control should be verified to be self-stabilizing. Otherwise, once the assumptions concerning the type and amount of failures are violated, the system may enter a state from which it will never recover.

Time and space redundancy are usually used to cope with fault prone environments. Error detection and correction (codes) are used to mask the faults in a

^{*} Partially supported by NSF Award CCR-0098305, IBM faculty award, STRIMM consortium, and Israel ministry of defense.

way that ensures input output relation requirements with high probability. Still soft-errors and other unpredictable faults may transfer the system to undesired states in which faults are not masked. Self-stabilization ensures that the system will eventually recover from such faults. Following a finite stabilization period the system will exhibit the desired relation of the inputs and the corresponding outputs streams.

Fault tolerance using space and/or time redundancy have been extensively studied. These approaches differ from the one we propose since they do not cope with the recovery from an arbitrary state. Our approach compliments the previous approaches: design your system to be started in an arbitrary state, so that even if the assumptions concerning its operation do not hold for a while the system will recover.

The basic assumption that the self-stabilizing algorithm designer uses is that the program that implements the algorithm is executed by the processor(s). This assumption should be examined. In fact, it is clear that current designs of microprocessors do not have the automatic recovery property, and hence the self-stabilizing program, that the microprocessor should execute, will not be executed, and obviously the system will not stabilize.

Next we elaborate on the different aspects that have to be addressed when designing a microprocessor that copes with soft error or transient faults, by (1) using a self-stabilizing design that ensures recovery following the occurrence of any combination of soft errors, namely recovery from arbitrary state, (2) reducing the probability of soft-errors to influence the computation, analyzing the fault masking capabilities, in order to achieve fault masking with high probability, and by doing so, (3) eliminate as much as possible (the fault non-masking) convergence periods (of the self-stabilizing microprocessor) that may cause higher level (self-stabilizing) algorithms to start (fault non-masking) convergence periods.

Soft Errors: Soft errors, also called single event upset (seu), are voltage changes caused by cosmic rays (or other disturbance); they can change the output value of a logical gate in the digital circuit. For our discussion we can model the behavior of a soft error as a single bit value change, from zero to one or vice versa, the assumption of any value change is different from the one way change assumed in [8]. The probability of a soft error increases when the feature size decreases, the voltage decreases and the micro-cycle time is shorten. Current technology considers soft errors in memory circuits by adding redundancy in the form of error correcting codes. The effect of soft errors in the logic circuit (alu, cpu, etc.) is not addressed, claiming that currently the probability for such a scenario is low (sizewise, memory devices contain much more targets for soft errors than logic circuits). Recent studies have shown that the probability of soft errors to influence the logic circuit will increase in less than a decade to the current probability for influencing memory. In addition, we note that the current typical number of internal registers of a cpu is large enough to require soft-error considerations.

Some tasks cannot allow even a small error probability. There is an extensive efforts for achieving robust design that can cope with soft errors. In particu-

lar, in on-going systems that never stop operating, such as systems of satellites, the eventual occurrence of such a fault is almost certain. Two bold examples for a robust system design are the IBM S-390 and the Compaq non-stop Himalaya server. Both architectures use a combination of space and information redundancy to try to mask the effect of soft-errors [17]. One approach is to use simultaneous multithreading processor that runs two copies of the same program, in order to increase the resiliency to soft-errors. This technique is called simultaneous and redundant thread [17,16]. The space and time redundancy solutions, using error correcting or recomputing, reduce the probability that the resulting computation will be corrupted but do not, and in fact may not, give a guarantee for masking the effect of soft-errors.

Self-Stabilization: Self-stabilization is an elegant approach for designing fault tolerant computing devices [1]. The idea is to explore the state space of the device, simply by considering any possible value for the bits in the memory and proving that from every such state the system eventually converges to the desired behavior. In other words, given a specification, we will say that an algorithm is a self-stabilizing algorithm for the given specification if: There exists a set of configurations, called *safe configurations*, that being in one of them ensures executions according to the specification, and this set of configurations is closed, being in a safe configuration ensures transition to a safe configuration. Starting from any configuration, a self-stabilizing algorithm must reach in finite time a safe configuration. Self-stabilizing algorithms can be combined in a way that the output of the first stabilizing algorithm serves the second stabilizing algorithm as an input. For example, self-stabilizing algorithm may assume correct behavior of the microprocessor and will stabilize after the (self-stabilizing) microprocessor will converge to its desired behavior.

Originally self-stabilizing algorithm were design to cope with transient faults, such as soft errors. The assumptions made by the designers of self-stabilizing algorithms are that the algorithm itself is not corrupted (one can assume that the algorithm is written in read only memory) and that it is *executed*. We examine, the assumption that the processor continues to execute the algorithm at any given time. When considering a micro-code controlled processor, we can imagine the case of this processor, getting into infinite loop in a subset of its micro-code, due to soft errors, and never reaching the micro-code part that evolves fetching a new command and executing it.

Analyzing Soft-Errors Influence: Von Neumann [11], and later Pippenger [14] suggested a model for a noisy computation of circuits. In this model a gate may fail with a specific probability. This model assumes that a gate computes it's output only once during the function computation, even if there are two paths of different length from the gate to one of the outputs of the circuit. This model does not fit the properties of single event upsets since seus are temporary faults which cause the gate to produce incorrect output for a short period of time. The probability of a seu hitting a gate to effect the computation is closely related to it's *computation crucial time* as we will define and compute.

We proposed an algorithm for analyzing the masking probability of a circuit that implements a boolean function. The algorithm results in a lower bound for the probability that the circuit computes correctly in the presence of soft errors. To the best of our knowledge, up to now, only simulations were used for analyzing the soft error resiliency of circuits [6,12,10,9]. Our approach, as opposed to simulations, gives the designer knowledge on which gates in the circuit are the problematic ones.

The rest of the paper is organized as follows. Two methodologies for designing self-stabilizing microprocessors appear in Section 2, the first requires detailed examination of the microprocessor circuit, while the second proposes an additional device to enforce stabilization. In Section 3 we present technique for analyzing the masking probability of a given circuit, as a function of the soft-errors probability. Concluding remarks appear in Section 4. Most of the proofs are omitted from this extended abstract.

2 Methodologies for Designing Self-Stabilizing Microprocessors

The configurations (or states) space of a microprocessor includes the value of every register of the microprocessor, including the micro program counter, or any other internal control variable of the microprocessor (e.g., pipeline control). An *input output configuration* of a microprocessor at a particular (clock) pulse is the binary value of the microprocessors pins when the pulse takes place. The *input and output stream* of a microprocessor is defined by a sequence of input and output configurations, such that every two successive configurations are related to two successive pulses, in an obvious way. We define a *legal behavior* of a microprocessor by every input output stream that correspond to an input output stream that starts in the (manufacturer) predefined initial state and handles a sequence of machine-code commands (in the set of the commands that are allowed by the manufacturer). We also include any suffix of input output stream to be in the set of the legal behaviors.

We next observe that the contents of the non-control microprocessors registers (including the program counter of the machine code) are in fact part of the state of the (machine code) program and therefore should be handled by designing the (machine-code) program to cope with every possible state, namely to be self-stabilizing. Thus, we only concern ourselves with the control variables of the microprocessor and requires that they will lead to eventual execution of the commands of the stabilizing (machine-code) program. The eventual execution of the machine-code commands is in fact an execution of fetch-decode-execute cycle. Thus, our legal behavior set includes execution that starts in every possible values of the non-control variables of the microprocessor, and proceeds according to the (manufacturer) definition of the machine-code commands execution.

A microprocessor is self-stabilizing iff every behavior that starts in any configuration reaches in a finite number of pulses a *safe configuration* after which its behavior is in the set of legal-behaviors.

In other words we require that a safe configuration is reached, and then the microprocessor will repeatedly execute fetch-decode-execute, where each machine code command is *executed according to the manufacturer specifications*. The manufacturer manual defines the specification of each machine code command. It is possible, as we will show in the sequel, that the microprocessor will be proven to stabilize for a certain specifications of a machine code and not stabilize for other specifications of the machine code. For example, whether the specification exposes the user to the internal registers used to implement the stack operations. In the rest of this section we consider the simplified case in which the criteria for a microprocessor to be self-stabilizing is a repeated execution of fetch-decode-execute sequence, where the execution is according to the microprocessor arbitrary internal state. Similar technique can be used for specific designs, monitoring and verifying that the execution of the commands is according to the manufacturer specifications.

2.1 Involved, State Diagram Method

Our first suggestion is to examine the designed microprocessor and verify whether it is self-stabilizing or not. To demonstrate the importance of the (internal) microprocessor control variables, we examine a micro-code controlled processor. We note that the same technique can be applied to any other microprocessor control method, considering the specific (internal) control variables used by the microprocessor. Our goal is to prove that there is no cycle of microinstruction executions that does not include a fetch-decode, and properly execution of the machine command pointed by the program counter. Usually this problem requires explicitly generating transition graph of the micro processor (which is too large to compute). Our scheme avoids this by using an *abstraction* of the transition graph. Given a micro-code of a processor, we convert it to a finite state representation. Where a node in the representation is defined by the (control variables, e.g., the) micro-code program counter value. In fact, every node in our finite state machine, represents a set of all possible microprocessor *refined states* that have the same micro-code program counter value. Where a refined state includes specific values for all the registers, in particular the non-control registers. The edges between the nodes of the finite state representation, represents transitions in the granularity of microinstruction execution. There is an edge between every two nodes, i and j , such that there is a transition (due to the occurrence of clock pulse) from a refined microprocessor state represented by the node i to a refined microprocessor state represented by j .

Once the finite state representation is constructed we can validate its stabilization property. We search for a state transition cycle that does not contain a fetch-decode-execute sequence. First we note that every state in the finite state representation of a self-stabilizing microprocessor must have an out-degree (of non self-loop edges) of at least one, otherwise the microprocessor can be driven (by soft-errors) into a configuration, where no change to the micro program counter is possible and therefore no fetch-decode-execute sequence is performed.

Then we can check whether, all cycles in the finite state representation include a fetch-decode-execute sequence. In such a case, we can conclude that the microprocessor is self-stabilizing. This test can be performed by executing a depth first search over the finite state representation. Otherwise, we will consider every cycle that does not include the fetch-decode-execute sequence in details. Namely, we will look whether the edges of the cycle are due to a possible transition of the microprocessor, considering refined states.

We use the above method to verify that the micro-code controlled processor Mic-1 presented at [18] (chapter 4) is a self-stabilizing microprocessor for certain machine code specifications and is not for others. We have succeeded proving stabilization of the Mic-1 microprocessor in the case in which the definitions of the machine code commands, that refer the top of stack register *TOS*, are allowed to return any value as long as it is not a value pushed to the stack following stabilization. On the other hand we proved that Mic-1 is not stabilizing (it is only pseudo-stabilizing) when we require that the machine code commands will be executed as if the *TOS* is the value of the top of stack address in the memory.

2.2 Blackbox, Watchdog Method

One can ensure that a fetch-decode-execute sequence is eventually executed by using an upper bound on the number of clock pulse that may pass in between every two successive executions of the fetch-decode-execute sequence. We assume that every processor repeatedly executes the fetch-decode-execute sequence when it is started in a predefined state (e.g., the initial state defined by the manufacturer). Thus, one can use a watchdog circuit that will detect the situation in which the processor has not executed the fetch-decode-execute sequence for a period longer than the given upper bound. In such a case, the watchdog reset the microprocessor to the predefined (initial) state. Note, that the (re)activation of the reset will occur only due to (additional) soft errors.

The watchdog circuit itself may experience soft-errors. Fortunately, it is possible to ensure that the watchdog circuit is self-stabilizing. One can implement the watchdog as a counter that is decremented in every clock pulse, using the exact number of bits needed to count the upper bound on the number of pulses in between two successive fetches. We assume that the watchdog counter can be initialized in any possible state (due to a soft error) causing in the worst case, an immature reset of the microprocessor. A self-stabilizing microprocessor recovers following the occurrence of faults that derive its state to an arbitrary state. During its automatic recovery the processor converges to a legal behavior (for example, a behavior that can be achieved from its predefined initial configuration). Naturally, the influence of soft-errors are not masked (immediately after an arbitrary state is reached, and) during the convergence period. Thus, (self-stabilizing) programs that the microprocessor executes may lose their consistency and will have to start a convergence period themselves. In [2] it was suggested to use Markov chains, to compute the probability to be in a safe state, when exits from safe states are possible. We enhance this approach by suggesting to measure the expected execution period length in which, a system that is in a

safe configuration, does not leave the set of safe configurations (in the presence of faults such as soft-errors). We use the term *legal execution period* for such execution period. In particular, we would like a self-stabilizing algorithm that is executed by our self-stabilizing microprocessor to have the longest expected *legal execution period*, which in turn implies high fault masking capabilities.

Next we analyze the masking probability of a circuit in order to enable the circuit designer to evaluate its soft-errors masking probability, identify resiliency bottlenecks, and modify the design if needed.

3 Analyzing Soft-Error Masking Probability

The self-stabilizing property of the microprocessor is essential as a fall-back mechanism that ensures automatic recovery, still it could (and should) be combined with techniques that mask faults with high-probability. The combination of self-stabilization with fault masking capabilities ensures high probability for tolerating faults with no output influence (masking faults) and automatic recovery following (a bursty) occurrence of faults that the processor cannot mask. Note that it is impossible to mask all combinations of transient errors, in particular the combination that changes the outputs of all the gates.

Let f be a function with n_{in} input bits and n_{out} output bits. We assume that the circuit implementing f has n_{in} input (one bit) latches and n_{out} output (one bit) latches. We present a method for estimating the probability that the circuit that implements f causes the outputs latches to store (the vector) $f(x)$ when (the vector) x is the current (fixed) values of the input latches. Given a circuit that implements f , we consider the *computation dag* of the circuit, $G_f = (V, E)$ to be the directed acyclic graph defined as follows: $V = V_{input} \cup V_{logic} \cup V_{output}$, where $v \in V_{input}$ represents an input latch, $v \in V_{logic}$ represents a gate in the circuit, and $v \in V_{output}$ represents an output latch. The edges of the *computation dag* are defined by $E = \{(u, v) \mid \text{the output of } u \text{ is wired into the input of } v\}$.

The input degree of every node v , $InDegree(v)$ is bounded by the maximal fan-in of a gate, $MaxDegree$. Note that $\forall v \in V_{input} InDegree(v) = 0$, because v represents an input latch. Similarly, $\forall v \in V_{output} OutDegree(v) = 0$, since v represents an output latch. In addition, the number of gates in the circuit is $n = |V_{logic}|$. We use *delay* for denoting the propagation time of signals in gates, that is, the time it takes for a particular gate to compute and output the result that reflects the current inputs of the gate. Our time axis origin (time 0) will be the first time the output latches are opened for writing, and we mark by ℓ the duration that the output latches should receive a stable (and correct) input, in order to store a correct result. In such a case, we say that the output latches are enabled for writing during the time interval $[0, \ell]$.

Definition 1. *We define the event the circuit computes correctly to be the event in which the output latches are disabled for writing, when their contents are the values defined by applying f to the contents of the input latches. We use p_f to denote the probability for such an event.*

Our analysis is based on understanding the locations and times in which the circuit is vulnerable to seu. Given a gate or a latch u in the implementation of f , the t time point will be considered an *input crucial time* for u iff at least one of the followings hold: (1) u is an output latch and $t \in [0..\ell]$ (2) There is a gate or a latch v s.t. $t + delay$ is an input crucial time in v and one of it's inputs is wired to the output of u . Intuitively, the time in which a gate or a latch is input crucial, is the time it is important that this gate receives a correct input.

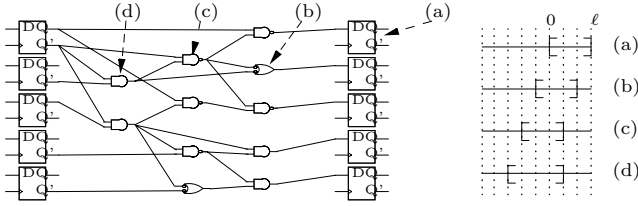


Fig. 1. The input crucial time of circuit elements is depicted on the right. A gap between two dotted lines represent *delay* time (e.g., $\ell = 3 \cdot delay$)

Definition 2. Given a gate g in the implementation of f , the t point of time is considered a computation crucial time for g iff the interval $[t - delay, t]$ overlaps input crucial time of g .

Intuitively, if it is crucial that a gate g receives correct input at time t then it is crucial that the computation of the gate will be correct at time t but also *delay* time later, when the (“last”) signal propagates through the gate. A seu hitting the circuit influences the correct evaluation of a certain gate for certain period. If this period does not overlap with the computation crucial time of the gate then the circuit will compute correctly. We will use this observation to compute a (lower) bound on the probability that the circuit computes correctly.

Next we present an algorithm to compute, ICT_i , the input crucial time, for each gate and latch represented by a node v_i . Later we will use the output of this algorithm to compute the computation crucial time of v_i for $v_i \in V_{logic}$ and then to compute a lower bound for p_f .

3.1 Calculating Input Crucial Time

An algorithm for computing ICT_i for all $v_i \in V$ is presented at Figure 2. Step (3) of the algorithm scans all the nodes (nodes represent gates) according to a topological sort of the computation dag, computing at each node, the input crucial time of the gate or latch that it represents. Figure 3 and Figure 4 describe an efficient implementation of step 3 of the algorithm. Step 3 of the algorithm computes ICT_i also for $v_i \in V_{input}$, that is, nodes that represent input latches. In the case where the input to the circuit does not come from a latch, ICT_i of $v_i \in V_{input}$ represents the time that it is crucial that the input to the circuit will be correct (offseted by *delay*).

Input: $G_f = (V_{input} \cup V_{logic} \cup V_{output}, E)$, $delay, \ell$
 Output: $\{ICT_i | v_i \in V_{logic} \cup V_{input}\}$ /* the input crucial times of each logic gate and input latch*/

- (1) Compute a topological sort of G_f
 where V_{input} (V_{output}) are the first (last, respectively).
- (2) **For each** $v_i \in V_{output}$ **do**:
 - (2.a) $ICT_i \leftarrow \{[0, \ell]\}$
- (3) **For each** $v_i \in (V_{logic} \cup V_{input})$ in decreasing order of topological sort **do**:
 - (3.a) $ICT_i \leftarrow \bigcup_{(i,j) \in E} \{t - delay | t \in ICT_j\}$

Fig. 2. Abstract presentation of the algorithm for computing input crucial time

Lemma 1. *The set ICT_i computed in Figure 2 is the set of input crucial time of the gate or input latch that is represented by the node v_i in the computation-dag.*

We use the fan-out limitations of a circuit to conclude that the number of edges of a computation-dag $|E|$ is in $O(n)$. The depth of the circuit, $depth$ is, $max\{|P(v_i, v_j)| \mid v_i \in V_{input}, v_j \in V_{output}\}$. Where $P(v_i, v_j)$ is a path in G_f connecting v_i to v_j , and $|P(v_i, v_j)|$ is the number of edges in that path. Note that the depth of every circuit cannot exceed the number of gates, (every path is a simple path), and hence $depth \in O(n)$, but is usually much less.

It turned out that the implementation of step (3) of the algorithm in Figure 2 influences the complexity of the crucial times computation. We propose representing the time periods as time segments as detailed in Figures 3 and 4.

Data structure: $\forall v_i \in V$, we hold ICT_i as a list of disjoint closed intervals ordered by the starting time of the interval, that is $C_i = \{\langle start_i^j, end_i^j \rangle\}$, where $\forall i, j : start_i^j \leq end_i^j$ and $\forall i, j_1, j_2, j_1 < j_2 : end_i^{j_1} < start_i^{j_2}$.

Comment: $ICT_j - delay$ in line (3.a) should be applied to both the start and end times, namely replacing every $\langle start, end \rangle \in ICT_j$ by $\langle start - delay, end - delay \rangle$.

- (3) **For each** $v_i \in (V_{logic} \cup V_{input})$ in decreasing order of topological sort **do**:
 - (3.a) $ICTs \leftarrow \{ICT_j - delay \mid \langle v_i, v_j \rangle \in E\}$
 - (3.b) $ICT_i \leftarrow ICTs[1]$
 - (3.c) **For** $2 \leq i \leq |ICTs|$ **do**
 - (3.c.1) $ICT_i \leftarrow MergeCrucialTimes(ICT_i, ICTs[i])$

Fig. 3. Detailed description of step (3) of Figure 2

Lemma 2. *The overall duration of input crucial times that can exist in a set ICT_i of node v_i cannot exceed $(depth \cdot delay) + \ell$.*

Input: Two lists, ICT_1, ICT_2 , of disjoint closed intervals ordered by the starting time of each interval.

Output: ICT , a list of disjoint closed intervals ordered by the starting time of each interval, holding the union of time in ICT_1 and ICT_2 .

```

(1)  $ICT \leftarrow \emptyset$ 
(2) While  $((ICT_1 \neq \emptyset) \vee (ICT_2 \neq \emptyset))$  do
    (2.a) Pull  $\langle start, end \rangle$ , interval with the first starting time from  $ICT_1$  or  $ICT_2$ .
    (2.b) Let  $\langle start', end' \rangle$  be the last interval in  $ICT$ . if  $start \leq end'$  then
        (2.b.1)  $end' \leftarrow \max(end', end)$ 
    (2.c) else
        (2.c.1) Add  $\langle start, end \rangle$  to  $ICT$ 

```

Fig. 4. MergeCrucialTimes routine

Lemma 3. *If ICT_i holds the input crucial times as a set of maximal continues time intervals, then for every interval $\langle start, end \rangle \in ICT_i$ it holds: (a) $end - start \geq \ell$, (b) $start \in \{-(i \cdot delay) | i \in \mathbb{N}\}$*

Lemma 4. *If the crucial time units ICT_i of a node $v_i \in V$ are represented as a set of maximal continues time intervals, then the number of intervals in ICT_i , denoted $|ICT_i|$, is in $O(\text{depth})$.*

Lemma 5. *The time complexity of the MergeCrucialTimes routine described at Figure 4 is $O(|ICT_1| + |ICT_2|)$.*

Lemma 6. *The time complexity of an iteration in step (3) for node v_i is $O(\text{OutDegree}(v_i) \cdot \text{depth})$.*

Lemma 7. *The time complexity of step (3) of the algorithm is $O(n \cdot \text{depth})$.*

Theorem 1. *The time complexity of the algorithm is $O(n \cdot \text{depth})$.*

3.2 Computing a Lower Bound for Correct Computation

p_f denotes the probability that a circuit computes correctly as defined in Definition 1. In this section we use the output of the algorithm presented in Section 3.1 to compute (a lower bound on) p_f .

Definition 3. *A computation of a function is single event upset free (seu free), if there exist no gate g and time t for which: (a) g is computation crucial at time t , and (b) a seu causes g to compute incorrectly during time t .*

Let e_f be the event in which a computation is seu free. For every event e , we denote the probability of e to occur by $Pr(e)$. It holds that $p_f \geq Pr(e_f)$ because in the case in which all the gates computes correctly during their computation crucial times, the output of the circuit is correct. This is an inequality due to the possibility that a logical masking will occur (that is, incorrect result of a gate may not necessarily imply incorrect result of the output, e.g., [14]).

Definition 4. We say that a computation of a function is a single event upset free (seu free) with respect to a gate g , if there exist no time t for which g is computation crucial at time t , and a seu caused g to compute incorrect result at time t .

Given a function implementation with n gates g_1, \dots, g_n . We define e_i to be the event in which the computation of the function is seu free with respect to g_i . We assume that any two events e_i and e_j , $i \neq j$ are independent. Therefore the following holds $Pr(e_f) = \prod_{i=1}^n (Pr(e_i))$. Replacing $Pr(e_i)$ by p_i we have $p_f \geq \prod_{i=1}^n p_i$. We show how to compute p_i for every gate g_i using ICT_i , the set of input crucial time units of g_i . In doing so we consider the physical characteristics of the influence of seus on the particular circuit technology. Recall that transient-faults (in the form of single event upsets) are electrical pulses caused by an ionized alpha particle hitting a gate. The result of such a hit is a pulse of electrical current in a typical shape (see Figure 5 which depicts the data given in [7]). The shape of the pulse depends not only on the technology used (e.g., packaging material) but also on the energy of the particle hitting the gate. Physical experiments relate a particle of a certain energy hitting a gate to specific pulse characteristics [12,10,9,6]. Physical experiments also supply us with the probability that a particle with an energy level Q (causing a pulse with a specific shape, for a given technology) hits a gate during a specified time period. The electrical pulse created by the seu damages the computation made by the gate it hits only when it exceeds a certain current threshold. This threshold can be determined given the technology of the circuit (e.g., process technology, voltage levels). Therefore, it is common to model the electrical pulse of a seu as a *logical pulse*. That is, a rectangle (or a step function) pulse with only two possible values, 0 and 1. The logical pulse is 0 whenever the electrical pulse does not exceed the gate's threshold and 1 when it exceeds it. The simulations of the effect of soft errors proposed by iRoC technologies [6], uses a similar model when injecting single event upsets into the simulated circuit.

Given ICT_i the input crucial time of a gate g_i , it is easy to compute CCT_i , its computation crucial time according to Definition 2. First we present a method to determine p_i from CCT_i , the crucial time units of the gate, assuming that all particles contain the same energy (thus causing an identical electrical pulse). Next, we will improve the method by considering particles of different energies. **Fixed duration of pulses:** In this method we consider all particles to contain the same energy. One way of doing so is by averaging the energy of particles with energy that causes a pulse that exceeds the gate's threshold (according to their occurrence probability). Thus the pulse can be modeled as logical pulses with constant duration (defined per a given technology). Denote by d_{seu} , the

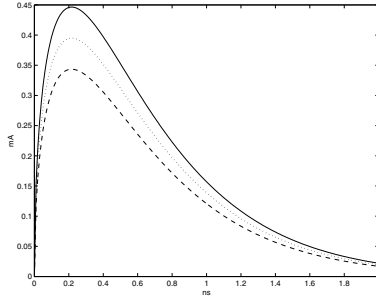


Fig. 5. The current over time of a particle hit (particles with different energies)

duration of the logical pulse. If t is a computation crucial time of a gate g_i , then any particle with an average energy, hitting g_i , causing a logical pulse which starts anywhere in the range $[t - d_{seu}, t]$ will cause a disruption in a the computation made by g_i at time t . For each $1 \leq i \leq n$, we denote \widehat{CCT}_i to be: $\bigcup_{t \in CCT_i} \{[t - d_{seu}, t]\}$. \widehat{CCT}_i is the time in which, if particle pulse begins in, then the gate g_i may have an incorrect output that will reach the output latches of the circuit at the time the latches store the circuit result. Figure 6 depicts an example of computing \widehat{CCT}_i given the set of computation critical time of g_i , CCT_i . The set \widehat{CCT}_i can be easily computed by extending each maximal continues duration in CCT_i to the left, by d_{seu} , and merging when needed.

Denote by $q_{seu}(d)$ the probability that a particle (with an energy level sufficient to create a pulse that exceeds the threshold of a gate) hits a certain gate during d time. Our bound for p_i in this case will be: $\prod_{(start, end) \in \widehat{CCT}_i} 1 - q_{seu}(end - start)$. Note that when no particle hits the gate during a duration equal by its size to \widehat{CCT}_i , non of the logical pulses overlap with the computation crucial times of the gate (implying that g_i does not violate the computation seu free property).

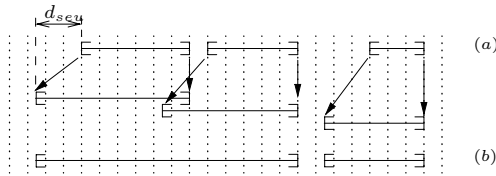


Fig. 6. Example of calculating time in which logic pulses may influence computation result (a) The computation crucial time periods, CCT_i (b) The periods in which logic pulses must not start in, \widehat{CCT}_i .

The above may not be accurate when particles contain different energy levels. To gain further accuracy, we propose considering the probabilistic distribution on the energy of the particles.

Different pulse durations: Given a gate threshold, we divide the particles hitting the gate into classes according to the duration of the logical pulse they cause. Particles with energy causing a logical pulse with a duration of $((D - 1)\delta, D\delta]$ will be in the D 'th class. The smaller value we choose for δ the more accurate our analysis will be. Since the duration of the logical pulse is a monotonic function of the energy of particle creating it, the classes are simply a division into continuous energy levels. We denote by $q_{seu}^D(d)$ the probability that during time period of length d , our gate will be hit by a particle of the D 'th class which will cause a logical pulse with a duration in the range $((D - 1)\delta, D\delta]$ time. Let Δ be the maximal duration of the logical pulse caused by a particle. Then we have $\lceil \frac{\Delta}{\delta} \rceil$ functions, one for each class of particles. Our analysis is by a simple reduction to the case of fixed duration of logical pulses. For each $1 \leq k \leq \lceil \frac{\Delta}{\delta} \rceil$ we will compute the probability that our gate crucial computation does not overlap with a logical pulse caused by particles of the k class, in doing so we will assign $q_{seu} := q_{seu}^D$ and $d_{seu} := k$. We bound p_f by the product of the results.

3.3 Logical Masking Analysis

First we will prove that that analysis of logical masking is an NP-complete problem then we will present techniques for analyzing limited (yet, important) cases for majority circuits.

Complexity of Logical Masking Problems

Definition 5. Given a circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}$, containing gates g_1, \dots, g_k , a function $q : \{g_1, \dots, g_k\} \rightarrow (0.5, 1]$ and an input $x \in \{0, 1\}^n$, we define $FT_{C,q}(x)$ to be the probability that C computes correctly on input x when the gate g_i ($1 \leq i \leq k$) of C computes correctly with probability $q(g_i)$.

Definition 6. Given a circuit $C : \{0, 1\}^n \rightarrow \{0, 1\}$, containing gates g_1, \dots, g_k , and a function $q : \{g_1, \dots, g_k\} \rightarrow (0.5, 1]$, we define $FT_{C,q} = \min\{FT_{C,q}(x) | x \in \{0, 1\}^n\}$. That is, the minimal resiliency of C with respect to q .

We show that unless $P = NP$, there is no polynomial algorithm for computing $FT_{C,q}$ even for the case in which C is a formula [19]. Specifically, we show that if such an algorithm exists, we can determine in polynomial time whether a formula ψ is satisfiable. Given a formula ψ , we create a formula Φ as depicted in Figure 7. The formula Φ consists of three copies of the formula ψ and two AND gates wired as depicted.

We choose $q(a) = q(b) = \frac{3}{4}$ (where a and b are the two AND gates depicted in figure 7), and $q(g) = 1$ for all other gates g . We will determine whether ψ is satisfiable using the observation that the probability that Φ computes correctly on input x ($FT_{\Phi,q}(x)$) is closely related to whether or not x satisfies ψ .

When Φ computes on input $x \in \{0, 1\}^n$ such that $\psi(x) = 0$, one of the inputs to the AND gate b is always 0 (with probability 1). Thus, the probability that the output of b will be correct is $\frac{3}{4}$, and we have $FT_{\Phi,q}(x) = \frac{3}{4}$.

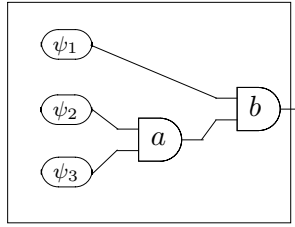


Fig. 7. The formula Φ as a function of ψ . ψ_i ($i=1,2,3$) are three independent copies of ψ .

When Φ computes on input $x \in \{0,1\}^n$ such that $\psi(x) = 1$, the input to the AND gate a is always $\langle 1, 1 \rangle$ (with probability 1) and the probability that the output of gate a is 1 is $\frac{3}{4}$. The probability that the AND gate b will produce 1 (the correct result) is composed of two situations, the first is when the input to gate b is $\langle 1, 1 \rangle$ and the gate b computes correctly, and the second is when the input of b is not $\langle 1, 1 \rangle$ and b does not compute correctly (error cancelation). Thus, in the case of x that satisfies ψ , we get that $FT_{\Phi,q}(x) = (\frac{3}{4})^2 + (1 - \frac{3}{4})^2 = \frac{5}{4}$.

In the case ψ is satisfiable, there exist $x \in \{0,1\}^n$ such that $\psi(x) = 1$ and $FT_{\Phi,q}$, the minimal resiliency of Φ with respect to q will be $\min(\frac{3}{4}, \frac{5}{8}) = \frac{5}{8}$. In the case ψ is not satisfiable, for all $x \in \{0,1\}^n$, $\psi(x) = 0$ and $FT_{\Phi,q} = \frac{3}{4}$.

Definition 7. Given a formula $\psi : \{0,1\}^n \rightarrow \{0,1\}$, containing gates g_1, \dots, g_k , a function $q : \{g_1, \dots, g_k\} \rightarrow (0.5, 1]$, and a constant $Q \in (0, 1)$ we define the Formula-Resiliency problem to be the problem of determining whether there exists $x \in \{0,1\}^n$ such that $FT_{\psi,q}(x) < Q$.

Theorem 2. The Formula-Resiliency problem is NP-Complete.

Proof. The Formula-Resiliency problem is NP-Hard: the proof is given by the reduction from SAT given above.

The Formula-Resiliency problem is in NP: The nondeterministic algorithm will guess $x \in \{0,1\}^n$, compute $FT_{\psi,q}(x)$ (described below), if the result is less than Q , return true, otherwise return false. Computing $FT_{\psi,q}(x)$ is done by recursively computing the probability that the output of a gate is 1. Given an input $x \in \{0,1\}^n$ we mark by $P_{\psi,q,x} : \{x_1, \dots, x_n, g_1, \dots, g_k\} \rightarrow [0, 1]$ a function mapping each gate and input of the formula to the probability that this gate/input will produce the value 1. Therefor, for inputs x_i s.t. $x_i = 1$ we get $P_{\psi,q,x}(x_i) = 1$ and for inputs x_i s.t. $x_i = 0$ we get $P_{\psi,q,x}(x_i) = 0$. For gates g_i , the function $P_{\psi,q,x}$ is computed recursively as follows: if g_i is an AND gate and it's inputs are A and B , with $P_{\psi,q,x}(A) = q_A$ and $P_{\psi,q,x}(B) = q_B$, then $P_{\psi,q,x}(g_i) = [(q_A \cdot q_B) \cdot q(g_i)] + [(1 - (q_A \cdot q_B)) \cdot (1 - q(g_i))]$. The above is due to the fact that an AND gate will produce 1 in two cases, the first is when it receives the input $\langle 1, 1 \rangle$ and computes correctly, and the second is when receiving another input and failing to compute correctly. Similarly, in the case the gate g_i is an OR gate, we get $P_{\psi,q,x}(g_i) = ((1 - q_A) \cdot (1 - q_B)) \cdot (1 - q(g_i)) + ((1 - ((1 - q_A) \cdot (1 - q_B))) \cdot q(g_i))$.

Denote g_o the output gate of the circuit, then $FT_{\psi,q}(x)$ will be either $P_{\psi,q,x}(g_o)$, when the correct output is 1, or $(1 - P_{\psi,q,x}(g_o))$, when the correct output is 0.

Majority circuits: Most error resilient circuitry employ some sort of error correction code (information redundancy) for masking errors in memory [4]. The common scheme used to mask errors in logic circuitry is using three or more parallel copies of the circuit and using a voter (majority circuit) to determine the correct result [4]. The voter in this scheme can be considered a decoder from an encoding in which the redundant word is a simple copy of the non-redundant word. The correctness of the computation made by the decoder is vital for the correctness of the scheme. We introduce a model for the definition of logical masking of functions that may receive incorrect input due to soft errors. Specifically we demonstrate the model on the majority function on three inputs I_1, I_2, I_3 . We use a discrete model for time, where each time unit represent a duration of *delay*, the propagation time in a gate. That is, time unit t represent the duration $[t \cdot \text{delay}, (t+1) \cdot \text{delay}]$. It turned out that adding timing information to the description allows better analysis and lower bounds for the probability that the majority circuit will be able to mask transient faults, in particular it is possible that there is no (fixed) majority during the input crucial times of the majority circuit (detailed are omitted from this extended abstract).

4 Concluding Remarks

In this paper we build a solid ground for executing self-stabilizing programs (in fact any program) by presenting a design for a self-stabilizing microprocessor. We have presented an accurate and efficient method for computing the probability of correct computation in the presence of soft errors. The analysis is algorithmic and is not based on simulations. Our method can serve the circuit designer to modify the design in order to avoid (weak) points of failures. In addition, we suggest a way to analyze logical masking using the (natural) example of a majority circuit. The goal is a processor that masks soft-errors with high probability and, has a fall back mechanism, in the form of its self-stabilization property, to automatically recover in severe cases in which soft-errors are not masked.

Acknowledgment. It is a pleasure to thank Amos Beimel and Enav Weinreb for helpful remarks and pointers to relevant literature.

References

1. S. Dolev, *Self-Stabilization*, MIT Press, 2000.
2. S. Dolev, Herman, T., "Dijkstra's Self-Stabilizing Algorithms in Unsupportive Environments" *WSS01*, LNCS:2194, pp. 67-81, 2001.
3. A. Fox and D. Patterson. "Self-Repairing Computers", *Scientific American*, June, 2003

4. C. N. Hadjicostis, *Coding Approaches to Fault Tolerance in Combinational and Dynamic Systems*, Kluwer, 2002.
5. J. L. Hennessey and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2002.
6. iRoC Technologies RobanTM, white papers. <http://www.iroctech.com>.
7. M. Kistler, P. Shivakumar, L. Alvisi, D. Burger, and S. Keckler. "Modeling the effect of technology trends on the soft error rate of combinational logic". In *ICDSN*, volume 72 of *LNCSE*, pages 216–226, 2002.
8. K. L. Parag, *Self-Checking and Fault-Tolerant Digital Design*, Morgan Kaufmann, 2001.
9. F. Lima, S. Rezgui, L. Carro, R. Velazco, R. Reis "On the use of VHDL Simulation and Emulation to Derive Error Rates". *Radiation Effects on Components and Systems Conference (RADECS), Grenoble, FRANCE*, 2001.
10. P. C. Murley and G. R. Srinivasan. "Soft-error Monte Carlo modeling program, SEMM". *IBM Journal of Research and Development*, vol. 40, Number 1 pp. 109–118, 1996.
11. J. von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components", *Automata Studies*, C. E. Shannon and J. McCarthy Eds., Princeton University Press, pp. 329-378, 1956.
12. E. Normand, "Single Event Upset at Ground Level," *IEEE Transactions on Nuclear Science*, vol. 43, pp. 2742–2751, 1996
13. D. Patterson. Recovery Oriented Computing. <http://www.cs.berkeley.edu/>.
14. N. Pippenger, "On networks of noisy gates", *Proc. of the 26th IEEE Symposium on Foundations of Computer Science*, pp. 30-36, 1985.
15. N. Pippenger, "Analysis of error correction by majority voting". *Advances in Computing Research*, Volume 5, JAI Press, pages 171–198, 1989.
16. S. K. Reinhardt and S. S. Mukherjee. "Transient fault detection via simultaneous multithreading". *ISCA*, pages 25–36, 2000.
17. E. Rotenberg. AR-SMT: "A microarchitectural approach to fault tolerance in microprocessors". *Symposium on Fault-Tolerant Computing*, pp. 84–91, 1999.
18. A. Tanenbaum. *Structured computer organization*, Prentice-Hall, 1984.
19. H. Vollmer. *Introduction to Circuit Complexity*. Springer-Verlag, Inc., 1999.