ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions

Nahmsuk Oh, *Member*, *IEEE*, Subhasish Mitra, *Member*, *IEEE*, and Edward J. McCluskey, *Fellow*, *IEEE*

Abstract—Errors in computing systems can cause abnormal behavior and degrade data integrity and system availability. Errors should be avoided especially in embedded systems for critical applications. However, as the trend in VLSI technologies has been toward smaller feature sizes, lower supply voltages, and higher frequencies, there is a growing concern about temporary errors as well as permanent errors in embedded systems; thus, it is very essential to detect those errors. *Software Implemented Hardware Fault Tolerance* (SIHFT) is a low-cost alternative to hardware fault tolerance techniques for embedded processors: It does not require any hardware modification of *Commercial Off-The-Shelf* (COTS) processors. ED^{4}I is a SIHFT technique that detects both permanent and temporary errors by executing two "different" programs (with the same functionality) and comparing their outputs. ED^{4}I maps each number, x, in the original program into a new number x', and then transforms the program so that it operates on the new numbers so that the results can be mapped backwards for comparison with the results of the original program. The mapping in the transformation of ED^{4}I is $x' = k \cdot x$ for integer numbers, where k determines the fault detection probability and data integrity of the system. For floating point numbers, we find a value of k_f for the fraction and k_e for the exponent separately and use $k = k_f \times 2^{k_e}$ for the value of k. We have demonstrated how to choose an optimal value of k for the transformation. This paper shows that, for integer programs, the transformation with $k = -2$ was the most desirable choice in six out of seven benchmark programs we simulated. It maximizes fault detection probability under the condition that data integrity is highest.

Index Terms—Software implemented hardware fault tolerance (SIHFT), low cost fault tolerance, concurrent error detection, data diversity, duplicated instructions.

1 Introduction

Embedded computing systems are widely used in remote, critical, and high-availability applications [1], [2], but errors in embedded systems can cause abnormal behavior and degrade system reliability, data integrity, and availability. Fault avoidance techniques such as radiation hardening and shielding or hardware redundancy such as duplicated or triplicated modules have been the traditional approaches to meet reliability requirements. However, as *Commercial Off-The-Shelf* (COTS) component microprocessors are widely used in *System-On-Chip* (SoC) designs in embedded systems, fault avoidance and hardware redundancy techniques are expensive. Radiation hardened processors are out-dated and slower compared to the state-of-art COTS processors. In the Stanford ARGOS project [3], it was shown that the throughput of the COTS board was an order of magnitude higher than that of the radiation-hardened board. On the other hand, hardware redundancy using COTS processors may require hardware modifications or design changes resulting in increased cost and design time. *Software-Implemented Hardware Fault Tolerance* (SIHFT) is a low cost alternative to hardware fault tolerance techniques because it does not require any hardware modification and can be easily adopted to most hardware platforms. For example, in [4], software redundancy techniques are described to achieve low cost fault tolerance in a robot controller. In [3], it is shown that, in radiation environments, the reliability of COTS components can be enhanced by using SIHFT techniques for detecting, correcting, and recovering from errors without changing the hardware.

Error Detection by Data Diversity and Duplicated Instructions (ED^{4}I) is a SIHFT technique that detects both temporary and permanent faults by executing two "different" programs with the same functionality but with different data sets and comparing their outputs. Temporary faults that corrupt either of the two programs can be detected by comparing the results. These temporary faults include transient faults in the processor and bit-flips in memory. A *bit-flip* is an undesired change in the state of a memory cell, and single event upsets (SEUs) are one of the major sources of bit-flips [3]. For example, bit-flips in the code portion of the program during the program execution may change the behavior of the program and generate incorrect results; then, comparing the incorrect results (produced by the corrupted program) with the correct results (produced by uncorrupted program) will detect the bit-flip. If the faults corrupt the original and duplicate programs, we can still detect the faults by comparing the results as long as the two programs produce two different incorrect results. Our technique cannot detect the fault that causes one of the programs to fall into an infinite loop and never stop. In this case, software control flow checking [5] or a watchdog timer [6] can be added to our technique to detect control flow errors.

• *The authors are with the Center for Reliable Computing, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.*
E-mail: {nsoh, smitra, ejm}@crc.stanford.edu.

Manuscript received 15 Oct. 2000; revised 16 May 2001; accepted 19 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114410.

Several SIHFT techniques using time redundancy have been proposed. Those include assertions [7], a watchdog task [8], block entry exit checking and error capturing instructions [9], but they can detect only temporary faults, not permanent faults. Software duplication techniques such as those in [10] can also detect temporary faults, but they cannot detect hardware permanent faults such as stuck-at faults because hardware faults corrupt the original and duplicate copies of a program in the same way and produce the same incorrect results. However, our technique can detect permanent faults in data paths in functional units by executing two programs with diverse data using different parts of the functional units and comparing the results. In this paper, we focus on detecting permanent faults, e.g., stuck-at faults in data paths.

Executing two different programs or duplicated instructions requires more memory and execution time resulting in memory and performance overhead. For example, in [11], it is shown that executing duplicated instructions causes 89.3 percent execution time overhead, on average, when eight benchmark programs are executed on a R4400 MIPS processor. However, if we have enough memory and use a faster processor that can accommodate the software redundancy in embedded systems, our technique is a low-cost substitute for costly hardware fault tolerance techniques, which often require the modification of the entire system design. Our technique can even be applied to existing embedded systems in applications by recompiling the program with ED^{4}I.

The "different" programs with diverse data are generated in the following way: Given a program, we automatically transform it into a new program in which all variables and constants are multiplied by a *diversity factor k*. Depending on the factor *k*, the original and the transformed program may use different parts of the underlying hardware and propagate fault effects in different ways (for instance, the fault is provoked in one of the programs, but not in the other); therefore, if the two different programs produce different outputs due to a fault, we can detect the fault by examining whether the results of the transformed program are also *k* times greater than the results of the original program. There are two ways to check the results. First, another concurrently running program can compare the results. Second, the main program that spawns the original program and the transformed program checks their results after they are completed.

The value of the factor *k* determines the hardware fault (such as stuck-at faults) detection capability of ED^{4}I. It should satisfy two goals: The primary goal is to guarantee *data integrity*; that is, the probability that the two programs do not produce identical erroneous outputs. The secondary goal is to maximize the probability that the two programs produce different outputs for the same hardware fault so that error detection is possible (error detection probability). However, the factor *k* should not cause an overflow in the functional units. In order to determine the optimum value of *k*, we have developed an analysis technique based on the probabilistic modeling of logic networks [12] and a design *diversity metric* [13]. The diversity metric was used in [13] to quantify diversity among several designs. We use this metric to measure the diversity between the original and transformed programs.

We have implemented a preprocessor that automatically transforms a program to a new program based on the algorithm described in Appendix 1.

Unlike previous data diversity techniques [14], [15], [16] that target software faults, this paper presents a new approach to data diversity for online hardware fault detection. Our contributions are 1) devising an algorithm that transforms a program to a new program with diverse data, 2) quantifying diversity between programs using a metric that was previously developed for hardware diversity, and 3) demonstrating how to choose an optimal value of *k* for the transformation. We discuss previous work in Section 2 and present the program transformation algorithm in Section 3. We present how to determine an optimal value of *k* in Section 4 and show benchmark simulation results in Section 5. We discuss how to handle overflow in Section 6 and floating point numbers in Section 7. Finally, conclusions are presented in Section 8.

2 Previous Work

Design diversity was proposed in the fault tolerance literature as a technique for increasing the reliability of a system. Design diversity is defined as the independent generation of two or more different software or hardware elements to satisfy a given requirement [17]. The main objective of design diversity is to protect a redundant system from *common-mode failures*, which are failures that affect more than one module at the same time [18]. Design diversity also has been applied to software systems [19]. *N version programming* (NVP) [20] is one example of diversity in software. Design diversity in N version programming targets software design faults. In N version programming, different designers develop independent versions of a program to avoid common design errors. The *Consensus recovery block* technique [21] is a hybrid system combining NVP and recovery blocks: It is another example in which diversity is used in software. A variant of NVP, *N self-checking programming*, also employs the design diversity concept and is used in the Airbus A310 system [16], [22].

In [23], [24], data diversity was proposed for detecting software faults. In [23], input data in unused input space (input data that are not used during normal execution of the program) are used to detect faults in software and, in [24], diverse data from replicated sensors are used to tolerate faults in software.

However, our technique, ED^{4}I, is different from the previous software diversity or data diversity techniques. First, our target is not software design faults but hardware faults—both permanent and transient faults in the system. Second, while the previous techniques preserve the original program structure and apply diverse data in different or unused input domains of the original program, we transform a program to a new program in which the data are automatically diversified using a transformation algorithm presented in Section 3.

In the transformed program, the values of all variables and constants are multiplied by the diversity factor *k* in our

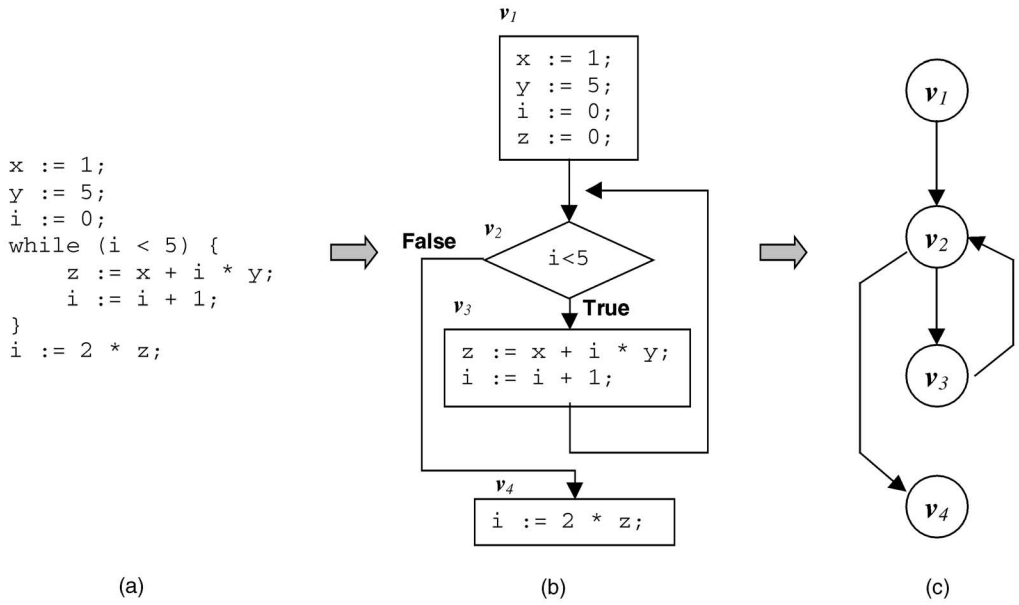


Fig. 1. (a) An example program. (b) A flow graph. (c) A program graph PG.

technique. This is similar to the *AN code* [25] in which each data word is multiplied by some constant A, but the error detection method in our technique is different from that in the *AN* code. The *AN* code detects an error by checking if the computation result is divisible by A, which requires expensive division computation if A is not an even number. By contrast, our technique detects an error by comparing the results of the original and the transformed program (not by checking whether the result is divisible by A or not). In addition, *AN* code is only applicable to fixed point numbers. Our technique is also different from Recomputing with Shifted Operands (RESO) presented in [26]. In RESO, the underlying hardware is modified so that each operation is recomputed with shifted operands. However, ED^{4}I is a SIHFT technique that does not change the original hardware. Instead, a "different" program is created by the transformation and comparing outputs of the original and the new program detects errors.

In [27], [28], the *vital coded processor* approach was presented. In this technique, three types of errors—operation, operator, and operand errors—can be detected by redundant code with static signatures. However, this technique also requires special hardware for encoding and decoding of the redundant code. Compared to this technique, our technique does not require any hardware modification and can be applied to programs easily during compilation.

Engel [29] suggests modifying the program so that all variables are negated. This is the same as using -1 for the value of k. However, our results show that the value of $k = -1$ is not the optimum value because for some functional units, data integrity is not guaranteed for $k = -1$. This will be shown in Sections 4 and 5. By contrast, our technique quantifies diversity and chooses the optimum diversity factor k (not limited to only -1) to maximize data integrity as well as fault detection capability. We describe the algorithm formally in Section 3 and prove its correctness in Appendix 1.

3 Program Transformation

This section presents a transformation algorithm that transforms a program (integer or floating point numbers) to a new program with diverse data. We will begin with definitions of terminologies and then describe the transformation algorithm. Finally, we will show an example illustrating how the transformation is implemented.

A *basic block* is *a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without branching except at the end*. By defining $V = \{v_1, v_2, \ldots, v_n\}$ as *the set of vertices representing basic blocks*, and $E = \{(i,j)|(i,j)$ is a branch from v_i to $v_j\}$ as *the set of edges denoting possible flow of control between the basic blocks*, a program can be represented by a *program graph*, $P_G = \{V, E\}$. For example, for the program in Fig. 1a (the corresponding flow graph is shown in Fig. 1b), there are four basic blocks: v_1, v_2, v_3, and v_4 as shown in Fig. 1c. Therefore, $V = \{v_1, v_2, v_3, v_4\}$, $E = \{(1,2),(2,3),(3,2),(2,4)\}$.

If x is k times greater than y, x is *k-multiple* of y. *Program transformation* transforms a program P to a new program P' with diverse data in which all variables and constants are k-multiples of the original values when the program P' is executed. It consists of two transformations: *expression transformation* and *branching condition transformation*. The expression transformation changes the expressions in P to new expressions in P' so that the value of every variable or constant in the expression of P' is always the k-multiple of the corresponding value in P. Since the values in P' are different from the original values, when we compare two values in a conditional statement, the inequality relationship may need to be changed. For example, the conditional statement `if (i < 5)` in P needs to be changed to `if (i > -10)` in P' when k is -2. Otherwise, the control flow determined by the conditional statements in P' would be

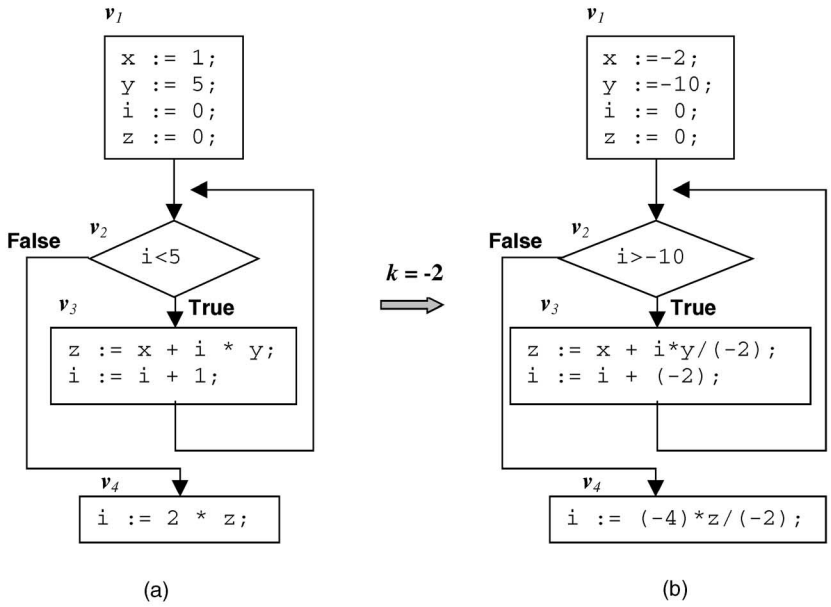


Fig. 2. (a) The original program. (b) The transformed program with $k = -2$.

different from the control flow in P, and the computation result from the diverse program P' would not be the k-multiple of the result from the original program. The branching condition transformation adjusts the inequality relationship in the conditional statement in P' so that the control flows in P and P' are identical.

A *k-factor diverse program* is *a program with a new program graph* $P'_G = \{V', E'\}$ *that is isomorphic to* P_G, *but all the variables and constants in* P' *are k-multiples of the ones in P.* Let us define S and S' as the sets of variables in P and P', respectively, and define n as the number of vertices (basic blocks) executed; then, $S(n)$ and $S'(n)$ are defined as:

- $S(n)$: *the set of values of the variables in S after n basic blocks are executed,*
- $S'(n)$: *the set of values of the variables in S′ after n basic blocks are executed.*

For example, in Fig. 1, the set of the variables in the program is $S = \{\mathtt{i}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. After the program is started and one basic block is executed, $n = 1$ and $S(1) = \{\mathtt{i} = 0, \mathtt{x} = 1, \mathtt{y} = 5, \mathtt{z} = 0\}$ because the four statements in the first basic block are executed. After v_2 and v_3 are executed, $n = 3$ and $S(3) = \{\mathtt{i} = 1, \mathtt{x} = 1, \mathtt{y} = 5, \mathtt{z} = 1\}$. The program transformation should satisfy:

1. P_G and P'_G are isomorphic.
2. $k \cdot S(n) = S'(n)$, for $\forall n > 0$. (where $k \cdot S(n)$ is obtained by multiplying all elements in $S(n)$ by k).

The condition in 1 tells us that the control flow in the two programs should be identical. The condition in 2 requires that all the variables in the transformed program are always k-multiples of those in the original program.

In the expression transformation, we build a parse tree for every expression in P and produce a new expression by recursively transforming the parse tree. In the branching condition transformation, we examine the inequalities in the conditional statements and modify them according to the value of k. Then, the transformed program always satisfies the conditions stated in 1 and 2. The formal description for the program transformation algorithm and the proof of the correctness are presented in Appendix 1.

The sample program in Fig. 1 is transformed to a diverse program shown in Figs. 2 and 3 where $k = -2$. We have assumed that the factor k is determined to cause no overflow when the transformed program is executed. However, we will also consider this overflow problem and present possible solutions in Section 6. Furthermore, in Section 7, we discuss applying ED^{4}I to floating point number arithmetic. The transformation is the same, but

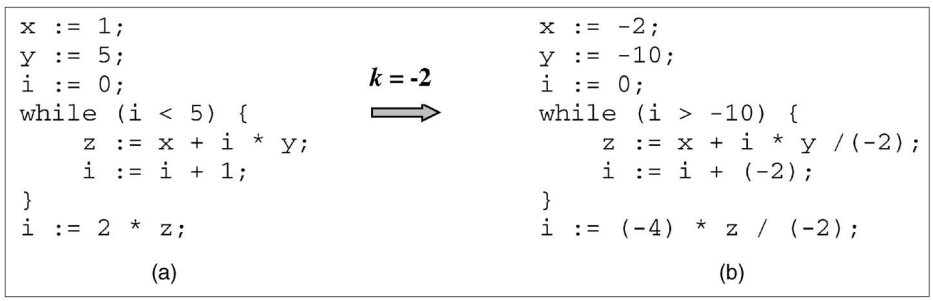


Fig. 3. (a) The original program. (b) The transformed program with $k = -2$.

floating point numbers use different optimal values of k. The factor k can be any integer if the program contains only integer arithmetic operations, but k should be 2^l (where l is an integer) if the program includes logical operations such as bit-wise AND, OR, or XOR.

If a program has mixed data types, such as integer as well as floating point numbers, we need multiple transformations with different values of k for each data type. The drawback of the multiple transformations is that it will increase performance and memory overhead.

4 Determination of k

The factor k determines how diverse the transformed program is. This section considers how to choose an optimal value of k that maximizes the diversity of the transformed program. For this purpose, we have developed two metrics to measure the diversity of the transformed program: data integrity and fault detection probability. An optimal value of k is the value that satisfies two goals: the primary goal is to maximize the data integrity and the secondary goal is to maximize the fault detection probability. Data integrity is more important because it guarantees no undetected error. For any given program, we first analyze the data integrity and fault detection probability of each functional unit of the system for various values of k. Next, as described in Section 4.6, we use these values to create an optimal value of k for the transformed program by looking at execution traces of the programs.

Section 4.1 presents a diversity metric we have adopted from [13], in which a diversity metric for systems with redundant hardware has been developed. From Section 4.2 to Section 4.5, we analyze data integrity and fault detection probability in functional units: bus, adders, multipliers and dividers, and a shifter. In our analysis, we consider integers from -5 to 5 for the value of k. Integers whose absolute values are greater than 5 are not considered for k in this paper because they have higher probability of overflow than the values considered. The analyzed data integrity and fault detection probability for functional units will be used in the next section to determine the optimum value of k for benchmark programs using execution traces of the programs.

4.1 Diversity Metric: Fault Detection Probability and Data Integrity

Researchers have studied techniques to quantify diversity in multiple designs and which technique should be used to measure the diversity [30], [31]. They use random variables Π and X to represent arbitrary programs and arbitrary inputs and the probability that Π fails on X is calculated. Our diversity metric is somewhat different because we need to compute the probability that a program fails due to a hardware fault in the system.

In our approach, fault detection probability and data integrity quantify diversity between two programs running on the same hardware. Suppose there are N_h number of functional units in the system, and denote the jth functional unit as h_j, $0 \le j < N_h$. Then, let us define:

- X *the set of inputs to* h_j *when a program P executes*
- X' *the set of inputs to* h_j *when a transformed program* P' *executes*
- x *a particular input to* h_j *produced by P; thus,* $x \in X$
- x' *a particular input to* h_j *produced by* P'*; thus,* $x' \in X'$
- $|X|$ *the number of elements in the set X*

Then, the output y of h_j with input x and the output y' of h_j with input x' should satisfy the relationship $y' = k \cdot y$ unless an overflow or a fault occurs in h_j. In the presence of a fault f_{ij} in h_j, let us define:

- E_i *the subset of X that contains inputs producing incorrect outputs in* h_j
- E_i' *the subset of* E_i *that contains inputs producing incorrect outputs that erroneously satisfy the relationship* $y' = k \cdot y$ *in the presence of a fault* f_{ij}

Then, $|E_i - E_i'|$ is the number of incorrect outputs that have the relationship $y' \neq k \cdot y$ and detect the fault by mismatch.

Assuming inputs are equally likely, we define the *fault detection probability in* h_j, $C_j(k)$, as:

$$C_j(k) = \sum_i \Pr\{f_{ij}\}\Pr_i\{y' \neq ky\} = \sum_i \Pr\{f_{ij}\}\left|\frac{|E_i - E_i'|}{|X|}\right|. \tag{3}$$

Moreover, we define the *data integrity probability in* h_j, $D_j(k)$, as:

$$D_j(k) = \sum_i \Pr\{f_{ij}\}\left|1 - \frac{|E_i - E_i'|}{|X|}\right|, \tag{4}$$

where $D_j(k)$ represents the probability of guaranteed data integrity.

If we assume a uniformuniform distribution for all faults,

$$\begin{aligned} C_j(k) &= \sum_i^{N_f} \Pr\{f_{ij}\}\left|\frac{|E_i - E_i'|}{|X|}\right| = \sum_i^{N_f} \frac{1}{N_f}\left|\frac{|E_i - E_i'|}{|X|}\right|, \\ D_j(k) &= \sum_i^{N_f} \Pr\{f_{ij}\}\left|1 - \frac{|E_i'|}{|X|}\right| = \sum_i^{N_f} \frac{1}{N_f}\left|1 - \frac{|E_i'|}{|X|}\right|, \end{aligned} \tag{5}$$

where N_f denotes the total number of faults in h_j.

For various values of k, $C_j(k)$, and $D_j(k)$ of h_j can be obtained either analytically or by simulation. By defining the *execution frequency* u_j of h_j as

$$\frac{\text{number of cycles in which } h_j \text{ is used}}{\text{total number of executions cycles}}$$

in the program execution trace, $C_j(k)$ and $D_j(k)$ are weighted by u_j and summed to obtain the overall $C(k)$ and $D(k)$ for a particular program.

$$\begin{aligned} C(k) &= \sum_j u_j C_j(k), \\ D(k) &= \sum_j u_j D_j(k). \end{aligned} \tag{6}$$

The *optimal value of k* is the value that maximizes $C(k)$ under the condition that $D(k)$ is highest.

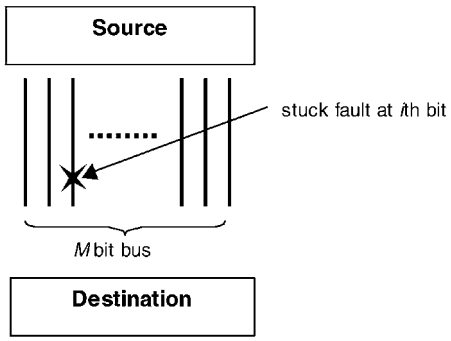


Fig. 4. An *M* bit bus transferring information from a source to a destination unit.

From Section 4.2 to Section 4.5, we derive $C_j(k)$ and $D_j(k)$ for various functional units; then, in Section 4.6, we show how to determine optimal values of k for each benchmark program.

4.2 Bus Signal Line

An M bit bus consists of M parallel signal paths. As shown in Fig. 4, the source places an M bit signal x on the M bit bus. If the ith bit of the bus has a stuck-at fault, the destination may receive a corrupted x. If there is a parity bit on the bus, a single bit error can be detected by parity check. However, if there is no parity on the bus, the stuck-at fault on this bus can be detected by the ED^{4}I technique.

If we put x of the original program and $x'(= kx)$ of the diverse program on the bus one after the other, the fault will affect x and x' in different ways. If the fault modifies either x or x', but not both, the relationship $x' = k \cdot x$ in the destination will not be satisfied. On the other hand, if the fault modifies both x and x', the relationship $x' = k \cdot x$ in the destination may or may not be satisfied. If neither x nor x' provoke the fault, the destination will receive correct values and also satisfy $x' = k \cdot x$. In this case, the fault does not corrupt the information.

If k is -1 or a power of 2, we can get closed form solutions for the data integrity and fault detection probability. The closed form solutions are derived in Appendix 2 and summarized in Table 1. Table 2 shows the fault detection probability $C_j(k)$ and the data integrity probability $D_j(k)$ for different values of k in a 12 bit ($M = 12$) bus as an example. Simulation applying exhaustive input patterns in the presence of faults is used when k is not -1 nor a power of 2. In the table, negative numbers are represented in a 2's complement representation, which is widely used in most microprocessors.

Assuming no overflow in x and x', the data integrity is guaranteed, i.e., $D_j(k) = 1$ as shown in Table 1 and Table 2 (handling overflow is discussed in detail in Section 6).

In Table 2, the highest fault detection probability occurs when $k = -1$. However, as we will see in later sections, the value -1 for k does not guarantee data integrity in adders and multipliers. On the other hand, $C(k)$ is larger when $|k|$ is 2 and 4 compared to the cases when $|k|$ is an odd number such as 3 and 5.

4.3 Adder

An iterative network such as a ripple carry adder consists of multiple logic cells cascaded in series. Each all receives carries from a previous cell and inputs to this cell; then, it produces outputs and, subsequently, generates a carry to the next cell.

The regularity of the array helps us to calculate the fault detection probability when the multiplying factor k is 2^l, where l is an integer. Suppose one node of the cell s_i has a stuck fault. If x is an output of the network, the ith bit of x may be corrupted. In the transformed program, if $x' = kx$ is the output of the array, the ith bit of x' may be corrupted; however, this is equivalent to the i-lth bit of x being corrupted because x' is an l bit shift of x. Therefore, we can calculate the probability of provoking the faults in two cells s_{i-l} and s_i, and get the probability of mismatch between x and x', i.e., $\Pr\{x' \neq kx\}$.

TABLE 1
Closed Form Solutions for $C_j(k)$, $D_j(k)$ in the Presence of a Fault in the ith Bit of an M-bit Bus

	$k = -2^l\ (l > 1)$	$k = -2$	$k = -1$	$k = 2$	$k = 2^l\ (l > 1)$
$C_j(k)$	$\frac{3}{4} + \frac{1}{2^i}\left(1 - \frac{1}{2^{l-1}}\right) - \frac{1}{2^M}$	$\frac{3}{4} - \frac{1}{2^M}$	$1 - \frac{1}{2^{i+1}}$	$\frac{3}{4}$	$\frac{3}{4} + \frac{1}{2^{i+1}}\left(1 - \frac{1}{2^{l-1}}\right)$
$D_j(k)$	1	1	1	1	1

TABLE 2
$C_j(k)$ and $D_j(k)$ in a 12 Bit Bus ($M = 12$)

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
C_j (k)	0.743	0.778	0.759	0.750	0.917	0.750	0.657	0.766	0.673
D_j(k)	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Equation (5) is used to calculate values.

TABLE 3
$C_j(k)$ and $D_j(k)$ in a 12-Bit Ripple Carry Adder with Various k Values

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.643	0.657	0.639	0.662	0.665	0.594	0.563	0.612	0.557
$D_j(k)$	1.000	1.000	1.000	1.000	0.951	1.000	1.000	1.000	1.000

TABLE 4
$C_j(k)$ and $D_j(k)$ in a 12-Bit Carry Look-Ahead Adder with Various k Values

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.609	0.620	0.603	0.629	0.639	0.559	0.537	0.578	0.522
$D_j(k)$	1.000	1.000	1.000	1.000	0.963	1.000	1.000	1.000	1.000

This section considers a ripple carry adder as an example of an iterative network. We analyze the probability that a signal changes from 0 to 1 and from 1 to 0 in each node in the adder and use the method in [9] to compute the probability of provoking the faults. For every single stuck-at fault f, we calculated the values of $C_j(k)$ and $D_j(k)$ using an analysis technique described in Appendix 2. They are shown in Table 3. For a noniterative network such as a carry look-ahead adder, we calculated the values of $C_j(k)$ and $D_j(k)$ using exhaustive simulation of all possible input combinations with various values of k. The numbers are reported in Table 4.

As shown in Tables 3 and 4, we cannot achieve data integrity of 1 when $k = -1$; this explains why our technique is better than [29] in which all variables are negated (same as $k = -1$ in our technique). If k is -1, there are some faults in the adder that will not be detected. For example, suppose one of the XOR gate's output is stuck at 0 in a full adder of the ith stage s_i of the adder (Fig. 5a). If the input **a** is 2^i and the other input **b** is $2^0 = 1$, this fault cannot be detected as shown in Fig. 5b. Due to the fault, **a** + **b** produces an incorrect output 1 and (-**a**) + (-**b**) also produces an incorrect output -1. Because the two outputs satisfy $x' = k \cdot x$, we cannot detect this fault by comparison of the two outputs using $k = -1$. From this observation, we can see that the value -1 is not suitable for k in terms of data integrity.

For the same absolute value of k, a negative k has higher fault detection probability than a positive k. Multiplying by an odd number such as 3 is not as efficient as multiplying by 2, which is just a one bit shifting operation. Therefore, we do not need to choose 3 for k, which is a more expensive operation than shifting by just one bit.

Among the values shown in the Tables 3 and 4, $k = -2$ shows the highest fault detection probability under the condition that data integrity is 1. The maximum difference between the lowest value and the highest value is 0.108.

4.4 Multiplier and Divider

Many different implementations for multipliers and dividers exist and data integrity and fault detection probability depend on the particular design implementation. In this section, we consider a parallel array multiplier [32], Wallace Tree multiplier [33], and a datapath of SRT divider [34] to demonstrate the dependence of $C_j(k)$ and $D_j(k)$ on various values of k. The parallel array and Wallace Tree multipliers have regular cell structures as shown in Fig. 6.

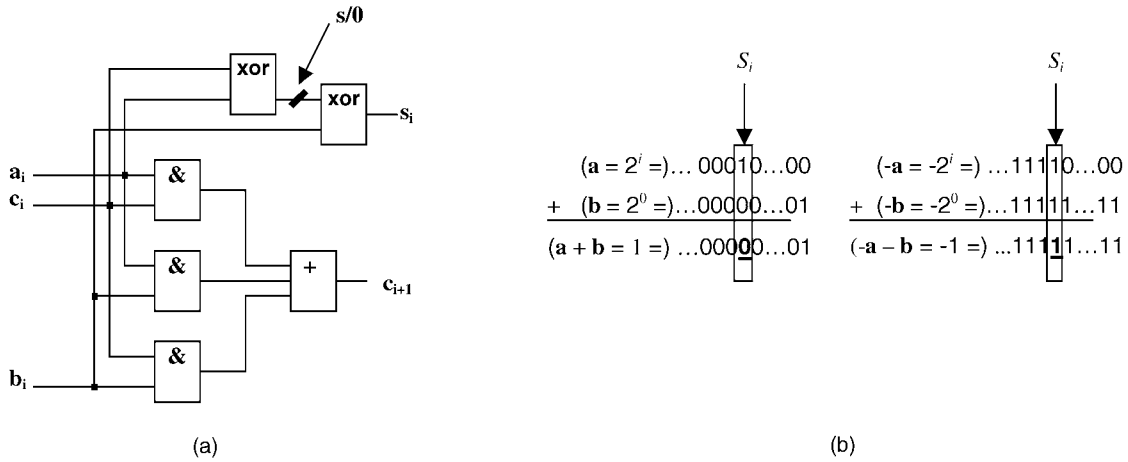


Fig. 5. (a) s/0 at the output of XOR gate in S_i, the ith cell of a ripple-carry adder. (b) The s/0 fault shown in (a) cannot be detected by comparing the two values, although, we have erroneous computation results.

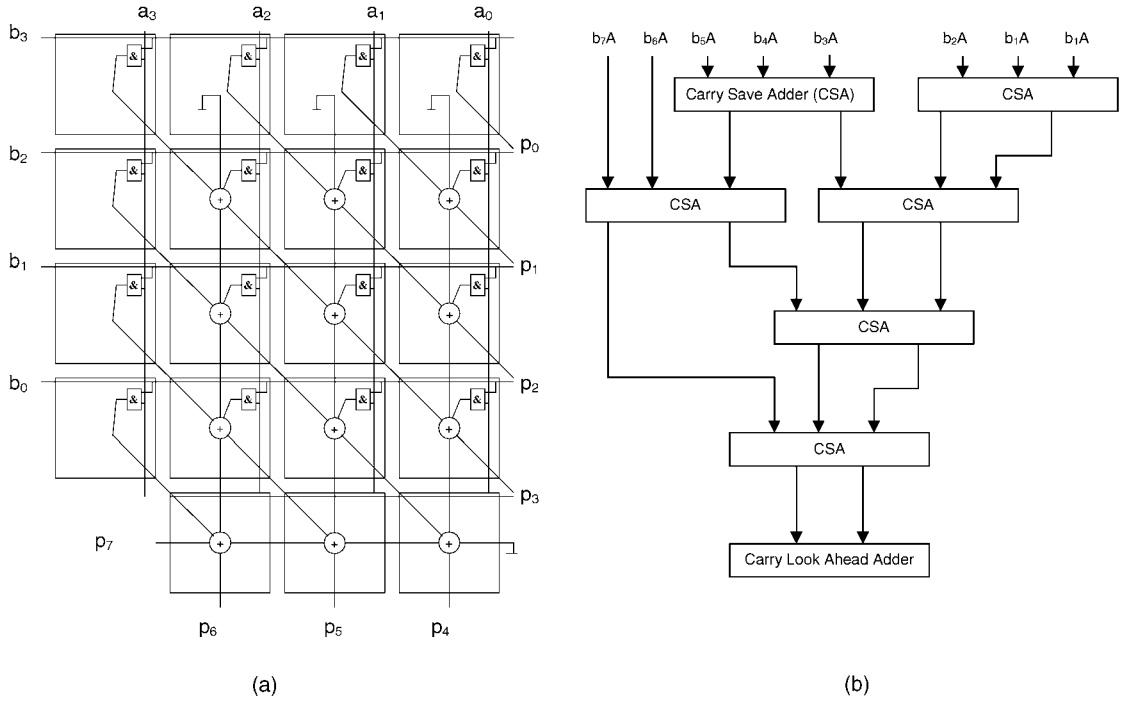


Fig. 6. (a) A 4x4 bit parallel array multiplier. (b) An 8x8 bit Wallace Tree multiplier.

Using a 12-bit array multiplier and an 8-bit Wallace Tree multiplier as examples, we randomly selected a node in the multiplier and injected a stuck-at fault into the node. For example, there are 1,176 nodes in the 12-bit array multiplier. We injected a stuck-at fault into a randomly selected node, applied exhaustive input patterns to the multiplier in the presence of a fault and repeated simulations 10^4 times. We show the result in Table 5. This table shows the positive and negative values of k in the same column because the array multiplier treats negative number multiplication as a positive number multiplication after changing the sign of the negative numbers.

Although the results in Table 5 do not differ by much (the difference between the highest value and the lowest value is about 0.01), it shows us that the highest $C_j(k)$ in the table occurs when $k = 4$. Note that, $k = -1$ is not shown in the table. The multipliers convert the transformed program's negative numbers to original positive numbers before multiplication. This results in the same multiplication in the original and transformed program and, consequently, the fault cannot be detected ($C_j(k) = D_j(k) = 0$). Similarly, we show the results of the divider in Table 5.

4.5 Shifter

Since multiplication or division by a power of 2 can be replaced by shifting operation, shifters are used frequently during program execution [32]. Because the multiplexer-based shifter shown in Fig. 7 [32] is widely used, we will take this design for our simulation.

In our simulation, we injected a stuck-at fault into a randomly selected node and applied exhaustive patterns to 16-bit shifter inputs. This simulation experiment was repeated for 10^4 times over 432 nodes in the shifter. The results are shown in Table 6.

In this shifter implementation, Table 6 shows that the value of -1 for k has the highest fault detection probability with guaranteed data integrity. The maximum difference between the highest value and the lowest value is 0.133.

4.6 Determination of k for Benchmark Programs

We have observed that, in different functional units, we have different values of k that maximize the fault detection probability and data integrity. For example, the bus has the highest fault detection probability when $k = -1$, but the array multiplier has the highest fault detection probability when $k = 4$. Therefore, programs, such as matrix multiplication that use a multiplier extensively, will need 4 or -4 for the value of k to obtain the highest fault detection

TABLE 5
$C_j(k)$ and $D_j(k)$ in Multipliers and Divider

	Parallel array multiplier				Wallace Tree multiplier				Divider			
	k=2,-2	*k*=3,-3	*k*=4,-4	*k*=5,-5	*k*=2,-2	*k*=3,-3	*k*=4,-4	*k*=5,-5	*k*=2,-2	*k*=3,-3	*k*=4,-4	*k*=5,-5
$C_j(k)$	0.598	0.585	0.600	0.589	0.589	0.589	0.593	0.586	0.802	0.783	0.798	0.785
$D_j(k)$	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.956	0.950	0.960	0.953

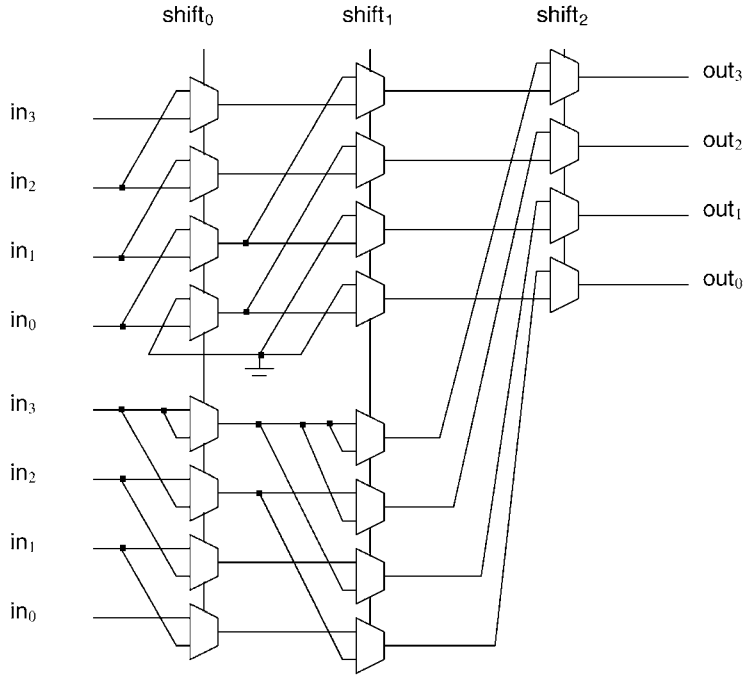


Fig. 7. A 4-bit multiplexer-based shifter.

probability unless there is overflow. In addition, as shown in Section 4.4, $k = -1$ is no help at all for detecting errors in the multiplier. However, for programs such as sorting that use memory buses heavily to communicate with memory for loading and storing data and do not use multipliers, -1 rather than 4 for the value of k will maximize the fault detection probability. Hence, an execution trace of a program showing execution frequencies of each functional unit is necessary to determine the best k for a particular program.

We chose seven benchmark programs from a public benchmark website [35] and obtained execution traces of them using a MIPS simulator (Table 7). The simulator gives us the execution frequency of each functional unit when benchmark programs are executed. The first three entries (add/sub, multiplication, and shift) in the table represent the execution frequencies of instruction types that use an adder, a multiplier, and a shifter, respectively. The fourth entry, a memory access type, is an operation that uses the memory bus. For a branch instruction type, we assume that a carry-look ahead adder is used. The rest of the instruction types such as handling status registers are included in the entry of the last row. In Table 8 and Table 9, $C(k)$ and $D(k)$ are calculated from (6),

$$C(k) = \sum_j u_j C_j(k), D(k) = \sum_j u_j D_j(k),$$

which uses the values of $C_j(k)$ and $D_j(k)$ from Tables 1, 2, 3, 4, 5, and 6.

TABLE 6
$C_j(k)$ and $D_j(k)$ in a 16-Bit Multiplexer-Based Shifter

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
$C_j(k)$	0.357	0.369	0.372	0.375	0.423	0.320	0.290	0.312	0.293
$D_j(k)$	1.000	1.000	1.000	1.000	1.000	1.000	0.996	1.000	0.996

TABLE 7
Execution Frequencies of Instruction Types in Benchmark Programs

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
add/sub	50.7%	50.7%	44.8%	46.4%	51.1%	42.3%	33.7%
multiplication	0%	0%	0%	0%	10.0%	1.0%	0%
shift	5.2%	6.3%	2.5%	0%	1.8%	33.7%	4.0%
memory access	21.6%	23.5%	32.7%	28.6%	29.8%	19.3%	51.5%
branch	22.5%	19.5%	19.4%	25.0%	7.3%	2.7%	10.8%
others	0%	0%	0.6%	0%	0%	1.0%	0%

TABLE 8
Data Integrity $D(k)$ Calculated with Various Values of k in Benchmark Programs

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
I-sort	1.000	1.000	1.000	1.000	0.964	1.000	0.999	1.0000	0.999
Q-sort	1.000	1.000	1.000	1.000	0.966	1.000	0.999	1.0000	0.999
Lzw	0.994	0.994	0.994	0.994	0.963	0.994	0.994	0.9940	0.994
Fib	1.000	1.000	1.000	1.000	0.965	1.000	1.000	1.0000	1.000
M-mul	1.000	1.000	1.000	1.000	0.871	1.000	0.999	1.0000	0.999
Shuffle	0.990	0.990	0.990	0.990	0.958	0.990	0.989	0.9900	0.989
Hanoi	1.000	1.000	1.000	1.000	0.978	1.000	0.999	1.0000	0.999

Note that shaded areas indicate the highest data integrity in a row.

TABLE 9
Fault Detection Probability $C(k)$ Calculated with Various Values of k in Benchmark Program

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
I-sort	0.649	0.662	0.651	0.666	0.707	0.613	0.569	0.626	0.584
Q-sort	0.648	0.661	0.651	0.665	0.709	0.613	0.568	0.626	0.582
Lzw	0.664	0.676	0.668	0.680	0.737	0.635	0.584	0.648	0.599
Fib	0.671	0.684	0.673	0.687	0.737	0.639	0.590	0.654	0.605
M-mul	0.662	0.681	0.665	0.677	0.669	0.636	0.588	0.648	0.602
Shuffle	0.559	0.571	0.565	0.575	0.619	0.526	0.484	0.533	0.495
Hanoi	0.670	0.693	0.677	0.682	0.768	0.652	0.590	0.664	0.605

Shaded areas indicate the highest fault detection probability under the condition that data integrity is the highest (shaded areas in Table 8).

An optimal value of k must satisfy the primary and the secondary goals. The primary goal is to maximize the data integrity and the secondary goal is to maximize the fault detection probability. Table 8 shows the values of k that satisfies the primary goal of maximizing data integrity. Those values are indicated by shaded entries in the table. Under the condition that this primary goal is satisfied, Table 9 shows the value of k that maximizes the fault detection probability (indicated by shaded entry in the table). Therefore, the value in the shaded entry in Table 9 is an optimum k for each benchmark program.

Although the highest fault detection probability occurs when $k = -1$ in six benchmark programs, the value of -1 is not a good choice for k because the data integrity is the lowest when $k = -1$. In those benchmark programs, -2 is the best choice for k because the fault detection probability is maximized (shaded entries in Table 9) under the condition that data integrity is the highest (shaded entries of Table 8).

During the execution of the six benchmark programs (I-sort, Q-sort, Lzw, Fib, Shuffle, and Hanoi), the most frequently used functional units are adders. A carry-look ahead adder and a ripple carry adder have the highest fault detection probability (under the condition that data integrity is highest) when $k = -2$; thus, $k = -2$ is also the optimum value for those programs. On the other hand, the matrix multiplication program extensively uses the multiplier more often than the other six benchmark programs. Because the fault detection probability in the array multiplier is the highest when $k = -4$, the fault detection probability in the matrix program is the highest when $k = -4$.

Finally, Table 10 shows the optimum value of k for each benchmark program.

5 Simulation Results

In Section 4, we determined the optimum value of k for each benchmark program by looking at the execution trace of the program. In this section, we will verify our choice of the optimum value of k by simulating the benchmark programs in a MIPS simulator. We built a MIPS simulator that reads an assembly program, executes instructions, and emulates the functional units at the gate level for a fault injection simulation. We injected a stuck-at fault into a randomly selected node of the functional units and generated output patterns, which may be corrupted if the inputs provoke the fault. We generated two output patterns in each functional unit: one from the original program and the other from the transformed program. We compared their outputs, counted

TABLE 10
Optimum Value of k Determined for Each Benchmark Programs

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
Optimum k	-2	-2	-2	-2	-4	-2	-2

TABLE 11
$D(k)$ With Various Values of k Simulated in Benchmark Programs

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
I-sort	1.000	1.000	1.000	1.000	0.964	1.000	0.999	1.000	0.999
Q-sort	1.000	1.000	1.000	1.000	0.959	1.000	0.999	1.000	0.999
Lzw	0.990	0.994	0.988	0.994	0.994	0.994	0.985	0.994	0.979
Fib	1.000	1.000	1.000	1.000	0.960	1.000	1.000	1.000	1.000
M-mul	1.000	1.000	1.000	1.000	0.972	1.000	0.999	1.000	0.999
Shuffle	0.988	0.980	0.987	0.990	0.968	0.990	0.983	0.990	0.976
Hanoi	1.000	1.000	1.000	1.000	0.831	1.000	0.999	1.000	0.999

Note that shaded areas indicate the highest data integrity in a row.

TABLE 12
$C(k)$ With Various Values of k Simulated in Benchmark Programs

	k = -5	k = -4	k = -3	k = -2	k = -1	k = 2	k = 3	k = 4	k = 5
I-sort	0.676	0.684	0.677	0.690	0.668	0.442	0.454	0.449	0.446
Q-sort	0.681	0.676	0.684	0.686	0.687	0.517	0.485	0.523	0.482
Lzw	0.720	0.716	0.719	0.731	0.740	0.515	0.488	0.532	0.484
Fib	0.697	0.720	0.703	0.725	0.755	0.532	0.504	0.531	0.501
M-mul	0.685	0.691	0.685	0.690	0.641	0.543	0.517	0.553	0.509
Shuffle	0.561	0.560	0.570	0.568	0.602	0.490	0.458	0.498	0.461
Hanoi	0.740	0.765	0.741	0.773	0.777	0.545	0.517	0.559	0.513

Shaded areas indicate the highest fault detection probability under the condition that data integrity is the highest (shaded areas in Table 11).

detected and undetected incorrect outputs, averaged them over all simulated single stuck-at faults, and, finally, obtained $C(k)$ and $D(k)$ of benchmark programs. Using the simulation results shown in Table 11 and Table 12, we determine the optimum value of k in Table 13. The reader can verify that Table 13 is identical to Table 10; this demonstrates that our results in Section 4 agree with the simulation results.

The numbers in Table 11 and Table 12 are slightly different from the numbers in Table 8 and Table 9, in which we assume uniform probability of all numbers in the entire range of the inputs. However, when we simulate benchmark programs in our MIPS simulator, we cannot assume uniform probability of actual inputs to the programs. For example, the sorting program in the simulation does not sort the entire range of integer numbers (from -2^{31} to 2^{31}) available in a 32-bit machine. Instead, it sorts the numbers that are smaller than ten thousand. Thus, there are slight discrepancies between the expected probabilities and the values obtained from simulation.

6 Handling Overflow

The primary cause of the overflow problem in the transformed program is that, after multiplication by k, the size of the resulting data may be too large to fit into the data word size of the processor. For example, consider an integer value of $2^{31}-1$ in a program (with 32-bit 2's complement integer representation). If the value of k is 2, then the resulting integer $(2^{32}-1)$ cannot be represented using 32-bit 2's complement representation. Previous hardware techniques like RESO [26] eliminated this overflow problem by adding extra bit-slices in the datapath. In SIHFT techniques such as ED^{4}I, hardware changes are not possible so that these techniques can not be used with COTS components. The overflow problem can be solved by scaling: scaling up to higher precision or scaling down the original data. We scale up the data type in the program to avoid overflow; data types such as 16-bit single-precision integers can be scaled up to 32-bit double-precision integer data type. Scaling up will cause performance overhead because the size of the data is doubled. On the other hand, scaling down the original data (the same effect as dividing the original data by k instead of multiplying by k) will not cause any performance overhead. However, there is a possibility that scaling down data may cause computation inaccuracy during the execution of the program. In this case, when we compare the scaled down values with the original values, we have to compare the higher order bits that are not affected by scaling down.

TABLE 13
Optimum Value of k Determined for Each Benchmark Program by Simulation

	I-sort	Q-sort	Lzw	Fib	Mat-mul	Shuffle	Hanoi
Optimum k	-2	-2	-2	-2	-4	-2	-2

The other solution to handle overflow is to use range check during compilation. Modern compiler techniques enable us to estimate the range of values for each variable in a program. Harrison [36] presents a technique to determine bounds on the ranges of values assumed by variables at various points in the program. Patterson [37] described using value range propagation for accurate static branch prediction. Stephenson [38] presents a compiler technique that minimizes the bitwidth of registers by finding the minimum number of bits needed to represent each program operand. This data range analysis can be applied to find a maximum bound of variables in the program and used to determine a value *k* that does not cause an overflow, if such *k* exists.

7 Floating Point Numbers

There are several ways that nonintegers can be represented. Examples include using fixed point representation and using a pair of integers (*a, b*) to represent the fraction *a/b*. However, only *floating point* representation has gained widespread use [34]. In this floating point system, a computer word is divided into three parts: a sign, an exponent, and a fraction. As an example, in the IEEE standard 754 [39], single-precision numbers are stored in 32 bits: one bit for the *signs*, eight bits for the *exponent exp*, and 23 bits for the *fraction f*. In this paper, we assume that floating point numbers are represented in the IEEE standard 754 format.

A floating point number in binary can always be written in the form $-1^s \times 1.\text{fff}\ldots\text{f} \times 2^b$, where *s* is a sign, $1.\text{fff}\ldots\text{f}$ is the (base 2) *mantissa* ("*significand*"), and *b* is the *exponent*. The exponent is a signed number represented using the bias representation with a bias of 127. In the biased representation, a bias quantity equal to half of the exponent range is added to each true exponent value to produce a biased exponent *b* (often called the *characteristic*). The fraction *f* in the IEEE standard represents the fraction part of the *mantissa*, i.e., $f = .\text{fff}\ldots\text{f}$. For example, a decimal number 1.5 is $1.1_2 \times 2^0$ in binary representation because $1.5_{10} = 1 + 1/2 = 1_2 + .1_2 = 1.1_2 \times 2^0$. Thus, in the IEEE standard format, *s* is 0, *f* is .1 (the mantissa is 1.1), and *b* is 127 (the biased exponent by adding 127). Since floating point numbers and integers have different representations, floating point computation is usually distinguished from integer computation and performed in a special functional unit called a *Floating Point Unit* (FPU).

Having three parts in one word in floating point representation creates some difficulty in applying ED^{4}I to floating point numbers because multiplying by *k* may not shift nor change many bits in one word. For example, if the value of *k* is -1 and we multiply the original data by this *k*, only one bit of the word (the sign bit in the representation) is changed, and the rest of the word remains unchanged. If *k* is a power of 2, only some bits in the exponent portion *b* are changed, but the fraction part *f* is not changed. For example, consider a decimal number 1.5 again. In binary representation, 1.5_{10} is $1.1_2 \times 2^0 (s = 0, b = 127, f = .1)$. If we multiply 1.5_{10} by 2, the result is 3_{10} in decimal representation and $1.1_2 \times 2^1 (s = 0, b = 128, f = .1)$ in binary representation. As we can see, there is no change in fraction part *f*. In other words, some of the data in the transformed program are not changed no matter whether we transform a program with the factor *k* or not; thus, we cannot guarantee data integrity if we have a stuck-at fault in the fraction part.

The first approach to change all bits in floating point numbers is to extract the sign, fraction, and exponent bits from floating point numbers and complement all bits [40]. However, extracting and merging bits require a high overhead in code size and execution time. For example, in MIPS R10000 processor, it takes three consecutive cycles to extract the sign, fraction, and exponent bits and two cycles to merge those bits; the execution time overhead is already over 400 percent only with extracting and merging bits.

The second approach is to find a value of *k* for the fraction (k_f) and the exponent (k_{exp}) separately and use those values to get the value of *k*. Detecting a fault in the exponent field requires two transformations with two values of *k*; thus, we need to execute the transformed program twice and the execution time overhead of our technique is over 100 percent for floating point numbers compared to integer numbers. In addition, scaling up to double precision is necessary to avoid overflow in the exponent field of floating point numbers. Therefore, overall execution time overhead will be over 200 percent.

In this section, we consider the second approach because the transformation algorithm discussed in Section 3 is also applicable to this approach, and it has a lower overhead than the first approach.

7.1 The Value of k for Fraction

First, let us consider the fraction part. We choose $k = k_f = \frac{3}{2}$ for the fraction part because it satisfies the following criteria: 1) guaranteed data integrity in the fraction, 2) no underflow in the transformed program, and 3) low probability of overflow.

1) Guaranteed Data Integrity.
Let us denote a floating point number *x* as $-1^s \times m \times 2^b$, where *s* is a sign, *m* is a mantissa, and *b* is an exponent. Note that the mantissa *m* is always $1 \le m < 2$. Suppose stuck-at 1 occurs and affect the *i*th bit (from the most significant bit) of the fraction of the mantissa *m*; then, the stuck-at fault will add $e = 2^{-i}$ to *m* if it is provoked.

1) In the original program.
If we denote x_e as a corrupted *x* by this stuck-at 1 fault,

$$x_e = -1^s \times (m + e) \times 2^b, 1 < m + e < 2. \tag{7}$$

2) In the transformed program.

$$x' = k\,x = \frac{3}{2}x = -1^s \times \frac{3}{2}m \times 2^b.$$

Since $1 \le m < 2$, $\frac{3}{2} \le \frac{3}{2}m < 3$.

However, a mantissa cannot be greater than 2; thus, if $\frac{3}{2}m > 2$, the mantissa is right-shifted by 1 (divided by 2) and normalized. This normalization will add one to the exponent. In other words,

$$x' = \begin{cases} -1^s \times (\frac{3}{2}m) \times 2^b, \textit{if } \frac{3}{2}m < 2 \\ -1^s \times (\frac{3}{4}m) \times 2^{b+1}, \textit{if } 2 \le \frac{3}{2}m < 3. \end{cases}$$

The same stuck-at 1 fault corrupts the *i*th bit of the fraction of the mantissa and adds $e = 2^{-i}$ as it did in the original program. If we denote x'_e as a corrupted x' by this stuck-at 1 fault,

$$x'_e = \begin{cases} -1^s \times (\frac{3}{2}m + e) \times 2^b, if\ \frac{3}{2}m < 2 \\ -1^s \times (\frac{3}{4}m + e) \times 2^{b+1}, if\ 2 \le \frac{3}{2}m < 3. \end{cases} \quad (8)$$

3) Comparison.
We have to check if the value of x'_e is erroneously equal to the value of kx_e. If the values are the same, the data integrity is not guaranteed. From (7),

$$k\,x_e = \frac{3}{2}x_e = -1^s \times \frac{3}{2}(m+e) \times 2^b = \begin{cases} -1^s \times \frac{3}{2}(m+e) \times 2^b, if\ \frac{3}{2}(m+e) < 2 \\ -1^s \times \frac{3}{4}(m+e) \times 2^{b+1}, if\ 2 \le \frac{3}{2}(m+e) < 3. \end{cases} \quad (9)$$

The mantissas in (8) are $\frac{3}{2}m + e, \frac{3}{4}m + e$. The mantissas in (9) are $\frac{3}{2}m + \frac{3}{2}e, \frac{3}{4}m + e$. When we compare the mantissas in (8) with those in (9), they are all different values. Thus, x'_e never equals to kx_e, and the stuck-at 1 fault can be detected by comparing these two values. Therefore, data integrity is guaranteed.

Let us show an example. Consider $x = 1.5$ again: The fraction f_x is .1. By multiplying k = 3/2, x' is 1.001×2^1; thus, the fraction is $f_{x'} = .001$. Suppose a stuck-at 1 fault occurred in the second most significant bit of the fraction and changes $f_{x'}$ to .011, and $f_{x'}$ to .11. Therefore, $x = 1.11 \times 2^0$ and $x' = 1.011 \times 2^1$. When we compare x' with $kx\ (= 1.0101 \times 2^1)$, we can detect the error by mismatch of the two values.

Similarly, we can prove that data integrity is also guaranteed when a stuck-at-0 fault occurs.

(2) No Underflow.
Since $k > 1$, we do not have an underflow in the floating point number of the transformed program.

(3) Low Probability of Overflow.
When we multiply the data by $k < 2$, the exponent of the original data increases at most by one. If a maximum value of data is less than $2^{2^8-2-127} = 2^{127}$ in IEEE standard 754 format, we can guarantee no overflow in the transformed program.

7.2 The Value of k for Exponent

Second, let us consider the value of *k* for the 8-bit exponent of the floating point number. We use two transformations to guarantee data integrity in the exponent. We choose $k = k_{exp} = 2^{10101010_2}$ for the first transformation and $k = k_{exp} = 2^{01010101_2}$ for the second transformation. In this case, we have assumed that these *k*'s will not cause an overflow. (However, if there is an overflow, we can use the scaling techniques described in Section 6 such as changing 32-bit single precision format to 64-bit double precision format.)

When we multiply the original value by *k*, the exponent of *k*, i.e., k_{exp} is added to the exponent of the original value. Thus, 10101010_2 is added to the exponent of the original data in the first transformation, and 01010101_2 is added to the exponent of the original data in the second transformation. We will prove later that every bit of the exponent of the original value can be complemented either by the first transformation or by the second transformation. If every bit of the exponent field can be complemented in the transformed programs, a single stuck-at fault in the exponent field can be detected.

First, suppose a stuck-at fault is provoked and produced an error in the original program. Since every bit in the exponent can be complemented in the transformed programs, the stuck-at fault will not be provoked in at least one of the transformed programs. Thus, when we compare the corrupted value in the original program with the uncorrupted value in the transformed program, we can detect the error. Second, suppose the stuck-at fault is not provoked in the original program. Then, it will be provoked in at least one of the transformed programs and the data will be corrupted. We can detect the error by comparing the corrupted value with the uncorrupted value. Therefore, a single stuck-at fault in the exponent field can be detected.

Now, let us prove that every bit of the exponent of the original value is complemented either by the first transformation or by the second transformation.

Proof. Let an exponent $n < 2^N$ can be represented by $n = \sum_{i=0}^{N-1} b_i 2^i$, where $b_i = 0$ or 1, and *N* is *the number of bits in the exponent*. For an integer $0 < h < N$, define n_h as $n_h = n\ \&\ (2^h - 1) = \sum_{i=0}^{h-1} b_i 2^i$, where & represents *logical AND* operation; then, n_h is always $0 \le n_h < 2^h$. First, let us assume *h* is an even number. Define p, p_h, q, and q_h as:

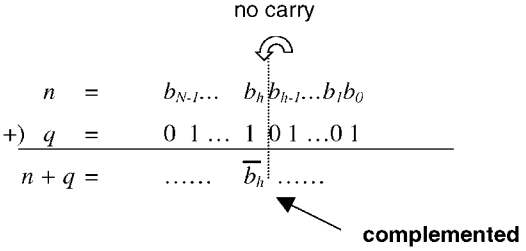
$$p = \sum_{i=0}^{\frac{N}{2}-1} 2^{2i+1},$$
$$p_h = p\ \&\ (2^h - 1),$$
$$q = \sum_{i=1}^{\frac{N}{2}-1} 2^{2i},$$
$$q_h = q\ \&\ (2^h - 1).$$

Fig. 8 shows the numbers we have defined.

Now, let us define z_h as $z_h = 2^h - q_h < 2^h$; then, z_h represents the smallest number of n_h that can produce a carry from the bit b_{h-1} to b_h when it is added by q.

1) When $0 \le n_h < z_h$.

$n + q$ does not produce a carry from b_{h-1} to b_h because n_h is less than z_h. Therefore, b_h is complemented by adding q.

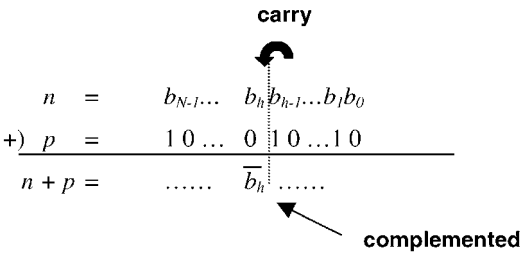


$$
\begin{array}{lcl}
n & = & b_{N-1}\ldots \; b_h b_{h-1}\ldots b_1 b_0 \\
n_h & = & 0\;0 \quad\; 0\; b_{h-1}\ldots b_1 b_0 \\
 & & \\
p & = & 1\;0\ldots \; 0\;1\;0\ldots 1\;0 \\
p_h & = & 0\;0\ldots \; 0\;1\;0\ldots 1\;0 \\
 & & \\
q & = & 0\;1\ldots \; 1\;0\;1\ldots 0\;1 \\
q_h & = & 0\;0\ldots \; 0\;0\;1\ldots 0\;1
\end{array}
$$

Fig. 8. n, n_h, p, p_h, q, and q_h.

2) When $z_h \leq n_h < 2^h$.

Since $p_h > q_h, n_h + p_h > n_h + q_h > z_h + q_h = 2^h$; thus, $n + p$ will always produce a carry (2^h) from b_{h-1} to b_h. Therefore, b_h is complemented by adding p.



From 1 and 2, b_h for any even number $0 < h < N$ is always complemented by either p or q. Similarly, we can prove that b_h is always complemented by either p or q when $0 \leq h < N$ is an odd number or zero. Therefore, every bit of n is always complemented by either p or q.□

7.3 The Value of k for Floating Point Numbers

We have selected $k_f = \frac{3}{2}$ for detecting an error in the fraction of a floating point number representation. We also have selected $k_{exp} = 2^{10101010_2}$ and $k_{exp} = 2^{01010101_2}$ for the exponent. Moreover, we need to negate the original data to detect a fault in the sign bit. As a result, we use $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ for the first transformation and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$ for the second transformation.

In a simulation experiment using $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$, we analyzed the data integrity and the fault detection probability in the bus, CLA adder, and multiplier. We applied all possible binary input patterns to these functional units in the presence of a randomly selected fault and checked whether the fault was detected or not. We repeated simulations 10^4 times for each functional unit. Fig. 9 shows the simulation results. Fig. 9a shows that the data integrity of the fraction part is guaranteed in these functional units. In Fig. 9b for the exponent, we can see that the data integrity is guaranteed in the bus, but not in the CLA adder. If the CLA adder has a stuck-at fault in some of the carry signal lines, we miss less than 3 percent of the injected faults. Also, note that the multiplier is not shown in (b) because exponent multiplication is not required in floating point addition, subtraction, multiplication, and division.

The simulation results show that using $k_1 = -\frac{3}{2} \times 2^{10101010_2}$ and $k_2 = -\frac{3}{2} \times 2^{01010101_2}$ guarantees data integrity for all floating point computations except exponent addition. Only in the exponent addition, data integrity is not guaranteed for 100 percent, but it is still higher than 97 percent.

8 Conclusion

In this paper, we have presented the ED^{4}I technique, which is based on data diversity, for detecting hardware faults without any hardware modifications. Unlike previous techniques, our technique is a pure software method that can be easily implemented in any system. ED^{4}I transforms a program to a new program with the same functionality but with diverse data without changing the complexity of the original program.

The diversity factor k determines how diverse the transformed program is. To determine the optimum value of k, we have used data integrity and fault detection

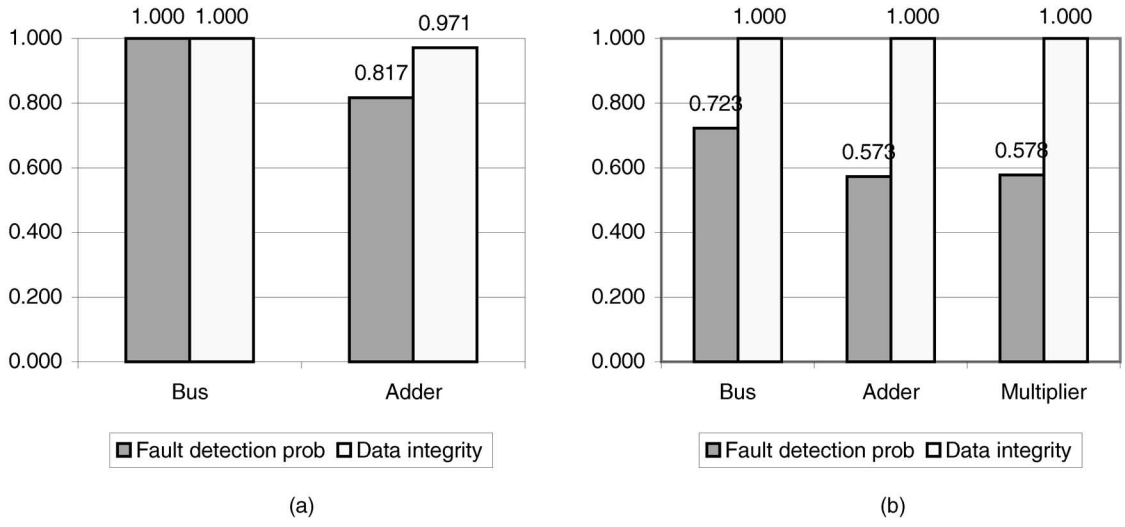


Fig. 9. Simulation results: Data integrity and fault detection probability in (a) fraction and (b) exponent of IEEE 754 single precision.

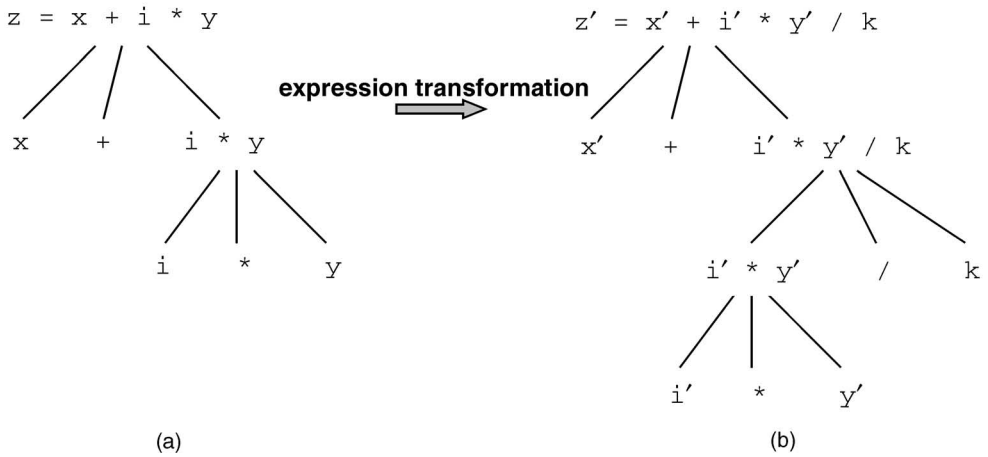


Fig. 10. An example of a parse tree of $z = x + i * y$ (a) before transformation (b) after transformation.

probability as metrics to quantify the diversity of the transformed program. Based on these metrics, we demonstrated how to choose the optimum value of k for creating data diversity. Program execution traces and fault injection simulations in functional units are necessary to get the optimum value for k because data integrity and fault detection probability depends on the implementation of functional units.

Our technique plays an important role when hardware design is fixed (such as COTS components or IP blocks in SoC design) and cannot be modified in embedded systems that operate in safety and mission-critical applications. Furthermore, embedded systems in applications such as handheld computers and mobile phones can employ ED^{4}I to improve the data integrity of the system.

APPENDIX A
PROGRAM TRANSFORMATION ALGORITHM

An expression $expr(x_1, x_2, \ldots, x_n)$ represents arithmetic or logical operations on variables $x_1, x_2, \ldots, x_n$. An *assignment* statement $y := expr(x_1, x_2, \ldots, x_n)$ defines y by assigning y the result of $expr(x_1, x_2, \ldots, x_n)$. A *branch* statement *if* $expr(x_1, x_2, \ldots, x_n)$ determines the control flow according to the result of $expr(x_1, x_2, \ldots, x_n)$. A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without branching except at the end.

By defining $V = \{v_1, v_2, \ldots, v_n\}$ as the set of vertices denoting basic blocks, and $E = \{(i, j) \mid (i, j)$ is a branch from v_i to $v_j\}$ as the set of edges denoting possible flow of control between the basic blocks, a program can be represented by a *program graph*, $P_G = \{V, E\}$.

Program transformation consists of two parts: expression transformations in basic blocks and branching condition transformations in branch statements. The expression transformation converts all the variables and constants in P' to k-multiples of corresponding variables and constants in P. Since the values of the variables and constants are different in P and P', the branching condition transformation modifies the expression in the branch statement and keeps the control flow in P' the same as the one in P.

A.1 Expression Transformation

Let $x'_1, x'_2, \ldots, x'_n$ be the variables in the transformed program P' that correspond to the variables $x_1, x_2, \ldots, x_n$ in P; then, $k \cdot x_i = x'_i$, $i = 1, 2, \ldots, n$ should be always true during runtime. Furthermore, any expression $expr'(x'_1, x'_2, \ldots, x'_n)$ in P' corresponding to $expr(x_1, x_2, \ldots, x_n)$ in P also should satisfy the relationship: The value of $k \cdot expr(x_1, x_2, \ldots, x_n)$ equals to the value of $expr'(x'_1, x'_2, \ldots, x'_n)$, which is achieved by the expression transformation to be described in this section.

An expression $expr(x_1, x_2, \ldots, x_n)$ can be recursively represented by a *parse tree*, a tree in which the leaves are the variables $x_1, x_2, \ldots, x_n$ or constants. An interior node in the parse tree has three children: the child on the left and right is either a node or a leaf, and the child in the middle denotes an arithmetic operation. An example is shown in Fig. 10.

A parse tree T is built from an expression $expr(x_1, x_2, \ldots, x_n)$ in P. Suppose T' represents a parse tree for a corresponding new expression $expr'(x'_1, x'_2, \ldots, x'_n)$ in P'; then, the values represented by the internal nodes in T' should be k-multiple of the values in the corresponding nodes in T.

An algorithm recursively transforming $expr(x_1, x_2, \ldots, x_n)$ to $expr'(x'_1, x'_2, \ldots, x'_n)$ follows:

```
1 transform (t) {
2    if (leaf)
3         if (constant) return x' = k * x;
4         else return x'
5    else {
6         left_child(t') := transform
             (left_child(t));
7         right_child(t') := transform
             (right_child(t));
8         middle_child(t') := middle_child(t);
9         if (middle_child(t) = * (or /)) {
10             right_child(t'') := k;
11             left_child(t'') = t';
12             middle_child(t'') = / (or *);
```

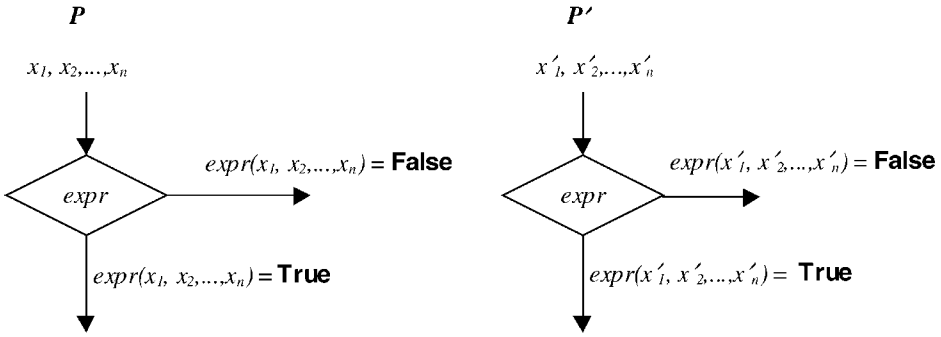


Fig. 11. Branch determined by the expression in a branch statement.

```
13            return t'';
14        } else {
15            return t';
16        }
17    }
18 }
```

Theorem 1.1. *The internal nodes in $T' =$ `transform` (T) are always k-multiple of the corresponding nodes in T.*

Proof. By induction.

Base case of the induction. All the leaves in T' are k-multiple of the leaves of T by line 2, 3, and 4 of the algorithm.

Induction step. Let us represent n as the result of the operation with left and right child of one internal node of T, and n' as the result of the corresponding node of T'. Assume that left and right child of the internal node in T' are k-multiple of the left and right child of the corresponding node in T. If the operation is an addition, n' is still k-multiple of n, i.e., $k \cdot n = n'$. If the operation is multiplication, n' is

$$n' = k \cdot \text{left\_leftchild} \cdot k \cdot \text{right\_child} = k \cdot k \cdot n.$$

Line 9 to 12 eliminates this extra k and keeps $n' = k \cdot n$. Similarly, it is also true that $n' = k \cdot n$ in division. Therefore, `transform`(t) always returns t' such that the value of the top node of t' is always k-multiple of the top node of t. □

A.2 Branching Condition Transformation

The branch statement compares two values (or expressions) and determines the control flow based on the comparison result. The expression in the branch statement has binary values: true or false. For example, if the expression in the branch statement is true, the branch is taken. If it is false, the branch is not taken. The branch statement can be represented by a decision triangle as shown in Fig. 11. If $expr(x_1, x_2, \ldots, x_n)$ is true, the branch is taken, and otherwise, the next statement is executed. This control flow should be preserved in $expr'(x'_1, x'_2, \ldots, x'_n)$ in P', i.e., when $k \cdot x_i = x'_i$, $expr'(x'_1, x'_2, \ldots, x'_n)$ always takes the same value (true or false) as $expr(x_1, x_2, \ldots, x_n)$ in P so that the control flow in P' always identical to the one in the original program.

An algorithm for branching condition transformation is:

1. $expr(x_1, x_2, \ldots, x_n)$ in a branch statement is transformed to $expr'(x'_1, x'_2, \ldots, x'_n)$ by expression transformation.
2. If $expr(x_1, x_2, \ldots, x_n)$ contains $\geq$, >, $\leq$, or <.
3. If $k < 0$, then $\geq$, >, $\leq$, or < is converted to $\leq$, <, $\geq$,or >, respectively.

Theorem 1.2. *The branching condition transformation makes the two control flows determined by $expr(x_1, x_2, \ldots, x_n)$ and $expr'(x'_1, x'_2, \ldots, x'_n)$ always identical.*

Proof. Inequality relationship such as $x_1 < x_2$ is preserved when both sides are multiplied by positive k. If multiplied by negative k, the inequality is reversed as in $x_1 > x_2$ by line 2 and 3. Therefore, if $k \cdot x_i = x'_i$, $expr(x_1, x_2, \ldots, x_n)$, and $expr'(x'_1, x'_2, \ldots, x'_n)$ have the same true or false value, and the control flows determined by $expr(x_1, x_2, \ldots, x_n)$ and $expr'(x'_1, x'_2, \ldots, x'_n)$ are the same. □

Fig. 12 illustrates an example control flow when $k \cdot x_i = x'_i$, and $k = -2$. The branch statement expression $expr(x) = \{x < 5\}$ in (a) is transformed to $expr'(x') = \{x' > -10\}$ in (b). When $x = \{7, 9\}$, the branches are taken. When $x' = \{-14, -18\}$ that is k-multiple of $x = \{7, 9\}$, $k = -2$, the branches are also taken in the transformed branch statement. Similarly, the branches are not taken both in $x = \{1, 3\}$ and $x' = \{-2, -6\}$, and the two control flows by the original and transformed expressions are identical.

A.3 Program Transformation

If $k \cdot S(n) = S'(n)$ after n vertices (basic blocks) are executed, Theorem 1.1 tells us that $k \cdot S(n+1) = S'(n+1)$ after one more vertex is executed. Also consider a branch statement s in P and the corresponding branch statement s' in P'. If $k \cdot S(n) = S'(n)$, the control flows determined by s and s' are identical by Theorem 1.2.

1.1 If $k \cdot S(n) = S'(n)$, then $k \cdot S(n+1) = S'(n+1)$.
1.2 For $\forall v_i \in V$ that have a branch statement s_i at the end and are executed after m-1 vertices are executed: if $k \cdot S(m) = S'(m)$ and s_i and s'_i are executed, then branching determined by s' is always identical to branching by s.

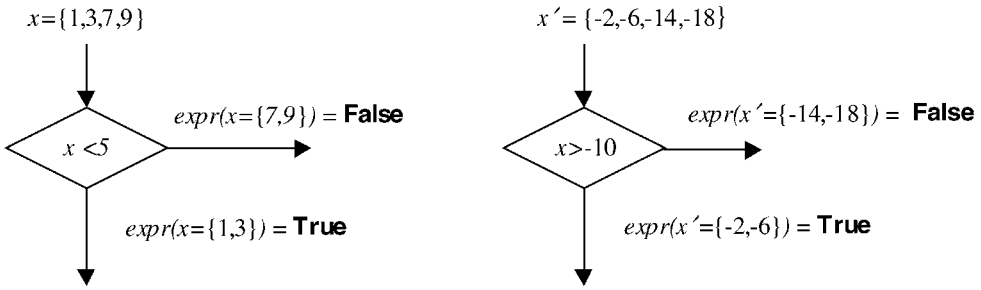


Fig. 12. Branching in the original and transformed expressions when $k = -2$.

Let us denote the first basic block in P and P' by v_0 and v'_0, and assume that $S(0) = S'(0)$ when the program starts. This constitutes the base case of the induction, and the inductive Step 1.1 and 1.2 inductively proves that the control flows in P and P' are always identical. This proof leads us to Theorem 1.3, which shows us that during run time, the variables in the transformed program are always kept as k-multiple of the corresponding variables of the original program.

Theorem 1.3. *For $n > 0$, it is always true that $k \cdot S(n) = S'(n)$.*

APPENDIX B

This section discusses how to get a closed form solution for an optimal value of k that maximizes fault detection probability under the condition that data integrity is maximized. We discuss a bus as an example. We can apply this technique to iterative networks.

An M bit bus consists of M parallel signal paths. As in Fig. 4, the source places an M bit signal x on the M bit bus and transfers information to the destination. If the ith bit of the bus has a stuck fault, the destination may receive a corrupted value of x.

We can detect this fault by applying x of the original program and $x'(= kx)$ of the transformed program on the bus one after the other and comparing the received values [41]. If the fault corrupts either x and x' in a different way at the destination after x and x' are received , $x' = k \cdot x$ will not be satisfied. If x and x' does not provoke the fault, the destination receives the correct values and $x' = k \cdot x$ is satisfied, i.e., the fault does not corrupt the information.

B.1 $k = -1$

In 2's complement representation, negating a number is equivalent to reversing all the bits until the first 1 (from LSB) as shown in Fig. 13a. Suppose that $\mathbf{b_i}$ is the first 1 in x. When x and $x'(= kx = -x)$ are applied on the bus, all the bits from $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ in $x'$are complements of the values of $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ in x. Because each bit from $\mathbf{b_{i+1}}$ to $\mathbf{b_{M-1}}$ can have the value of 0 and 1 by x and x', any stuck fault in those signal lines are provoked and the fault can be detected by comparing two values.

Theorem 2.1. *Assuming that input x is randomly chosen with equal probability, if a stuck-at fault exists on the bit $\mathbf{b_i}$ of the bus, the data integrity of x and x' for the bus is 1 when $k = -1$ and*

$$\Pr\{k \cdot x \neq x'\} = 1 - \frac{1}{2^{i+1}}.$$

Proof. Fig. 13b helps us to calculate $\Pr\{kx \neq x'\}$ when a stuck fault is present on the ith bit of the bus. Assume we have an M-bit bus. Since the MSB bit is a sign bit, 2^{M-1} positive numbers can be represented. If x is positive, x' is always negative. If x is negative, x' is always positive. Thus, we need to consider only positive x to calculate $\Pr\{kx \neq x'\}$. These numbers are grouped according to the position of the first 1 from $\mathbf{b_0}$, as shown in Fig. 13b. In Fig. 13, x represents the bit whose value is complemented when the number is negated. First, suppose $\mathbf{b_i}$ is stuck-at-0 (we also denote stuck-at-0 as s/0). The numbers from the first row to the $i+1$th row can provoke the fault because either x or x' is always 1 as shown in Fig. 13b.

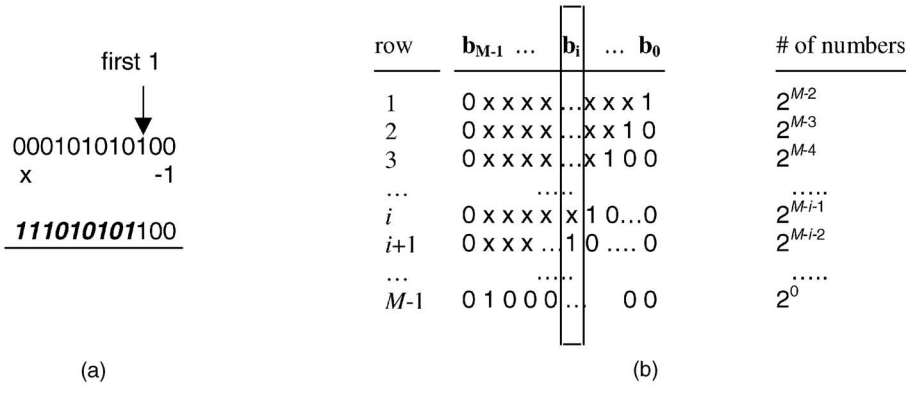


Fig. 13. Negating numbers complements the bits until the first 1 from LSB $\mathbf{b_0}$.

Therefore, by adding the numbers shown in the entries from the first row to the $i+1$th row of the last column in Fig. 13, we can calculate $\Pr\{kx \neq x'\}$ as:

$$\Pr\{k \cdot x \neq x' \mid s/0 \text{ at } \mathbf{b_i}\} = \frac{1}{2^{M-1}}\left(2^{M-2} + 2^{M-3} + \ldots + 2^{M-i-2}\right) = 1 - \frac{1}{2^{i+1}}.$$

Second, suppose the *i*th bit is s/1; then, the numbers except for the $i+1$th row can provoke the fault.

$$\Pr\{k \cdot x \neq x' \mid s/1 \text{ at } \mathbf{b_i}\} = \frac{1}{2^{M-1}}\left(2^{M-2} - 2^{M-i-2}\right) = 1 - \frac{1}{2^{i+1}}.$$

Therefore,

$$\begin{aligned}&\Pr\{k \cdot x \neq x'\}\\ &= \frac{1}{2}\{\Pr\{k \cdot x \neq x' \mid s/0 \text{ at } \mathbf{b_i}\}\\ &+ \Pr\{k \cdot x \neq x' \mid s/1 \text{ at } \mathbf{b_i}\}\}\\ &= 1 - \frac{1}{2^{i+1}}.\end{aligned}$$

□

B.2 $k = 2$

Multiplying by 2 is equivalent to shifting one bit to the left (assuming the most significant bit MSB is on the left). Suppose we have an *M* bit bus and the bit *i* has stuck-at-0 fault. If a variable *x* is one of the variables in the original program and x' is a corresponding variable in the diverse program, x' is one bit left shifted *x*. When *x* is applied to this bus, the *i*th bit of *x* is corrupted if it is 1 and the corrupted value is $x - 2^i$. If x' is applied to the bus and the *i*th bit is corrupted, it could be seen as the same case as the $i-1$th bit of *x* is corrupted and left shifted; so, the corrupted value is $x' = 2(x^{i-1}) = 2x - 2^i = kx - 2^i \neq kx$. Hence, if at least one of *x* and x' is corrupted, $k \cdot x = x'$ is not satisfied, so the comparison of two values will detect the fault. If neither of them is corrupted—both the $(i-1)$th and *i*th bit of *x* are zero—the fault (stuck-at-0) does not affect the output, and $k \cdot x = x'$ is satisfied.

Theorem 2.2. *Assume that input x is randomly chosen with uniform probability and no overflow in the bus. If $k = 2$ and a stuck-at fault exists in a bus, we can detect the fault with the probability of $\frac{3}{4}$ and the data integrity for the bus is 1.*

Proof. Suppose the *i*th bit of an *M* bit bus is stuck-at-0. Consider the $i-1$ and *i*th bit of *x* that is applied to this bus. The s/0 fault is provoked if the *i*th bit of *x* is 1. The $i-1$th bit of *x* corresponds to the *i*th bit of x' if *x* is shifted by one bit left; thus, if the $i-1$th bit of *x* is 1, the s/0 fault is provoked. Four combinations are possible for the two bits: 00, 01, 10, and 11 as shown in Fig. 14. Three of them have at least one 1 and they can provoke the fault so that $k \cdot x = x'$ is not satisfied. Similarly, s/1 fault is also provoked by the three of four possible combinations; therefore, if a stuck fault is present on the bus, $\Pr\{k \cdot x \neq x'\} = \frac{3}{4}$ and, with this probability, we can detect the fault by mismatch.

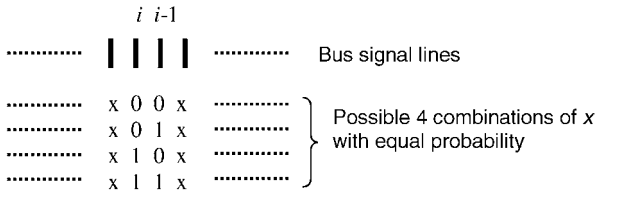


Fig. 14. A bus signal lines applied by x.

If the fault is not provoked, it does not change *x* or x'. Thus, $k \cdot x = x'$ is satisfied. Therefore, the data integrity of *x* and x' for the bus is:

$$1 - \Pr\{k \cdot x = x', \text{and both of } x \text{ and } x' \text{ are corrupted in the same way}\} = 1 - 0 = 1.$$

□

B.3 $k = -2$

Multiplying -2 is equivalent to negating a number and shifting it one bit to the left. Suppose $\mathbf{b_j}$ is the first 1 in *x*. All the bits from $\mathbf{b_{j+1}}$ to $\mathbf{b_{M-1}}$ in *x* are complemented and left shifted by one bit when *x* is multiplied by -2. Each bit from $\mathbf{b_{j+1}}$ to $\mathbf{b_{M-1}}$ has the same probability of $\frac{1}{2}$ for 0 and 1, and the probability remains the same although the bit is complemented. If $\mathbf{b_i}, j < i \leq M-1$ is s/0 in the bus, 01, 10, and 11 in $\mathbf{b_i}$, $\mathbf{b_{i-1}}$ will provoke the fault and the probability of detecting the fault is $\frac{3}{4}$. Similarly, the s/1 in $\mathbf{b}_i$ is provoked with the probability of $\frac{3}{4}$.

Theorem 2.3. *Assuming that input x is randomly chosen with uniform probability if a stuck-at fault exists in $\mathbf{b_i}$, the data integrity for the bus is 1 when $k = -2$ and*

$$\Pr\{k \cdot x \neq x'\} = \frac{3}{4} - \frac{1}{2^M}.$$

Proof. Suppose $\mathbf{b_i}$ is s/0 as shown in Fig. 13. The numbers from the first row to the $i-1$th row can provoke the fault with the probability of $\frac{3}{4}$ since the values of the bits denoted by x are complemented and shifted when multiplied by -2. The number of those numbers is $\left(2^{M-2} + 2^{M-3} + \ldots + 2^{M-i}\right)$. The numbers represented by the *i*th and $i+1$th rows can provoke the fault with probability 1 since $\mathbf{b_{i-1}}$ and $\mathbf{b_i}$ are always 1, respectively. The number of those numbers is $2^{M-i-1} + 2^{M-i-2}$. Therefore, we can calculate $\Pr\{kx \neq x' \mid s/0 \text{ at } \mathbf{b_i}\}$ as:

$$\begin{aligned}&\Pr\{kx \neq x' \mid s/0 \text{ at } \mathbf{b_i}\}\\ &= \frac{1}{2^{M-1}}\left|\frac{3}{4}\left(2^{M-2} + 2^{M-3} + \ldots + 2^{M-i}\right) + 2^{M-i-1} + 2^{M-i-2}\right|\\ &= \frac{1}{2^{M-1}}\left|\frac{3}{4}2^{M-1}\left(2^{i-1} - 1\right) + 2^{M-i-1}\left|1 + \frac{1}{2}\right|\right| = \frac{1}{2^{M-1}}\left|\frac{3}{4}2^{M-1}\right|\\ &= \frac{3}{4}.\end{aligned}$$

Now, suppose $\mathbf{b_i}$ is s/1. The numbers from the first row to the $i-1$th row can provoke the fault with the probability of $\frac{3}{4}$. The number of those numbers is $2^{M-2} + 2^{M-3} + \ldots + 2^{M-i}$. Half of the numbers in the *i*th row can provoke the fault because $\mathbf{b_{i-1}}$ is always 1 and $\mathbf{b_i}$ can be either 0 or 1 with equal probability. The

number of these numbers is 2^{M-i-1}. The $i+1$th row to the last row can always provoke the fault because $\mathbf{b_{i-1}}$ or $\mathbf{b_i}$ is always 0. The number of these numbers is $2^{M-i-1}+2^0$. Thus, we can calculate $\Pr\{k\cdot x \neq x' | s/1 \text{ at } \mathbf{b_i}\}$ as:

$$\begin{aligned}
&\Pr\{kx \neq x' \mid s/1 \text{ at } \mathbf{b_i}\} \\
&= \frac{1}{2^{M-1}}\left|\frac{3}{4}\left(2^{M-2}+2^{M-3}+\ldots+2^{M-i}\right)+2^{M-i-1}\cdot\frac{1}{2}\right. \\
&\quad \left.+\left(2^{M-i-2}+\ldots+2^0\right)\right| \\
&= \frac{1}{2^{M-1}}\left|\frac{3}{4}2^{M-i}\left(2^{i-1}-1\right)+2^{M-i-1}\frac{3}{2}-1\right| = \frac{1}{2^{M-1}}\left|\frac{3}{4}2^{M-1}-1\right| \\
&= \frac{3}{4}-\frac{1}{2^{M-1}}.
\end{aligned}$$

Therefore, if M is large enough, the probability approximates to:

$$\begin{aligned}
&\Pr\{k\cdot x \neq x'\} \\
&= \frac{1}{2}\{\Pr\{k\cdot x \neq x' \mid s/0 \text{ at } \mathbf{b_i}\}+\Pr\{k\cdot x \neq x' \mid s/1 \text{ at } \mathbf{b_i}\}\} \\
&= \frac{1}{2}\left\{\frac{3}{4}+\frac{3}{4}-\frac{1}{2^{M-1}}\right\} = \frac{3}{4}-\frac{1}{2^M} \approx \frac{3}{4}.
\end{aligned}$$

□

ACKNOWLEDGMENTS

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate, and administered through the US Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047, and in part by the US National Aeronautics and Space Administration and administered through the Jet Propulsion Laboratory, California Institute of Technology under Contract No. 1216777. The authors would like to thank Philip Shirvani for helpful comments.

REFERENCES

[1] M. Hiller, "Executable Assertions for Detecting Data Errors in Embedded Control Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2000),* pp. 24-33, June 2000.

[2] G. Bauer and H. Kopetz, "Transparent Redundancy in the Time-Triggered Architecture," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2000),* pp. 5-8, June 2000.

[3] P. Shirvani, "Fault Tolerant Computing for Radiation Environment," PhD thesis, Stanford Univ., pp. v., June 2001, (http://www-crc.stanford.edu).

[4] D.L. Hamilton, "Fault Tolerant Algorithms and Architectures for Robotics," *Proc. Seventh Electrotechnical Conf.,* vol. 3, pp. 1034-1036, 1994.

[5] N. Oh, P. Shirvani, and E. McCluskey, "Control Flow Checking by Software Signatures," to appear in *IEEE Trans. Reliability,* Dec. 2001, (http://www-crc.stanford.edu).

[6] A. Mahmood and E.J. McCluskey, "Watchdog Processor: Error Coverage and Overhead," *Proc. 15th Ann. Int'l Symp. Fault-Tolerant Computing (FTCS-15),* pp. 214-219, June 1985.

[7] A. Ersoz, D.M. Andrews, and E.J. McCluskey, "The Watchdog Task: Concurrent Error Detection Using Assertions," Technical Report TR 85-8, Stanford Univ., Center for Reliable Computing, 1985.

[8] G. Miremadi, J. Karlsson, J.U. Gunneflo, and J. Torin, "Two Software Techniques for On-Line Error Detection," *Proc. 22nd Ann. Int'l Symo. Fault-Tolerant Computing,* pp. 328-335, July 1992.

[9] G. Miremadi, J.TJ. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow checking," *Proc. Int'l Working Conf. Dependable Computing for Critical Applications, Springer-Verlag Series for Dependable Computing Systems,* Sept. 1995.

[10] J.G. Holm and P. Banerjee, "Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions," *Proc. Int'l Conf. Parallel Processing,* pp. 102-195, 1992.

[11] N. Oh, P. Shirvani, and E. McCluskey, "EDDI: Error Detection by Duplicated Instructions in Superscalar Processors," to appear in *IEEE Trans. Reliability,* Dec. 2001, (http://www-crc.stanford.edu).

[12] K. Parker and E.J. McCluskey, "Probabilistic Treatment of General Combinational Networks," *IEEE Trans. Computers,* vol. 24, no. 6, pp. 668-670, June 1975.

[13] S. Mitra, N. Saxena, and E.J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Int'l Test Conf.,* pp. 662-671, 1999.

[14] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution," *Proc. Int'l Computer Software and Application Conf.,* pp. 145-155, 1977.

[15] R.K. Scott, J.W. Gault, and D.F. McAllister, "The Consensus Recovery Block," *Proc. Total Systems Reliability Symp.,* pp. 74-85, Dec. 1983.

[16] J.C. Laprie et al., "Definition and Analysis of Hardware and Software Fault Tolerant Architectures," *IEEE Computer,* vol. 23, no. 7, pp. 39-51, July 1990.

[17] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer,* pp. 67-80, Aug. 1984.

[18] J.H. Lala and R.E. Harper, "Architectural Principles for Safety-Critical Real-Time Applications," *Proc. IEEE,* vol. 82, no. 1, pp. 25-40, Jan. 1994.

[19] M.R. Lyu and A. Avizienis, "Assuring Design Diversity in N-Version Software: A Design Paradigm for N-Version Programming," *Proc. Int'l Working Conf. Dependable Computing for Critical Applications,* pp. 197-218, 1991.

[20] L. Chen and A. Avizienis, "N Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," *Proc. Eighth Int'l Symp. Fault-Tolerant Computing,* pp. 3-9, 1978.

[21] R.K. Scott, J.W. Gault, and D.F. McAllister, "Fault-Tolerant Reliability Modeling," *IEEE Trans. Software Eng.,* vol. 13, no. 5, pp. 582-592, May 1987.

[22] J.B. Dugan and R.V. Buren, "Reliability Evaluation of Fly-by-Wire Computer Systems," *J. Systems and Software,* vol. 25, no. 1, pp. 109-120, Apr. 1994.

[23] P.E. Ammann and J.C. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Trans. Computers,* vol. 37, no. 4, pp. 418-425, Apr. 1988.

[24] J. Christmansson, A. Kalbarczyk, and J. Torin, "Dependable Flight Control System Using Data Diversity with Error Recovery," *Computer Systems Science and Eng.,* vol. 9, no. 2, pp. 142-150, Apr. 1994.

[25] D.T. Brown, "Error Detecting and Correcting Binary Codes for Arithmetic Operations," *IRE Trans. Electronic Computers,* vol. 9, pp. 333-337, Sept. 1960.

[26] J.H. Patel and L.Y. Fung, "Concurrent Error Detection in ALU's by Recomputing with Shifted Operands," *IEEE Trans. Computers,* vol. 31, no. 7, pp. 589-595, July 1982.

[27] P. Forin, "Vital Coded Microprocessor Principles and Application for Various Transit Systems," *Proc. IFAC Conf. Control, Computers, Comms. in Transportation, (CCCT '89),* pp. 137-142, 1989.

[28] P. Chapront, "Vital Coded Processor and Safety Related Software Design," *Proc. Int'l Federation of Automatic Control, SAFECOMP '92,* pp. 141-145, 1992.

[29] H. Engel, "Data Flow Transformations to Detect Results which are Corrupted by Hardware Faults," *Proc. IEEE High-Assurance Systems Eng. Workshop,* pp. 279-285, 1997.

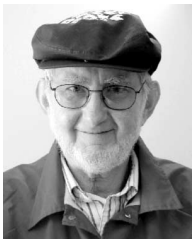
[30] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis for the Analysis of Multi-Version Software Subject to Coincident Errors," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1511-1517, 1985.

[31] B. Littlewood and D.R. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software," *IEEE Trans. Software Eng.,* vol. 15, no. 12, pp. 1596-1614, Dec. 1989.

[32] N.H. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Addison Wesley, 1992.
[33] J.B. Kuo, K.W. Su, and J.H. Lou, "A BiCMOS Dynamic Multiplier Using Wallace Tree Reduction Architecture and 1.5-V Full-Swing BiCMOS Dynamic Logic Circuits," *IEEE J. Solid-State Circuits,* vol. 30, no. 8, Aug. 1995.
[34] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* Second ed. San Francisco, Calif.: Morgan Kaufman Publishers, 1996.
[35] http://www.netlib.org/benchweb, 2001
[36] W. Harrison, "Compiler Analysis of the Value Ranges for Variables," *IEEE Trans. Software Eng.,* vol. 3, no. 3, pp. 243-250, May 1977.
[37] J. Patterson, "Accurate Static Branch Prediction by Value Range Propagation," *Proc. SIG-PLAN Conf. Programming Language Design and Implementation,* vol. 37, pp. 67-78, June 1995.
[38] M. Stephenson, J. Babb, and S. Amarashinghe, "Bitwidth Analysis with Application to Silicon Compilation," *SIGPLAN Notices,* vol. 35, no. 5, pp. 108-120, May 2000.
[39] W. Stallings, *Computer Organization and Architecture,* Fourth ed. Prentice-Hall, 1996.
[40] N. Oh, "Software Implemented Hardware Fault Tolerance," PhD thesis, Stanford Univ., Stanford, Calif., 2000.
[41] J.J. Shedletsky, "Error Correction by Alternate-Data Retry," *IEEE Trans. Computers,* vol. 27, no. 2, pp. 106-112, Feb. 1978.

Nahmsuk Oh received the BS degree in electronics engineering from Yonsei University, Seoul, Korea, in 1996. He received the MS degree in electrical engineering in 1998, the PhD degree in electrical engineering and PhD minor in computer science in 2001, all from Stanford University, Stanford. He is a research associate at the Center for Reliable Computing, Stanford University, and his research interests include fault-tolerant computing, digital testing, computer architecture, and logic synthesis. He is a member of both the IEEE and the IEEE Computer Society.

Subhasish Mitra received the BE degree in computer science and engineering from Jadavpur University, Calcutta, India, in 1994, the MTech degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1996, and the PhD degree in electrical engineering from Stanford University in 2000. He is a research associate at Stanford University's Center for Reliable Computing. His research interests include digital testing, logic synthesis, and fault-tolerant computing. He is a member of the IEEE.



Edward J. McCluskey is a professor of electrical engineering and computer science, as well as director of the Center for Reliable Computing. He developed the first algorithm for designing combinational circuits—the Quine-McCluskey logic minimization procedure—as a doctoral student at the Massachusetts Institute of Technology, Cambridge. At Bell Labs and Princeton, he developed the modern theory of transients (hazards) in logic networks and formulated the concept of operating modes of sequential circuits. His Stanford research focuses on logic testing, synthesis, design for testability, and fault-tolerant computing. McCluskey and his students at the Center for Reliable Computing worked out many key ideas for fault equivalence, probabilistic modeling of logic networks, pseudoexhaustive testing, and watchdog processors. He collaborated with Signetics researchers in developing one of the first practical multivalued logic implementations and then worked out a design technique for such circuitry. He served as the first president of the IEEE Computer Society. He is the recipient of many awards, including the IEEE Emanuel R. Piore Award. He is a fellow of the IEEE, AAAS, and ACM, and a member of the National Academy of Engineering. He has published several books, including two widely used texts.