Hindawi
Scientific Programming
Volume 2018, Article ID 7306837, 16 pages
https://doi.org/10.1155/2018/7306837

*Research Article*

# SEDC-Based Hardware-Level Fault Tolerance and Fault Secure Checker Design for Big Data and Cloud Computing

**Zahid Ali Siddiqui**,[1] **Jeong-A Lee**,[2] **and Unsang Park**[1]

[1]*Department of Computer Science and Engineering, Sogang University, 35 Baekbeom-ro, Mapo-gu, Seoul 04107, Republic of Korea*
[2]*Department of Computer Engineering, Chosun University, 309 Pilmun-daero, Dong-gu, Gwangju 61452, Republic of Korea*

Correspondence should be addressed to Unsang Park; unsangpark@sogang.ac.kr

Received 15 December 2017; Revised 15 March 2018; Accepted 3 April 2018; Published 7 June 2018

Academic Editor: Shangguang Wang

Copyright © 2018 Zahid Ali Siddiqui et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Fault tolerance is of great importance for big data systems. Although several software-based application-level techniques exist for fault security in big data systems, there is a potential research space at the hardware level. Big data needs to be processed inexpensively and efficiently, for which traditional hardware architectures are, although adequate, not optimum for this purpose. In this paper, we propose a hardware-level fault tolerance scheme for big data and cloud computing that can be used with the existing software-level fault tolerance for improving the overall performance of the systems. The proposed scheme uses the concurrent error detection (CED) method to detect hardware-level faults, with the help of Scalable Error Detecting Codes (SEDC) and its checker. SEDC is an all unidirectional error detection (AUED) technique capable of detecting multiple unidirectional errors. The SEDC scheme exploits data segmentation and parallel encoding features for assigning code words. Consequently, the SEDC scheme can be scaled to any binary data length "n" with constant latency and less complexity, compared to other AUED schemes, hence making it a perfect candidate for use in big data processing hardware. We also present a novel area, delay, and power efficient, scalable fault secure checker design based on SEDC. In order to show the effectiveness of our scheme, we (1) compared the cost of hardware-based fault tolerance with an existing software-based fault tolerance technique used in HDFS and (2) compared the performance of the proposed checker in terms of area, speed, and power dissipation with the famous Berger code and m-out-of-2m code checkers. The experimental results show that (1) the proposed SEDC-based hardware-level fault tolerance scheme significantly reduces the average cost associated with software-based fault tolerance in a big data application, and (2) the proposed fault secure checker outperforms the state-of-the-art checkers in terms of area, delay, and power dissipation.

## 1. Introduction

Big data is promising for business applications and is rapidly increasing as an important segment of the IT industry. Big data has also opened doors of significant interest in various fields, including remote healthcare, telebanking, social networking services (SNS), and satellite imaging [1]. Failures in many of these systems may represent significant economic or market share loss and negatively affect an organization's reputation [2]. Hence, it is always intended that whenever a fault occurs, the damage done should be within an acceptable threshold rather than beginning the whole task from scratch, due to which fault tolerance becomes an integral part in cloud computing and big data [3]. Fault tolerance prevents a computer or network device from failing in the event of an unexpected error [2]. A recent study [4] showed that the cost of fault tolerance in cloud applications with high probability of failure and network latency is around 5% for the range of application sizes, hence providing improved performance at a lower cost.

The fault tolerance schemes in popular big data frameworks like Hadoop and MongoDB are composed of some sort of data replication or redundancy [5, 6]. MongoDB replicates its primary data in secondary devices. In a faulty event, the data is recalled from the secondary or the secondary temporarily acts as a primary. Fault tolerance in Hadoop relies on multiple copies of data stored on different data nodes. Although replication schemes allow complete data recovery,

they consume a lot of memory and communication resources. Hence, in recent years, many researchers have proposed fault tolerance algorithms for improved data recovery, effective fault detection, and reduced latency in big data and cloud computing [2, 5–10]. All of which detect fault at the software (SW) level. Even though faults propagated due to transient errors in hardware are also detected by these schemes, and software-based techniques are more flexible, the amount of data required to process to detect a fault costs a lot more than hardware- (HW-) based fault tolerance schemes. A recent study [11] investigated the cause of data corruption in a Hadoop Distributed File System (HDFS) and found that when processing uploaded files, HW errors such as disk failure and bit-flips in processor and memory generate exceptions that are difficult to handle properly. Liu et al. [7] implemented some level of HW-based fault tolerance by modelling CPU temperature to anticipate a deteriorating physical machine. Liu et al. [7] proposed the CPU temperature monitoring as an essential step for preventing machine failure due to overheating as well as for improving the data center's energy efficiency.

Parker [12] discussed how in many cases the faults are a direct consequence of tightly integrating digital and physical components into a single unit at a sensor or field node. In fact, many modern systems rely so heavily on digital technology that the reliability of the system cannot be decomposed and partitioned into physical and SW components due to interactions between them. There is a cost associated with the storage, transmission, and analysis of these higher-dimensional data. Furthermore, many of the SW-based approaches are simulation intensive, which may lead to broad implementation challenges. To overcome some of these challenges, he suggested that onboard, embedded processing will be a practical requirement.

Transient errors in HW, if propagated, may cause chain reaction of errors at the SW layer, causing potential failure at the node/server level. Detection at the HW level requires less computation time (as low as single clock cycle) as compared with detection at the SW level (several machine cycles), while a simple recovery mechanism called recomputation at the HW level can save a lot of data swapping and signaling at the SW level. As discussed in [13], big data has created opportunities for semiconductor companies to develop more sophisticated systems to cover the challenges faced in big data and cloud computing, and a trend towards integration of more functions onto a single piece of silicon is likely to continue. Also, due to advances in semiconductor processing, there has been a reduction in the cost of digital components [12]. For these reasons, we propose the detection of transient faults, as they occur in HW, through a HW-based fault tolerance scheme, while the SW-based fault tolerance stays at the top level as a second check for HW errors and first check for SW errors. As a result, the transient errors that arise in HW are mostly taken care of by lightweight processing at the HW level with little overhead (in terms of area, power, and delay), saving tremendous computation resources at the system level. The potential for catastrophic consequences in big data systems justify the overhead incurred due to HW-based fault tolerance method.

On the other hand, fault tolerance has also become an integral part of very large-scale integration (VLSI) circuits, where downsized, large-scaled, and low-power VLSI systems are prone to transient faults [14]. Transient faults or soft errors are transient-induced events on memory and logic circuits caused by the striking of rays emitted from an IC package and high energy alpha particles from cosmic rays [14–18]. Also, in multilevel cell memories like NAND Flash memories, these errors are mostly caused by cell-to-cell interference and data retention errors [19]. Physical protection such as shielding, temperature control, and grounding circuits are not always feasible; hence, in such cases, concurrent error detecting (CED) methods are employed for protection against these errors. Since CED circuits add to the overall area and delay of the system, the selection of appropriate error detecting, and even error correcting, circuits for a particular application leads to an efficient design [18]. It has been reported that the biggest portion of errors that occur in VLSI circuits and memories are related to unidirectional errors (UE) [19–21] because these errors shift threshold voltage levels to either the positive or negative side [22], causing the circuit node logic from "0" to "1" or from "1" to "0," but not both at the same time.

Many all unidirectional error detection (AUED) schemes have been proposed and implemented, among which the Berger code technique [23] is agreed to be the least redundant. With the ability to detect single- as well as multiple-bit unidirectional errors, this technique provides error detection by simply summing the logic 0's (a B0 scheme) or 1's (a B1 scheme) in the information word, expressing its sum in binary. If the information word contains "n-bits," then a Berger code will require $\lceil \log_2(n + 1) \rceil$-bits. A Berger code checker employs a 0's (or 1's) counter circuitry for reencoding the information word to check bits and then compares it with the preencoded check bits using a two-rail checker [23]. A chain of adders and a tree of two-rail checkers are required to design these checker circuits [23], where area and latency increase drastically as data length increases.

An m-out-of-n code is one in which all valid code words have exactly "m" 1's and "n-m" 0's. These codes can also detect all unidirectional errors when n = 2m. This condition not only increases the code size, but also the checker's area. Cellular realization of an m-out-of-2m code circuit was deemed by Lala [24] as more area- and delay-efficient than the previous implementations.

Given the importance of fault tolerance at the HW level in big data and cloud computing applications, in this paper, we present a fault secure (FS) SEDC checker used with SEDC codes [25]. An FS checker has the ability to safely hide or self-check (detect) its own faults as they occur in its circuitry. The SEDC partitions the input data into smaller segments (2, 3, and 4 bits) and encodes them in parallel. This unique scaling feature makes the system faster and less complex to design for any binary data length. The FS SEDC checker inherits all these features of SEDC codes (i.e., simple scalability, constant latency, and less power dissipation) which suits its implementation in online fault detection in processors, cache memories, and NAND Flash-based memories for big data

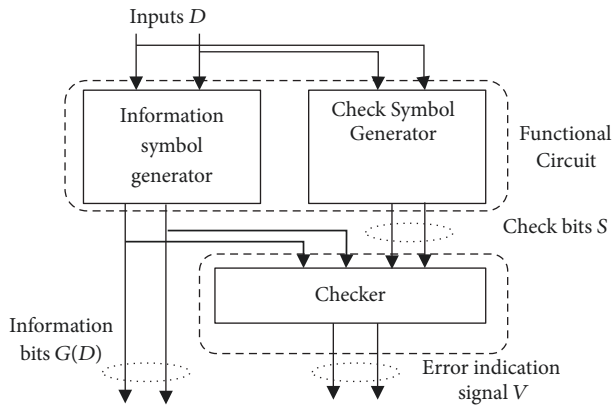



Figure 1: Block diagram of the proposed hardware-level fault tolerance system.

applications. The major contributions of this paper are as follows:

(1) We propose HW-level fault tolerance for circuits designed to process big data and cloud computing applications.

(2) In order to show the effectiveness of the proposed HW-level fault tolerance scheme in a big data scenario, we compare the cost associated with and without the proposed fault tolerance scheme and present results that show a significant reduction in the overall cost of fault tolerance in big data when the proposed HW-based fault tolerance scheme is applied.

(3) We also present a novel FS SEDC checker for use with SEDC-based HW-level fault tolerance systems.

(4) In order to prove the superiority of the FS SEDC checker presented in contrast with state-of-the-art AUED checkers, we show that the FS SEDC checker achieves state-of-the-art performance in terms of area, delay, and power dissipation.

The rest of the paper is organized as follows. We present an overall system diagram of the proposed HW-level fault tolerance system in Section 2. We give a brief mathematical foundation of the SEDC scheme and an example to encode logical circuits using SEDC in Section 3. Design details of the FS SEDC checker are described in Section 4. The proposed checker is shown to be FS through the fault testing methods; and its area, delay, and power comparison with state-of-the-art are derived in Section 5. We compute the fault coverage of the proposed SEDC-based fault tolerance system and present the experimental details and results in Section 5. To show the effectiveness of the proposed method in big data and cloud computation applications, we also perform a cost-performance analysis of fault tolerance at the SW level versus HW level in Section 5. Finally, we conclude the paper in Section 6.

## 2. Introduction to the Overall System

Figure 1 shows the main components of an error detecting codes based HW-level fault tolerance. The functional circuit consists of two subcircuits: an information symbol generator (ISG) and a check symbol generator (CSG). These two circuits do not share any logic. The ISG takes input $\mathbf{D}$ and performs some operation $G$ and produces output $G(\mathbf{D})$. The CSG is a carefully chosen logic function that acts as the encoder and generates check bits $\mathbf{S}$ using the same input $\mathbf{D}$, such that $\mathbf{S} = \phi(G(\mathbf{D}))$, where $\phi$ denotes the particular coding function. The checker normally contains another encoder that reencodes the information bits $G(\mathbf{D})$ into $\mathbf{S}' = \phi(G(\mathbf{D}))$ and then compares both $\mathbf{S}$ and $\mathbf{S}'$. A mismatch between $\mathbf{S}$ and $\mathbf{S}'$ is treated as an error, which is indicated by the error indication or verification signal $\mathbf{V}$.

The checker shown in Figure 1 plays a vital role in the overall fault tolerance system. The checker must exhibit a self-checking property or failsafe property to make sure that the whole system is fault secure (FS). If the checker is both self-checking and failsafe, the overall system is said to be as totally self-checking (TSC). In order to formally define these properties, let us consider the output of the functional circuit shown in Figure 1 to be represented by $G(\mathbf{D}) = G(x, f)$, where $x$ is the input and $f$ is the fault, and then in fault-free operation, i.e., $f = \emptyset$, the output can be represented by $G(x, \emptyset)$. Also, consider the input code space $\mathbf{D} \subseteq X$, output code space $\mathbf{S} \subseteq Y$, and an assumed fault set $\mathbf{F}$; then according to the definition of totally self-checking (TSC), $G$ is

(1) self-testing if for each fault $f$ in $\mathbf{F}$ there exists at least one input code $\mathbf{d} \in \mathbf{D}$ that produces a noncode output; i.e., $\forall f \in \mathbf{F}\, \exists\, \mathbf{d} \in \mathbf{D} \ni G(\mathbf{e}, f) \notin \mathbf{S}$,

(2) fault secure (FS) if for all faults $f$ in $\mathbf{F}$, and all code inputs $\mathbf{d} \in \mathbf{D}$, the output is either correct or is a noncode word; i.e., $\forall f \in \mathbf{F}$ *and* $\forall \mathbf{d} \in \mathbf{D}$, $G(\mathbf{e}, f) = G(\mathbf{e}, \emptyset)$ or $G(\mathbf{e}, f) \notin \mathbf{S}$.

In the proposed SEDC-based HW-level fault tolerance system, the CSG circuit is realized by an SEDC check symbol generator (SCSG) circuit, which generates the SEDC code words corresponding to the information bits $G(\mathbf{D})$. We presented a realization of an SEDC encoded SCSG circuit in [27], i.e., an SEDC encoded arithmetic logic unit (ALU) of a microprocessor. The SEDC encoded ALU circuit (SCSG) computes the SEDC codes corresponding to the output of the

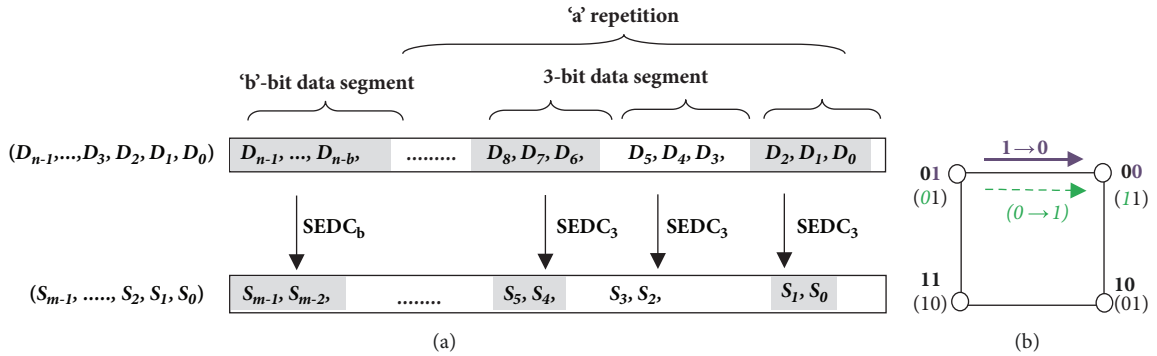



FIGURE 2: (a) SEDC scheme for given data word, and (b) 2D illustration of $SEDC_2$ scheme.

ISG (in [27] a normal ALU). Any fault that causes multiple unidirectional errors at the output of the normal ALU is detected by the SEDC checker. Any logic circuitry including SRAM-based memory cells [28] can be made fault tolerant by encoding them similar to the methods given in [27, 28]. In the next section we briefly introduce the SEDC scheme with an example to encode an adder circuit, while in the rest of the paper we focus on the proposed FS SEDC checker that can be used with any SEDC-based HW-level fault tolerance system.

## 3. Scalable Error Detection Coding (SEDC) Scheme

The Scalable Error Detection Coding scheme [25] is an AUED scheme formulated and designed in such a way that only the resultant circuit area is scaled, while its latency depends on a small portion of the input data (explained later).

For any binary data **D** of length $n$-bits represented as $(D_{n-1}, \ldots, D_2, D_1, D_0)$ with $D_i \in \{0, 1\}$ for $0 \leq i \leq n-1$, two parameters, $a$ and $b$, are computed using

$$a = \frac{n - \max(b)}{3} \tag{1}$$

where parameter $a$ can only take a positive integer value, i.e., $a \in \mathbb{Z}^+$, and parameter $b \in \{2, 3, 4\}$. Satisfying the condition for parameter$a$, the maximum possible value for parameter $b$ is selected. The SEDC code word **S** is represented as $(S_{m-1}, \ldots, S_j, \ldots, S_2, S_1, S_0)$ with $S_j \in \{0, 1\}$ for $0 \leq j \leq m-1$, where $m$ denotes the length of the SEDC code word and is computed by

$$m = \lceil \log_2 (n + 1 - 3a) \rceil + 2a \tag{2}$$

After computing the values for parameters $a$ and $b$, the SEDC code **S** for binary data **D** is computed. SEDC is designed to generate codes basically for 2-, 3-, and 4-bit data and is accordingly referred to as the $SEDC_2$, $SEDC_3$, and $SEDC_4$ scheme, respectively. It is then extended for any integer values of $n$, as shown in Figure 2(a).

*3.1. $SEDC_2$ Code.* A two-dimensional (2D) illustration of a 2-bit SEDC ($SEDC_2$) scheme is shown in Figure 2(b), where nodes represent data words, and their corresponding code words are written in brackets.

The SEDC coding scheme assigns code words to different data words with a unique criterion. Whenever there is a change of a bit (or bits) in a data word from "1" → "0," as shown with a bold arrow in Figure 2(b), the change is reflected in the code word in the opposite way; i.e., the code changes from "0" → "1," as shown with the dashed arrow in Figure 2(b), and vice versa. Equation (3) is used to assign 2-bit code words $S_1S_0$ to the 2-bit data words $D_1D_0$. Clearly, we can interchange the bit positions of $S_1$ and $S_0$ for another variant of $SEDC_2$ codes. This will not affect the code characteristics.

$$\begin{aligned}[S_1 : S_0] &= SEDC_2(D_1, D_0) \\ &= [XNOR(D_1, D_0) : NAND(D_1, D_0)]\end{aligned} \tag{3}$$

In (3), $[S_1 : S_0]$ represent the concatenated SEDC code bits, *XNOR* and *NAND* are the logical operations, and $SEDC_2$ is the basic coding scheme.

*3.2. $SEDC_3$ Code.* $SEDC_3$ code for 3-bit data is computed using (4) as follows:

$$\begin{aligned}[S_1 : S_0] &= SEDC_3(D_2, D_1, D_0) \\ &= \begin{cases} SEDC_2(D_1, D_0), & if\ D_2 = 0 \\ \overline{SEDC_2(\overline{D_1}, \overline{D_0})}, & if\ D_2 = 1 \end{cases}\end{aligned} \tag{4}$$

where the bar sign (e.g., $\overline{D_1}$) in (4) represents the logical NOT operation.

Figure 3 shows a 3D cube, illustrating the unidirectional error detection mechanism of $SEDC_3$ codes. The same notations are used in Figure 3 as in Figure 2(b). The dashed side of the cube represents the embedded $SEDC_2$ coding scheme in $SEDC_3$. Note that when there is a 2-bit unidirectional change in data word "001" → "111" (the two MSBs changing from "00" → "11"), the code changes in the opposite direction (the least significant bit of the code changes from "1" → "0"). In a similar way, the $SEDC_n$ scheme detects $n$-bit or all unidirectional errors in the data word **D**.

*3.3. SEDC$_4$ Code.* A SEDC$_4$ code for 4-bit data is formulated by (5) as follows:

$$\begin{aligned}\left[S_2 : (S_1, S_0)\right] &= SEDC_4\left(D_3, D_2, D_1, D_0\right) \\ &= \left[\overline{D_3} : SEDC_3\left(D_2, D_1, D_0\right)\right]\end{aligned} \tag{5}$$

The MSB of the code word is completely dependent upon the MSB of the data word for SEDC$_4$; hence, any change in the MSB of the data word is detected. The rest of the three data bits are encoded using the same SEDC$_3$ scheme.

It can be observed from (3), (4), and (5) that the SEDC$_2$ is embedded in 3-bit SEDC (SEDC$_3$) and consequently in 4-bit SEDC (SEDC$_4$) to detect all unidirectional errors in 3-bit and 4-bit data, as shown later. This ability to scale codes is not present in any other concurrent error detecting (CED) coding scheme.

In general, for SEDC$_n$, the $n$-bit binary data is grouped into one $b$-bit segment and the $a$ number of 3-bit segments, and then these segments are encoded using one SEDC$_b$ and $a$ number/numbers of SEDC$_3$ modules in parallel, as shown in Figure 2(a). It is noteworthy that each group of data segments and corresponding code segments is independent of each other. This independence makes our scheme scalable and able to detect some portion of bidirectional errors (BE) (discussed in Section 5.3).

If we interchange $S_1$ and $S_0$ for SEDC$_3$ in Figure 3, the corresponding SEDC$_3$ code is equal to Berger codes for a 3-bit segment, but our way of deriving the SEDC$_3$ code is a lot different from that of Berger codes. SEDC$_3$ codes are basically scaled from SEDC$_2$ codes, and SEDC$_2$ codes have no commonality with 2-bit Berger codes.

*3.4. SEDC-Based HW-Level Fault Tolerance System Example.* In order to illustrate the designing of a HW-level fault tolerance system using the SEDC scheme, we take the example of a 4-bit adder. Let us consider that this 4-bit adder is a part of a processor which processes big data applications, and we want to make this 4-bit adder fault tolerant against transient errors that arise in its circuitry, so the general HW-level fault tolerance system diagram shown in Figure 1 will be converted to the one shown in Figure 4. As shown in Figure 4, the 4-bit adder acts as an ISG and its equivalent SEDC encoder acts as a CSG. The SEDC encoder or CSG can be implemented using (6) as follows:

$$\left[S_3 : S_0\right] = SEDC\left(\mathbf{A}_{[3:0]} + \mathbf{B}_{[3:0]} + C_{in}\right) \tag{6}$$

As the output of 4-bit adder is a 5-bit value, hence the equivalent SEDC code has a 4-bit value according to (2). We used Altera's Quartus II software to synthesize the 4-bit adder (ISG), SEDC encoder (CSG), and the SEDC checker shown in Figure 4 and utilized the synthesized circuit for computing the fault coverage of the SEDC scheme, which is presented in Section 5.3. In the next section, we present the proposed FS SEDC checker, which completes the overall proposed SEDC-based HW-level fault tolerance system.

Table 1: Code table for FS SEDC$_1$ checker.

| $G_0$ | $S_0$ | $V_1$ | $V_0$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

## 4. The FS SEDC Checker

As shown in Figure 4, the FS SEDC checker takes $n$-information bits and $m$-SEDC check bits from the functional unit. The FS SEDC checker is also composed of one $b$-bit FS SEDC checker and $a$ sets of 3-bit FS SEDC checkers. With 1-, 2-, and 3-bit FS SEDC checkers, the output can be directly used as an error indication signal, but for $n > 3$, one level of wired-AND-OR logic gates is used to combine all the output of subblocks of FS SEDC checkers and generate the 2-bit error indication signal. Subsections discuss logic and circuit diagrams for primitive FS SEDC checkers (SEDC$_1$, SEDC$_2$, SEDC$_3$, and SEDC$_4$ checkers) which can be used to scale the SEDC checker to an $n$-bit FS SEDC checker (i.e., an FS SEDC$_n$ checker).

*4.1. The FS SEDC$_1$ Checker.* Table 1 shows the logic for a 1-bit SEDC (FS SEDC$_1$) checker. The valid input code words are "10" and "01" and the valid output code word is "10". $G_0$ denotes the 1-bit information word that is the output of ISG, and $S_0$ denotes the 1-bit SEDC check bit generated by the SEDC check symbol generator (SCSG). $V_1V_0$ is the 2-bit error indication signal of the FS SEDC$_1$ checker. $V_1$ and $V_0$ signals are generated by the circuits shown in Figure 5(a).

*4.2. The FS SEDC$_2$ Checker*

$$\begin{aligned}\left[V_1 : V_0\right] = \Big[&\overline{S_1.\left(G_1 + G_0\right)\left(S_0 + G_1G_0\right)} : \\ &\overline{\left(G_1 + G_0 + S_0\right)\left(S_1 + G_1G_0S_0\right)}\Big]\end{aligned} \tag{7}$$

In Figure 5, the symbols P1-P13 and N1-N13 represent the PMOS and NMOS transistors, respectively, and Vss represents the voltage supply. For simplicity, we used the CMOS-based implementation of SEDC checker circuits. Any other technology can be used to design these circuits, but the underlying algorithm, i.e., SEDC, will remain the same.

*4.3. The FS SEDC$_3$ Checker.* Figure 6(a) shows the block diagram and the logic for a 3-bit FS SEDC checker. Three-bit data $G_2G_1G_0$ from the ISG and 2-bit SEDC check bits $S_1S_0$ from the SCSG are first converted to $G_1'G_0'$ and $S_1'S_0'$, respectively, and then are checked using the same 2-bit FS SEDC checker, as shown in Figure 6(a). When the $G_2$ bit is "1," $G_1G_0$ and $S_1S_0$ are inverted, whereas if $G_2$ is "0," then $G_1G_0$ and $S_1S_0$ remain the same. As the outputs of the XOR gates are fed to the FS SEDC$_2$ checker, any error in the XOR gates is detected. This makes the overall 3-bit SEDC checker FS.

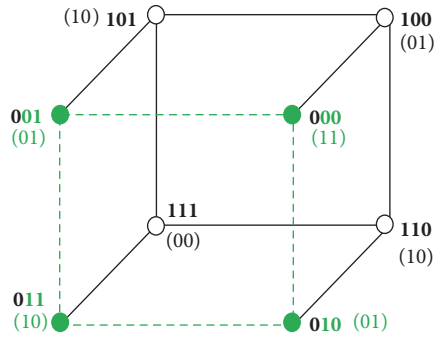



Figure 3: 3D illustration of $SEDC_3$ scheme.

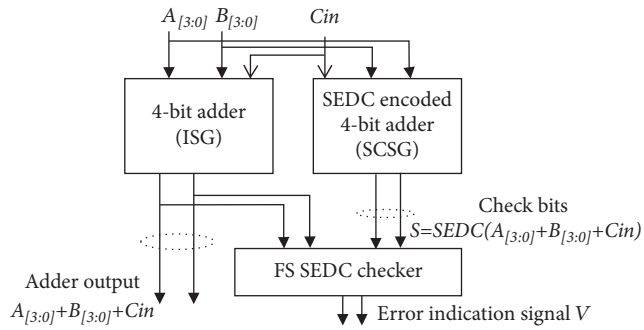



Figure 4: Example of SEDC-based HW-level fault tolerance system.

*4.4. The FS $SEDC_4$ Checker.* A 4-bit FS SEDC checker consists of one FS $SEDC_1$ checker and one FS $SEDC_3$ checker, as shown in Figure 6(b). Both $SEDC_1$ and $SEDC_3$ checkers generate 2-bit output $V_1V_0$. Because the valid code word is "10," to make sure that both checker units generate the "10" output during error-free operation, we "AND" the $V_1$ output-bit of the FS $SEDC_1$ checker with the $V_1$ output-bit of the FS $SEDC_3$ checker. Also, we "OR" the $V_0$ output-bits of both FS SEDC checkers using wired logic gates. We checked and confirmed by fault simulation that wired-AND and wired-OR gates are also FS for single faults (stuck-at-0, stuck-at-1, transistor-stuck-on, and transistor-stuck-off).

*4.5. The FS $SEDC_n$ Checker.* Like the SEDC code generator, the FS SEDC checker also consists of multiple 1-, 2-, and 3-bit FS SEDC checkers, depending upon the value of $a$ and $b$ from (1). For example, if $n = 8$ bits, then (1) $\Rightarrow a = 2$ and $b = 2$. This requires one FS $SEDC_2$ checker and two FS $SEDC_3$ checkers to realize an 8-bit FS SEDC checker.

The area of wired-AND-OR gates will also definitely increase as $n$ is increased. Figure 7 shows the block diagram of an $n$-bit FS SEDC checker. For $n = 8$ bits, there will be total of three FS SEDC checkers, each with 2-bit output; hence, a 3-input wired-AND and a 3-input wired-OR gate is required to compare all $V_1$ and $V_0$ bits. In general, for $n$-bit input, there are "$a + 1$" FS SEDC checkers, each with 2-bit output. So we require "$k = 2 \times (a + 1)$"-input wired-AND and wired-OR gates. With each increasing input to the wired-AND-OR network, one extra transistor is required by each of the wired gates. This causes the circuit to expand width-wise; hence, the latency of the wired logic remains constant for any value of $n$.

The size of the load transistor driving these wired-AND and -OR gates will also increase with increasing input, so we consider the maximum fan-in of one gate as equal to 4. For $k > 4$, an extra load transistor is connected in parallel. Generally, for k-inputs we require $r = \lceil k/4 \rceil$ load transistors. A total of $k + r$ transistors is required to design the k-input wired AND-OR network with a constant latency of 1 transistor.

## 5. Experiments and Results

In this section, we present the experiments we conducted on the proposed FS SEDC checker and the overall proposed SEDC-based HW-level fault tolerance system. The results of each experiment are given along with the experimental details in the subsections below.

*5.1. Fault Test on FS SEDC Checker.* The FS $SEDC_1$, $SEDC_2$, $SEDC_3$, and $SEDC_4$ circuits in our paper were tested for stuck-at-0, stuck-at-1, transistor-stuck-ON, and transistor-stuck-OFF faults. We assume a single-fault model where faults occur one at a time, and there is enough time between detection of the first fault and the occurrence of another fault [29]. In Table 2, we provide a summary of fault analysis of an $SEDC_1$ checker circuit. We applied one fault at a time in

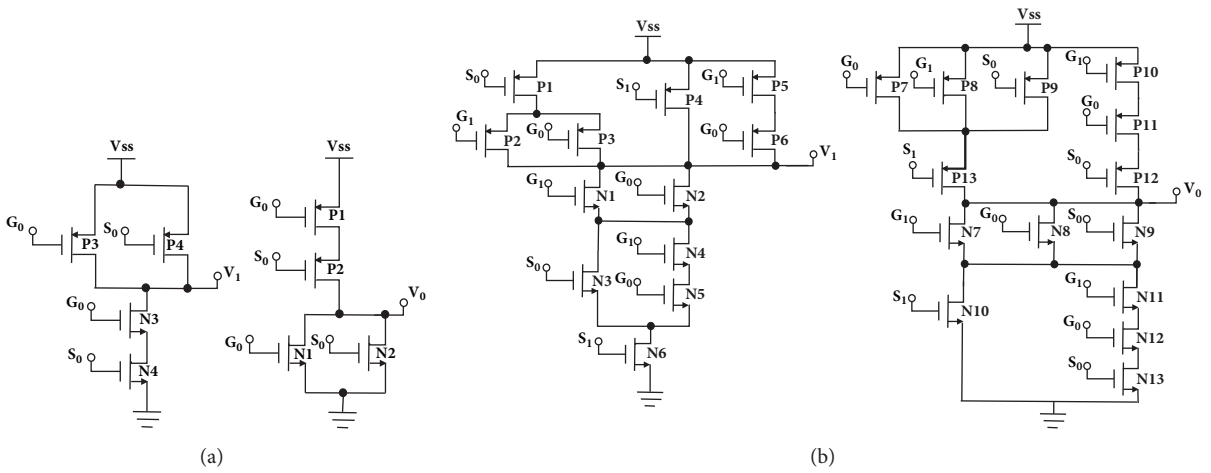



Figure 5: CMOS-based circuits of FS (a) $SEDC_1$ checker and (b) $SEDC_2$ checker.

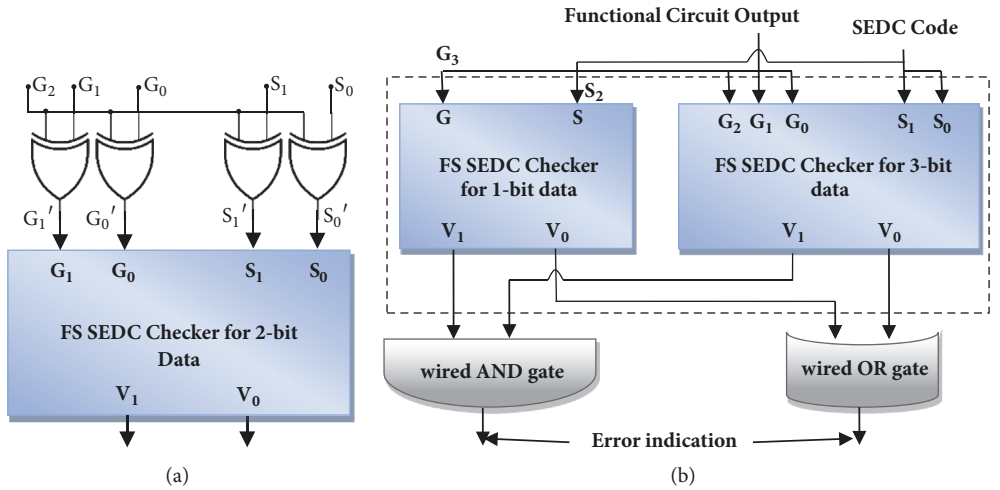



Figure 6: Block diagram of FS (a) $SEDC_3$ checker, and (b) $SEDC_4$ checker.

the circuit of Figure 5(a) and observed the output. In single-fault operation, the circuit either produced the correct output or never produced any invalid code words (exhibiting FS property), as shown in Table 2.

*Case 1* (transistor stuck ON). In Table 2, we show all six cases of transistor stuck ON faults (one at a time). For the cases with N3 or N4 stuck ON, the circuit shows fault detection by one input code combination (represented with * symbol), and hence, the circuit is self-testing, whereas other cases showed that the circuit is fault secure as well as code disjoint.

*Case 2* (transistor stuck OFF). In Table 2, all six cases for transistor stuck OFF faults are shown. In cases where N1 or N2 was stuck OFF, the circuit demonstrates the self-testing property (represented with * symbol) and for the rest of the cases, the circuit is fault secure.

*Case 3* (input stuck at 0). When input $G_0$ or $S_0$ is stuck at 0, the circuit demonstrates the self-testing property; otherwise, it remains fault secure.

*Case 4* (input stuck at 1). When input $G_0$ or $S_0$ is stuck at 1, the circuit shows the self-testing property; otherwise, it remains fault secure.

There is one case where the output becomes floating (i.e., P3 or P4 stuck OFF). In either case, if we consider the floating voltage as logic high, then the circuit is fault secure, and if we consider the floating voltage as logic low, then the circuit is self-testing. Hence, we can say that the circuit in Figure 5(a), which is a 1-bit SEDC checker, is FS. Similar analysis was carried out when testing 2-, 3-, and 4-bit SEDC checkers, and we found that all these checkers are FS.

*5.2. Area, Delay, and Power Comparison.* In this section, we compare the area and delay of TSC Berger, FS SEDC, and m-out-of-2m code checkers. We use the two possible TSC Berger checker implementations from Piestrak et al. [23] and Pierce Jr. and Lala [26], with the m-out-of-2m code checker from Lala [24] for comparison. For the sake of fairness, the area overhead was measured in terms of the number of equivalent

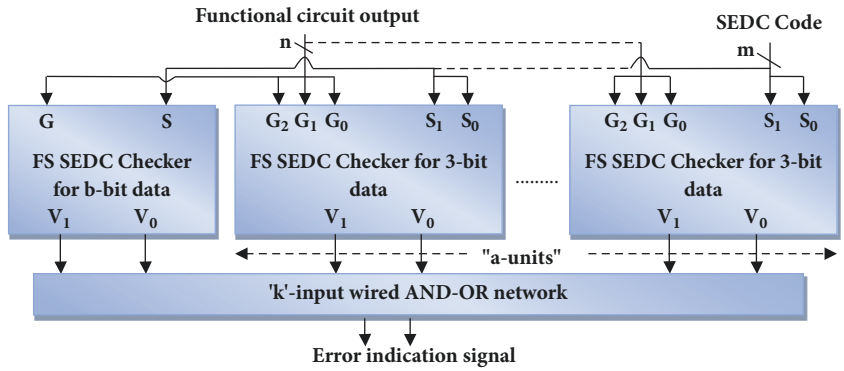



Figure 7: Block diagram of FS $SEDC_n$ checker.

Table 2: Results of single faults on FS $SEDC_1$ checker.

| $G_0$ | $S_0$ | $V_1$ | $V_0$ | $G_0$ | $S_0$ | $V_1$ | $V_0$ | $G_0$ | $S_0$ | $V_1$ | $V_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ***MOS P1or P2 is stuck ON*** | | | | ***MOS P1 or P2 is stuck OFF*** | | | | ***Input $C_0$ stuck at zero*** | | | |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | ⋆0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| ***MOS P3 or P4 is stuck ON*** | | | | ***MOS P3 or P4 is stuck OFF*** | | | | ***Input $F_0$ stuck at zero*** | | | |
| 0 | 1 | 1 | 0 | 0 | 1 | Floating | 0 | ⋆0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| ***Transistor N1 is stuck ON*** | | | | ***Transistor N1 is stuck OFF*** | | | | ***Input $C_0$ stuck at 1*** | | | |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | ⋆1 | 0 | 1 | 1 | ⋆1 | 1 | 0 | 0 |
| ***Transistor N2 is stuck ON*** | | | | ***Transistor N2 is stuck OFF*** | | | | ***Input $F_0$ stuck at 1*** | | | |
| 0 | 1 | 1 | 0 | ⋆0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | ⋆1 | 1 | 0 | 0 |
| ***Transistor N3 is stuck ON*** | | | | ***Transistor N3 is stuck OFF*** | | | | - | - | - | - |
| ⋆0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | - | - | - | - |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | - | - | - | - |
| ***Transistor N4 is stuck ON*** | | | | ***Transistor N4 is stuck OFF*** | | | | - | - | - | - |
| ⋆0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | - | - | - | - |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | - | - | - | - |

⋆ The cases where circuit shows self-testing property.

transistors. We made use of the assumptions by Smith [30] to translate gate-level circuits to transistor-level circuits.

Before comparison, we illustrate the functional dissimilarities of the three checkers with the help of Figure 8. Figure 8(a) shows the general block diagram of a TSC Berger code checker. For all the information symbols that the ISG of the functional circuit can produce in normal operation, the check symbol complement generator (CSCG) outputs ($S_B{}'$) correspond to the bit-by-bit complement of the expected check symbol $S_B$. The TSC two-rail checker validates that each bit of $S_B$ is the complement of corresponding bit of $S_B{}'$. As the size of the input data increases, the length of check symbol $S_B$ also increases, resulting in a longer length for the TSC two-rail checker tree, and hence the resulting delay.

A general block diagram of a TSC m-out-of-2m code checker is shown in Figure 8(b). The checker takes the information bits and check bits $S_W$ and partitions them into two parts. The numbers of 1's, i.e., the weight, of both parts are mapped to a pair of values which in binary belongs to a code, in most cases a two-rail code. The checker consists of a cellular structure of AND-OR gates as given by Lala [24].

Figure 8(c) depicts the general block diagram for an FS SEDC checker that resembles the structure of an m-out-of-2m code checker and differs from a Berger code checker. The FS SEDC checker block receives the information and check bits from the functional unit. If the input data length increases, the size of the FS checker block increases widthwise. The FS $SEDC_n$ block contains "$a + 1$" pairs of small SEDC checkers (subblocks). Each subblock of the FS SEDC checker produces "10" as the valid code output. The overall SEDC checker has a final 2-bit output $S_{10}$; unlike two-rail

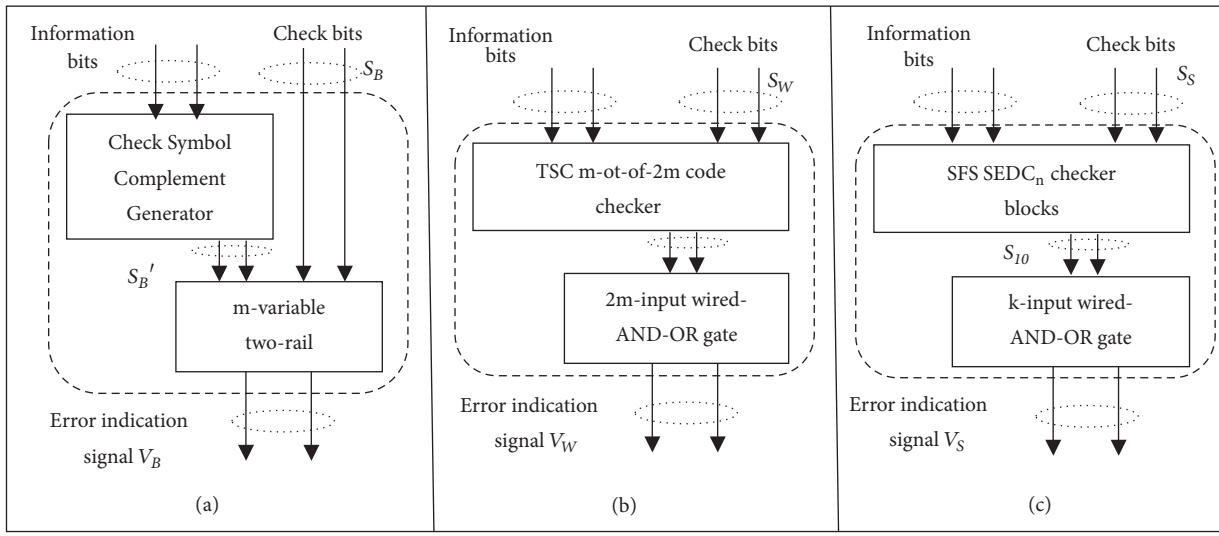



Figure 8: Block diagrams of (a) TSC Berger checker, (b) m-out-of-2m code checker, and (c) FS SEDC checker.

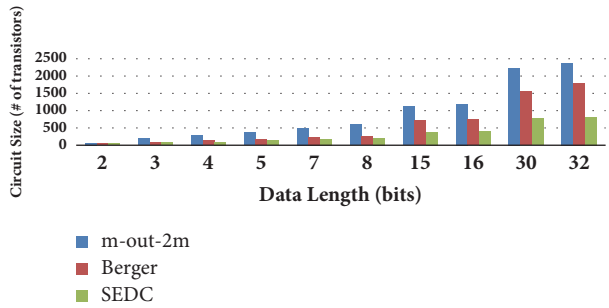



Figure 9: Area comparison of area-optimized Berger [23], SEDC, and m-out-of-2m [24] code checkers.

codes, only one of the output combinations "10" is considered a valid code word. A nonvalid checker output "00," "01", or "11" at output $S_{10}$ indicates the presence of a fault in the functional circuit or the FS checker itself. The k-input wired AND-OR network takes the "$a+1$" pairs of output from each SEDC checker subblock and then converts them into a final 2-bit error indication signal $V_S$.

*5.1. Fault Test on FS SEDC Checker.* Area-optimized realization of TSC Berger code checkers in Piestrak et al. [23] showed less area overhead than m-out-of-2m code checkers, which is apparent from Figure 9. But, if we consider the delay-optimized implementation of the TSC Berger code checker from Pierce Jr. and Lala [26], we see that the TSC Berger code checker requires more area than the FS SEDC and m-out-of-2m codes checkers [24], as shown in Table 3. For clarity, we discretely listed the area overhead offered based on code storage area and code checker area in Table 3. Also listed separately are the area overhead required by the TRC tree for the TSC Berger code checker, the wired-AND-OR network for FS SEDC, and the m-out-of-2m code checker.

For a fair comparison, the extra cost of the code storage area is also taken into account. We assumed that 1-bit storage is implemented by 12-MOS transistors [30]. Table 3 lists the area (in terms of the number of transistors) occupied by FS SEDC, delay-optimized Berger code, and m-out-of-2m code checkers for up to 32-bit data.

The FS SEDC$_n$ checker block shown in Figure 8(c) requires fewer gates, implemented with [26 + (a × 50)] MOS transistors if "b = 2," [50 + (a × 50)] MOS transistors if "b = 3," and [58 + (a × 50)] MOS transistors if "b = 4." The m-out-of-2m code checker implementation of Lala [24] requires $2m^2$ - 2m + 2 gates. The gate-level circuit is also translated to transistor-level circuits using data from Smith [30].

The results show that when scaling a 7-bit 0's counter to an 8-bit 0's counter, 154 extra MOS transistors are required. The m-out-of-2m code checker requires 60 MOS transistors when scaling a 7-out-of-14 checker to an 8-out-of-16 checker, whereas the SEDC checker requires only 18 extra MOS transistors. That is because a 7-bit SEDC checker is implemented with one SEDC$_3$ and one SEDC$_4$ circuit that contain 50 and 58 MOS transistors, respectively (a total of 108 transistors). An 8-bit SEDC checker is implemented using one SEDC$_2$ and two SEDC$_3$ checkers, requiring 26 and 100 (50x2) MOS transistors (a total of 126 transistors). This means that SEDC saves 88% of the number of transistors compared to a Berger code checker [26], and it saves 70% of the transistors when

Table 3: Area overhead of Berger [26], SEDC, and m-out-of-2m [24] code checkers.

| Data Bit | Berger Code | | | | SEDC | | | | m-out-of-2m | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Code storage Area* | *1's counter Area* | *TRC Area* | *Total Area* | *Code storage Area* | *Checker Area* | *AND-OR Network* | *Total Area* | *Code Storage Area* | *Checker Area* | *AND-OR Network* | *Total Area* |
| 2 | 24 | 22 | 4 | **50** | 24 | 26 | 0 | **50** | 24 | 36 | 0 | **50** |
| 3 | 24 | 80 | 8 | **112** | 24 | 50 | 0 | **74** | 36 | 152 | 0 | **188** |
| 4 | 36 | 180 | 12 | **228** | 36 | 58 | 6 | **100** | 48 | 240 | 10 | **298** |
| 5 | 36 | 178 | 16 | **230** | 48 | 76 | 6 | **130** | 60 | 300 | 14 | **374** |
| 7 | 36 | 396 | 24 | **456** | 60 | 108 | 8 | **176** | 84 | 420 | 18 | **522** |
| 8 | 48 | 550 | 28 | **626** | 72 | 126 | 8 | **206** | 96 | 480 | 20 | **596** |
| 15 | 48 | 1106 | 56 | **1210** | 120 | 250 | 14 | **384** | 180 | 900 | 38 | **1118** |
| 16 | 60 | 1308 | 60 | **1428** | 132 | 258 | 16 | **406** | 192 | 960 | 40 | **1192** |
| 30 | 60 | 2586 | 116 | **2762** | 240 | 500 | 26 | **766** | 360 | 1800 | 76 | **2236** |
| 32 | 72 | 3048 | 120 | **3240** | 264 | 526 | 28 | **818** | 384 | 1920 | 80 | **2384** |

compared to m-out-of-2m code checkers. Although Berger and m-out-of-2m checkers are TSC, while the proposed SEDC checker is only FS, all three checkers provide the same fault security.

*5.2.2. Delay.* As far as delay is concerned, the FS SEDC checker also performs better than Berger and cellular implementations for an m-out-of-2m code checker, as shown in Table 4. For the sake of uniformity, we designed all the basic gates using the same technology transistors (PMOS = $8\mu/2\mu$, NMOS = $4\mu/2\mu$) and evaluated the worst-case propagation delay of each circuit.

The SEDC checker shows almost a constant delay for n > 3 bits due to its parallel implementation, whereas the delay in the Berger code checker increases owing to an increase in gate levels (from 6 to 16) in the critical path, as shown by Pierce Jr. and Lala [26]. The delay for m-out-of-2m code checkers also continues to increase with increasing data lengths because the cellular implementation requires "m (= input data length)" gate levels in the critical path.

*5.2.3. Power Dissipation.* In order to evaluate the power dissipation of the three checkers, we used the PowerPlay power analyzer tool. We implemented the Berger [24], m-out-of-2m [26], and SEDC checker using Verilog and synthesized the circuits using Altera's Quartus II software. We targeted the circuit for a Cyclone II EP2C5AF256A7 chip, which has the least power dissipating properties among the Cyclone family. We allowed the synthesizer to create a balance between area and delay while synthesizing in order to get a better power estimate. We also enabled the synthesizer to use synthesizing model that takes intensive steps to optimize power for all three circuits. We clocked the inputs of the circuit with the default toggle rate and estimated the total thermal power dissipation for different values of input data width.

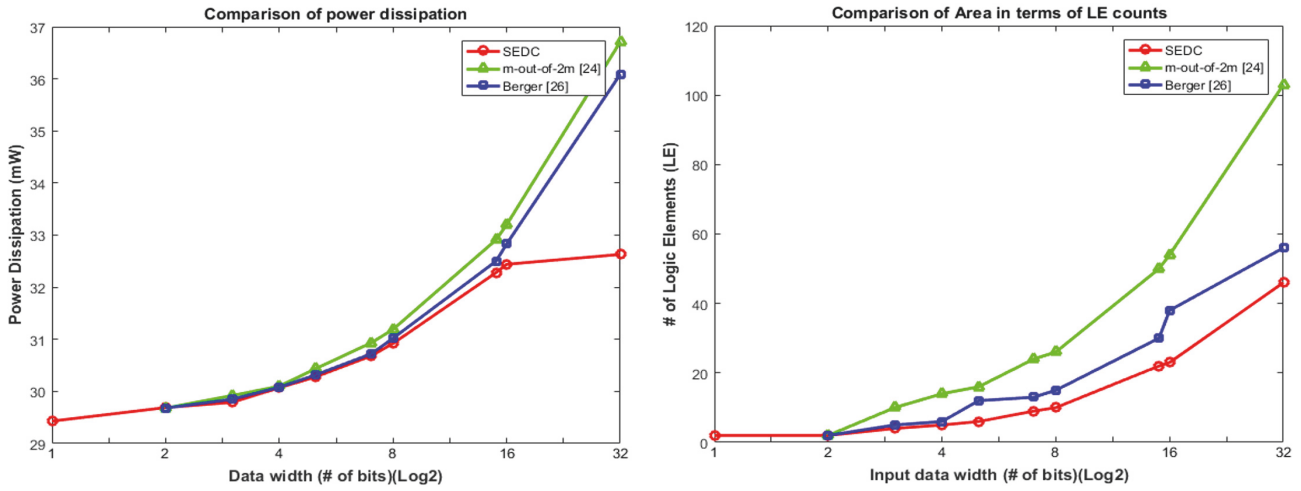
Figure 10(a) shows a comparison of power dissipation between the three checkers. The Berger and m-out-of-2m checkers exhibited a sudden increase in power dissipation when the input data width was changed from 16-bits to 32-bits, while SEDC showed a minimal change. This happens due to the increase in the number of two-rail checkers in the case of the Berger checker and due to the increase in the checker circuitry itself in the case of the m-out-of-2m checker, which is also evident in Figure 10(b), which depicts an area comparison between the three checkers in terms of # of logic elements (LE) occupied by the checkers.

*5.3. Fault Coverage of the Proposed HW-Level Fault Tolerance Scheme.* In order to elaborate the effectiveness of the SEDC CSG and its FS checker, we computed the fault coverage of the proposed SEDC-based HW-level fault tolerance scheme. We applied faults in the example circuit of Figure 4, given in Section 3.4. As most of the VLSI combinational circuits designed for mathematical operations, like add, subtract, multiply, division, etc., consist of multiple instances of 1-bit adders (full adders), hence the example circuit, i.e., a 4-bit adder, is a simple and good candidate for presenting the effectiveness of our scheme. We injected two major types of transient errors, i.e., stuck-at-0 and stuck-at-1 [29], at 24 nodes (at 6 nodes per full adder, as shown in Figure 11(b)). We injected these errors using 2-to-1 multiplexers, whose output is given by

$$mux_u = \begin{cases} in1\,(normal\ gate\ output) & if\ select\ (f\_enable) = 0 \\ in2\,(stuck-at-fault\ f \in \mathbf{F}) & if\ select\ (f\_enable) = 1 \end{cases} \quad (8)$$

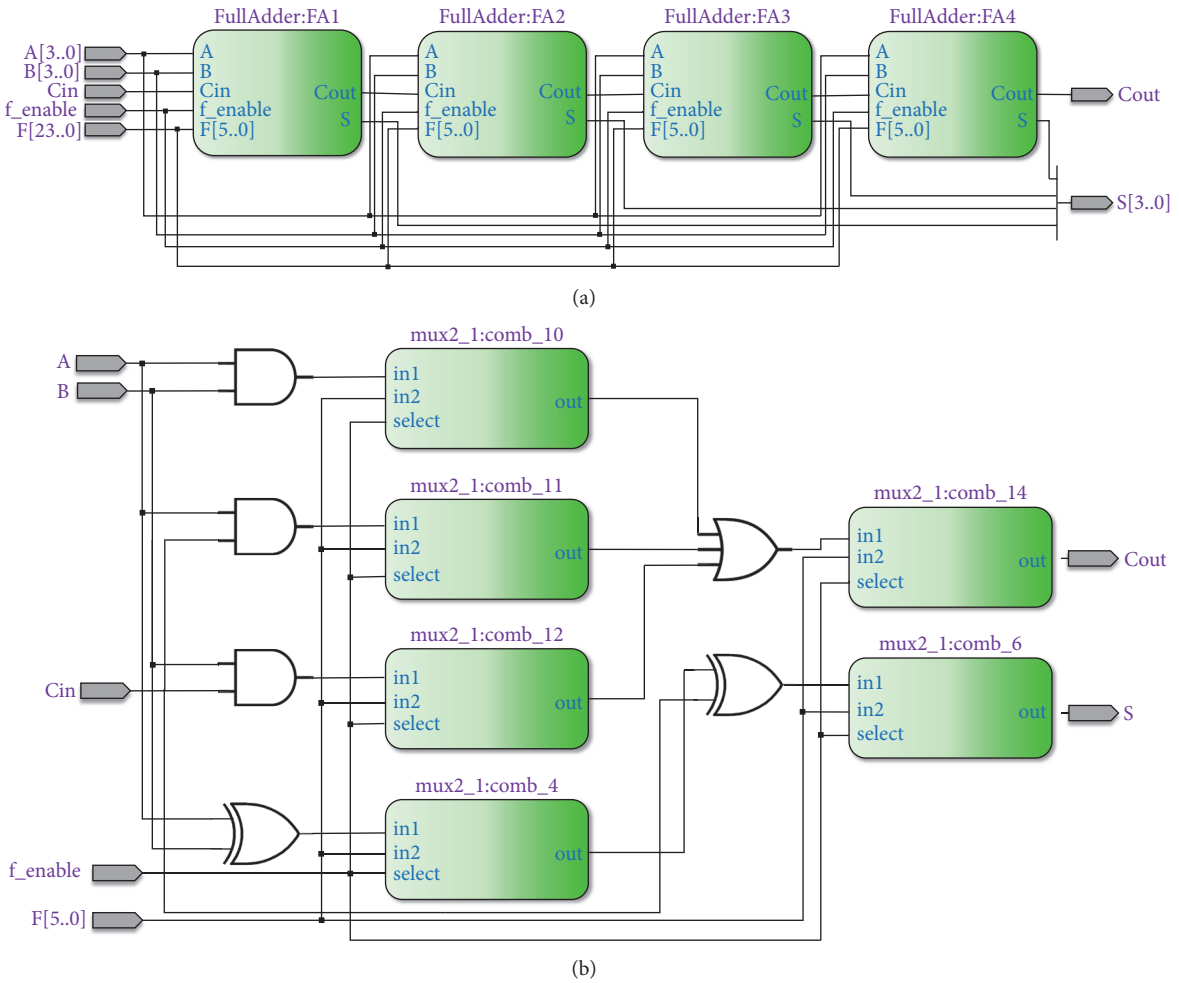
In Figure 11(a), the symbols A[3:0], B[3:0], Cin, f_enable, and F[23:0] denote the 4-bits input A, 4-bits input B, 1-bit carry-in, 1-bit fault enabling signal, and 24-bits fault signals, respectively, while Cout is the carry-out and S[3:0] represents the 4-bits sum output of the 4-bits adder. Figure 11(b) shows the detailed schematic of a single full adder.

We considered that the faults can occur at the outputs of the logic gates only and adopted a single-fault model according to which only one fault can occur at a time [29].




(a) (b)

FIGURE 10: Comparison of (a) power dissipation and (b) area in terms of LE counts, between Berger [26], m-out-of-2m [24], and SEDC checkers.




(a)

(b)

FIGURE 11: (a) RTL schematic of a 4-bit adder, and (b) 1-bit full adder, with fault injection.

Table 4: Critical path (CP) delay comparison of Berger, SEDC, and m-out-of-2m codes checker (unit = microseconds).

| Data Bits | Berger | SEDC | m-out-2m |
|---|---|---|---|
| 2 | 3.888 | 0.514 | 1.024 |
| 3 | 4.151 | 2.524 | - |
| 4 | 7.741 | 2.738 | 5.490 |
| 5 | - | 2.713 | 5.558 |
| 7 | 7.821 | 2.77 | 8.297 |
| 8 | 7.599 | 2.76 | 9.284 |
| 15 | 10.566 | 2.826 | - |
| 16 | 12.956 | 2.751 | |
| 32 | 17.964 | 2.771 | - |

Table 5: Summary of fault testing experiment on SEDC-based fault tolerant 4-bit adder.

| | (a) Total errors at the output of the adder | (b) BEs | (c) Detected BEs | (d) UEs | (e) Detected UEs | (f) Total detected errors | (g) Total undetected errors |
|---|---|---|---|---|---|---|---|
| **Total** | 1748 | 252 | 120 | 1496 | 1496 | 1616 | 132 |
| **Percentage** (%) | 100 | 14.42 w.r.t. (a) | 47.62 w.r.t. (b) | 85.58 w.r.t. (a) | 100 w.r.t (d) | 92.45 w.r.t. (a) | 7.55 w.r.t. (a) |

We used Altera's Quartus II software to design and synthesize the overall system and then simulated the system using ModelSim. We designed a self-checking test bench to evaluate the overall fault coverage. The statistics of the fault injection and its results are summarized in Table 5.

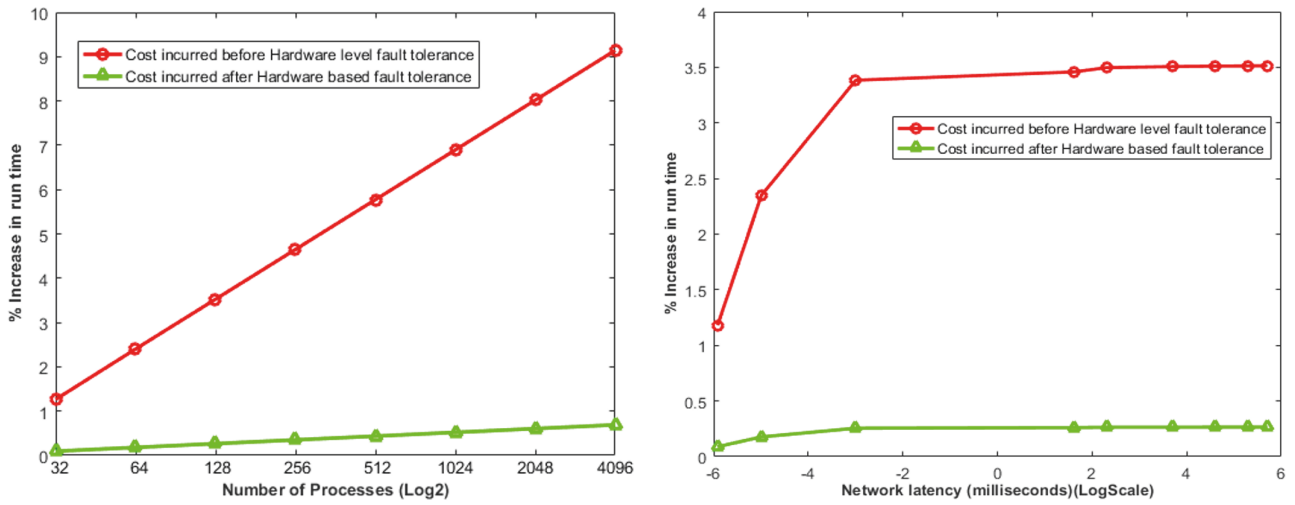
In total, we injected 6425 faults exhaustively, out of which 1748 faults actually caused a logical error at the output of the adder circuitry. Only 14.42% of these injected faults resulted in bidirectional errors (BEs), while most of the faults caused unidirectional errors (UEs). This also proved the fact that most of the errors in VLSI circuits result in UEs at the output [19–21]. Even though SEDC is an AUED scheme, and it provides 100% fault coverage against UEs, it also successfully detected 47.62% of the BEs, as shown in Table 5. This is due to the reason that SEDC partitions the input data word into multiple parts and encodes and decodes each part independently. Consequently, a subset of BEs is also partitioned into multiple UEs and thus detected by the proposed SEDC scheme.

*5.4. Cost Analysis: SW-Based Fault Tolerance Versus HW-Based Fault Tolerance.* In this section, we discuss the effect of fault propagation and the estimated cost of recovery from failure (also known as repair time) in big data computing in two cases: (a) when HW-based fault tolerance is applied, and (b) when only SW-based fault tolerance is applied. For simplicity in our analysis, we take the example of a coordinated checkpointing (CC) algorithm, which is widely used in HDFS for data recovery [31].

In HDFS, an image is used to define metadata (which contains node data and a list of blocks belonging to each file), while checkpoint defines the persistent record of the image, stored on a secondary NameNode (SNN) (also called DataNode) or Checkpoint Node, or in some cases on the primary NameNode (PNN) itself. If the PNN uses the CC data recovery algorithm, the checkpoints are distributed among multiple SNNs. During normal operation, the SNN sends heartbeats (a communication signal) to the PNN periodically. If the PNN does not receive a heartbeat from the SNN for certain fixed amount of time, the SNN is considered to be out of service, and the block replicas it hosts are considered to be unavailable. In this case, the PNN initiates the CC recovery algorithm, which includes signaling (sending heartbeats with control signals to other nodes) and replicating the copy of failed SNN data (available on the checkpoint nodes) to the other nodes in a coordinated way [31].

For our cost analysis, we would like to compute the cost associated with the CC data recovery algorithm for which we assume a cloud application, such as a message passing interface (MPI) program that comprises $p$ logical processes that communicate through message passing (heartbeats). Each process is executed on a virtual machine and sends a message to remaining $p-1$ processes with equal probabilities. We also consider that the message sending, checkpointing, and fault occurrence events are independent of each other. Assuming that a process is modelled as a sequence of deterministic events, i.e., every step taken by the process has a known outcome, and failure only occurs during message passing with equal probability and not during checkpointing or recovery, we use the analytical cost model given in [4] for cost analysis of fault tolerance at the SW level. According to [4], $T$ denotes the total execution time of a process without fault tolerance, while $T_{CP}$ and $T_{RO}$ represent the checkpointing and failure recovery overheads, respectively. Then, the total cost of fault tolerance per process is given by

$$C = \frac{T_{CP} + T_{RO}}{T} \times 100 \quad (9)$$




(a) (b)

Figure 12: Effect of (a) number of processes, and (b) network latency, on data recovery overhead in CC algorithm.

Assuming that the average time to roll back a failed process is $C_{rb}$and mean time between failures is $1/P(f)$, where $P(f)$denotes the probability of failure, then according to [4], the average recovery cost in CC per process is given by

$$\overline{T_{RO}} = \frac{C_{rb}}{(1/P(f))} = P(f)\,C_{rb} \tag{10}$$

Let $P(cp)$ denote the probability that a process starts checkpointing, then $(1 - P(cp))^p$ becomes the probability that $p$ processes do not start checkpointing, while $1 - (1 - P(cp))^p$becomes the probability that at least one process starts a checkpoint. Consequently, $1/(1 - (1 - P(cp))^p)$ represents the checkpointing interval. A process can be the initiator of checkpointing with probability $1/p$ and generate request (REQ) and acknowledgement signals (ACK) to the rest of the $p - 1$ noninitiators (total $2(p - 1)$ signals) and likewise be a noninitiator with probability $1 - 1/p$ and generate only one ACK signal in response to the initiator. As a result, there are $3(p - 1)/p$ average messages generated per checkpoint, and the average overhead per checkpoint is $C_w + (3(p - 1)/p)C_{nl}$, where $C_w$denotes the average time to write a checkpoint to a stable node and $C_{nl}$denotes the average network latency. Then, the average checkpointing cost for a process is given by

$$\begin{aligned}\overline{T_{CP}} &= \frac{C_w + (3(p-1)/p)\,C_{nl}}{1/\left(1 - (1 - P(cp))^p\right)} \\ &= \left(1 - (1 - P(cp))^p\right)\left(C_w + \frac{3(p-1)}{p}C_{nl}\right)\end{aligned} \tag{11}$$

Using the cost model given in (9), (10), and (11), we carried out the cost of data recovery in the CC algorithm with the parameters, $p = 128$ processes (virtual machines), $P(cp) = 1/15$ (one checkpointing per 15 minutes), $C_{nl} = 20$ *msecs*, $C_w = 1$ *sec*, $C_{rb} = 2$ *secs*, as given in [4]. We consider the value of $P(f) = 1/168$ which implies that 100% of the faults in hardware are propagated to the SW level in the absence of HW-level fault tolerance, while each fault occurs after 168 hours (one week's time). After we apply HW-level fault tolerance, the probability of failure $P(f)$ reduces to $P'(f) = 0.755 \times P(f)$, where the value 0.755 signifies that only 7.55% of the faults are unhandled by the proposed HW-level fault tolerance system (see Table 5). We vary one of the above parameters by keeping the other constant and observe the effect of data recovery cost with and without the proposed HW-level fault tolerance.

The graph in Figure 12(a) shows the average cost of data recovery when the number of processes $p$ is increased from 32 to 4096 (virtual machines). We consider that an application is partitioned into $p$ processes and each process runs on a virtual machine. The increase in number of processes causes a sharp increase in data recovery cost in the CC algorithm because every process has to coordinate with each other in case of a failure.

Figure 12(b) depicts the effect of network latency on the cost of data recovery. In this case we increased the network latency from 2 milliseconds to 300 milliseconds. Network latency depends heavily upon the traffic situation, network bandwidth, data size, and number of active nodes in the network. Figure 12(b) shows that increasing network latency has a negative impact on data recovery because it takes a longer time for processes to communicate with each other, resulting in delayed data recovery.

Figure 13 illustrates the situation where we increase the checkpointing frequency from one checkpoint per hour (1/60) to one checkpoint per minute. Even though the increase in checkpointing frequency improves the overall fault tolerance, it also increases the overall fault tolerance overhead, as shown in Figure 13.

Finally, we show the effect of the increasing probability of failure on the cost of data recovery in Figure 14. We varied the failure frequency from one failure per 1024 hours to one

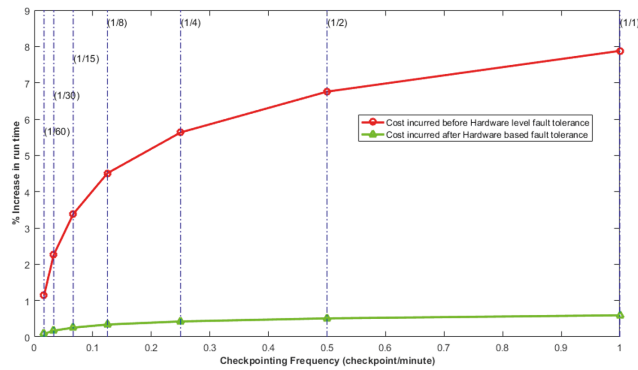



Figure 13: Effect of checkpointing frequency on data recovery cost in CC algorithm.

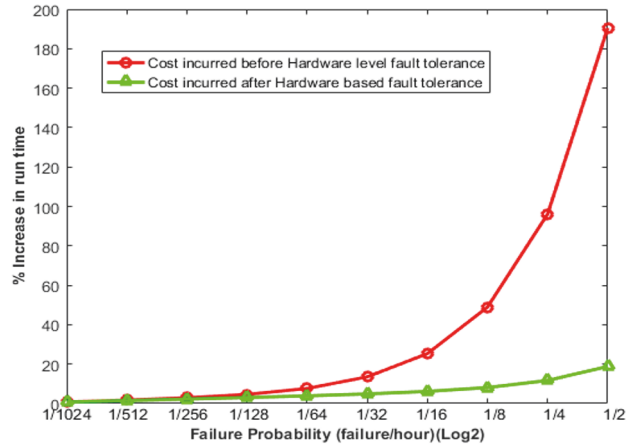



Figure 14: Effect of failure probability on data recovery in CC algorithm.

failure per 2 hours, which caused a huge impact on fault tolerance overhead, as shown in Figure 14. But, if we detect most of the errors at the hardware level, the average cost of data recovery reduces to a tolerable limit, as shown in Figure 14.

Because of the errors arising at the HW level, the average cost of data recovery in terms of percent increase in runtime in all of the above cases is much higher if we apply fault tolerance at the SW level only. Among the four parameters, i.e., # of processes, network latency, checkpointing frequency, and frequency of failure, frequency of failure has the worst effect on the average cost of data recovery. The proposed HW-level fault tolerance reduces the average cost to a tolerable limit, which is promising for big data and cloud computing applications. Although there is a one-time cost associated with HW-level fault tolerance, it provides high reliability against potential failures leading to severe socioeconomic consequences in big data and cloud computing.

## 6. Conclusions and Future Work

In this paper, we presented a concurrent error detection coding-based HW-level fault tolerance scheme for big data and cloud computing. The proposed method uses SEDC codes to protect against transient errors, which is a major problem in modern VLSI circuits. We also presented an FS SEDC checker that not only detects errors in the functional circuitry but also remains failsafe under s-a-1, s-a-0, s-open, and s-short errors within checker circuitry. We compared the performance of the proposed SEDC checker with Berger and m-out-of-2m checker in terms of area, delay, and power dissipation, which proves the superiority of the proposed SEDC checker. Using the example of a 4-bit adder circuit, we presented a complete SEDC-based HW-level fault tolerance system and computed its fault coverage by exhaustive fault injection. The SEDC-based HW-level fault tolerance method shows 100%, 47%, and 92.5% fault coverage against unidirectional, bidirectional, and total errors, respectively. In order to show the effectiveness of the proposed SEDC-based HW-level fault tolerance method in big data and cloud computing applications, we compared the average cost of fault tolerance overhead with and without HW-level fault tolerance. The results show that HW-level fault tolerance reduces the probability of failure due to transient errors, consequently reducing the average cost of fault tolerance overhead to a great extent when compared with SW level fault tolerance only.

From hardware-level evolution such as microprocessors, memories, and parallel computing devices, to system-level advancements such as networking, data security, resource

sharing protocols, and operating systems, the underlying technologies have changed a lot since the emergence of big data and cloud computing. Fault tolerance plays a vital role in big data and cloud computing because of the uncertain failures associated with the huge amount of data, both at SW and HW levels. Given this, we believe that this research opens new opportunities for fault tolerance at the hardware-level for big data and cloud computing.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This study was partly supported by research funds from Chosun University, 2017, Sogang University Research Grant of 2012 (201210056.01) and MISP (Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW (2015-0-00910) supervised by the IITP (Institute for Information & communications Technology Promotion).

## References

[1] M. Chen, S. Mao, and Y. Liu, "Big data: A survey," *Mobile Networks and Applications*, vol. 19, no. 2, pp. 171–209, 2014.

[2] R. Jhawar, V. Piuri, and M. Santambrogio, "A comprehensive conceptual system-level approach to fault tolerance in Cloud Computing," in *Proceedings of the 2012 6th Annual IEEE Systems Conference (SysCon)*, pp. 1–5, Vancouver, Canada, March 2012.

[3] A. Katal, M. Wazid, and R. H. Goudar, "Big data: issues, challenges, tools and good practices," in *Proceedings of the 6th International Conference on Contemporary Computing (IC3 '13)*, pp. 404–409, IEEE, Noida, India, August 2013.

[4] Y. M. Teo, B. L. Luong, Y. Song, and T. Nam, "Cost-performance of fault tolerance in cloud computing," *Special Issue of Journal of Science and Technology*, vol. 49, no. 4A, pp. 61–73, 2011.

[5] M. Nazari Cheraghlou, A. Khadem-Zadeh, and M. Haghparast, "A survey of fault tolerance architecture in cloud computing," *Journal of Network and Computer Applications*, vol. 61, pp. 81–92, 2016.

[6] J. Deng, S. C.-H. Huang, Y. S. Han, and J. H. Deng, "Fault-tolerant and reliable computation in cloud computing," in *Proceedings of the 2010 IEEE Globecom Workshops, GC'10*, pp. 1601–1605, Miami, Fla, USA, December 2010.

[7] J. Liu, S. Wang, A. Zhou, S. Kumar, F. Yang, and R. Buyya, "Using proactive fault-tolerance approach to enhance cloud service reliability," *IEEE Transactions on Cloud Computing*, p. 1, 2017, http://ieeexplore.ieee.org/document/7469864.

[8] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proceedings of the 2013 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN '13*, pp. 109–114, Hong Kong, China, August 2013.

[9] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD Conference on Management of Data, SIGMOD '13*, pp. 725–736, New York, NY, USA, June 2013.

[10] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computer*, p. 10, Berkeley, Calif, USA, 2012.

[11] P. Wang, D. J. Dean, and X. Gu, "Understanding Real World Data Corruptions in Cloud Systems," in *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, pp. 116–125, Tempe, Ariz, USA, March 2015.

[12] P. A. Parker, "Discussion of Reliability Meets Big Data: Opportunities and Challenges," *Quality Engineering*, vol. 26, no. 1, pp. 117–120, 2014.

[13] H. Bauer, P. Ranade, and S. Tandon, "Big data and the opportunities it creates for semiconductor players," in *McKinesy on Semiconductors, BIG DATA for Semiconductors*, McKinesy & Company, 2012.

[14] H. Ueno and K. Namba, "Construction of a soft error (SEU) hardened Latch with high critical charge," in *Proceedings of the 29th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, DFT '16*, pp. 27–30, September 2016.

[15] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," *The Computer Journal*, vol. 38, no. 2, pp. 43–52, 2005.

[16] T. Karnik, P. Hazucha, and J. Patel, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[17] L.-T. Wang, X. Wen, and K. S. Abdel-Hafez, "Design for testability," *VLSI Test Principles and Architectures*, pp. 37–103, 2006.

[18] N. Alves, "State-of-the-art techniques for detecting transient errors in electrical circuits," *IEEE Potentials*, vol. 30, no. 3, pp. 30–35, 2011.

[19] S. Kotaki and M. Kitakami, "Codes correcting asymmetric/unidirectional errors along with bidirectional errors of small magnitude," in *Proceedings of the 20th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC '14*, pp. 159-160, Singapore, November 2014.

[20] B. S. Manjunatha, G. S. D. Pateel, and V. Shah, "Oral fibrolipoma. A rare histological entity: report of 3 cases and review of literature," *Journal of Dentistry*, vol. 7, no. 4, pp. 226–231, 2010.

[21] N. K. Jha and M. B. Vora, "A t-unidirectional error-detecting systematic code," *Computers & Mathematics with Applications*, vol. 16, no. 9, pp. 705–714, 1988.

[22] J. Kim, D.-H. Lee, and W. Sung, "Performance of rate 0.96 (68254, 65536) EG-LDPC code for NAND Flash memory error correction," in *Proceedings of the 2012 IEEE International Conference on Communications, ICC '12*, pp. 7029–7033, June 2012.

[23] S. Piestrak, D. Bakalis, and X. Kavousianos, "On the design of self-testing checkers for modified Berger codes," in *Proceedings of the Seventh International On-Line Testing Workshop*, pp. 153–157, Taormina, Italy, 2001.

[24] P. K. Lala, *Self-Checking and Fault Tolerant Digital Design*, Academic press, UK, 2001.

[25] J.-A. Lee, Z. A. Siddiqui, N. Somasundaram, and J.-G. Lee, "Self-checking look-up tables using scalable error detection coding (SEDC) scheme," *Journal of Semiconductor Technology and Science*, vol. 13, no. 5, pp. 415–422, 2013.

[26] D. A. Pierce Jr. and P. K. Lala, "Modular implementation of efficient self-checking checkers for the Berger code," *Journal of Electronic Testing*, vol. 9, no. 3, pp. 279–294, 1996.

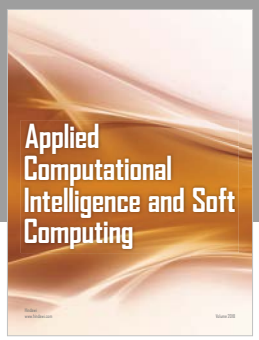
[27] Z. A. Siddiqui, P. Hui-Jong, and J. Lee, "Area-Time Efficient Self-Checking ALU Based on Scalable Error Detection Coding," in *Proceedings of the 2013 Euromicro Conference on Digital System Design (DSD)*, pp. 870–877, Los Alamitos, CA, USA, September 2013.

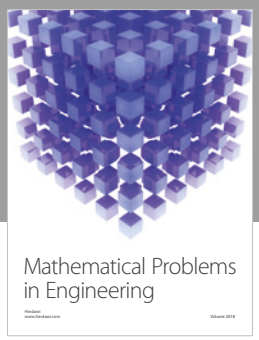
[28] Z. A. Siddiqui and J.-A. Lee, "Online error detection in SRAM based FPGAs using Scalable Error Detection Coding," in *Proceedings of the 5th Asia Symposium on Quality Electronic Design, ASQED '13*, pp. 321–324, Penang, Malaysia, August 2013.

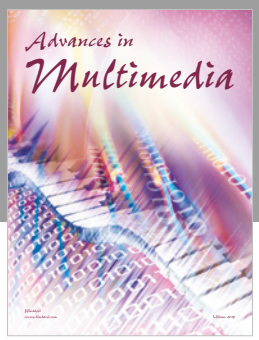
[29] D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers*, vol. C-22, no. 3, pp. 263–269, 1973.

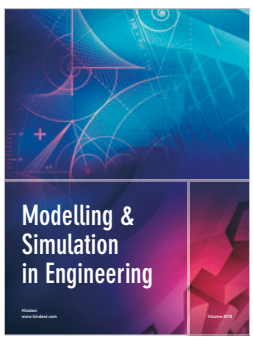
[30] M. A. Smith, Transistor counts, http://en.wikipedia.org/wiki/Transistor_count, April 05, 2018.

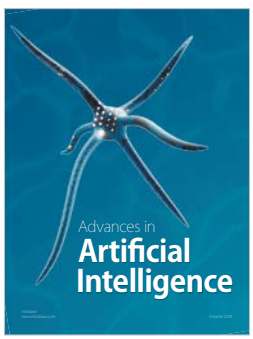
[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*, 10, 1 pages, Piscataway, NJ, USA, May 2010.

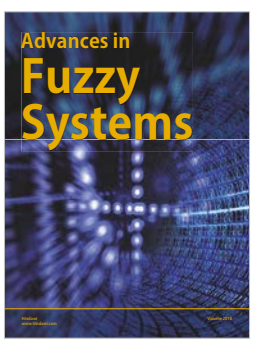


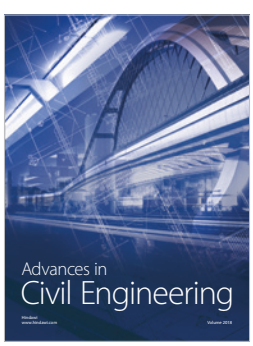


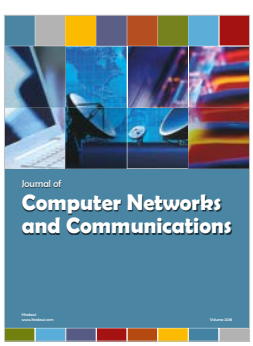


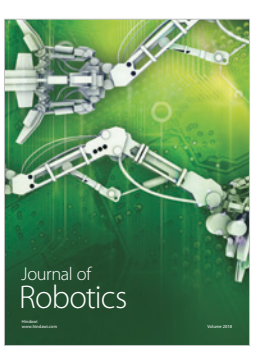


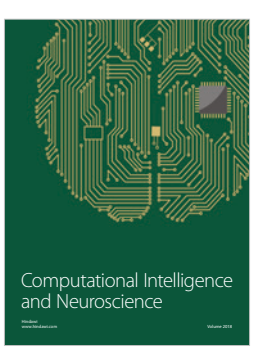












Hindawi

Submit your manuscripts at
www.hindawi.com