Hindawi Publishing Corporation
International Journal of Aerospace Engineering
Volume 2016, Article ID 8213638, 10 pages
http://dx.doi.org/10.1155/2016/8213638

Research Article

Detecting Silent Data Corruptions in Aerospace-Based Computing Using Program Invariants

Junchi Ma,[1,2] Dengyun Yu,[3] Yun Wang,[1,2] Zhenbo Cai,[3] Qingxiang Zhang,[3] and Cheng Hu[1,2]

[1]*School of Computer Science & Engineering, Southeast University, Nanjing 211189, China*
[2]*Key Laboratory of Computer Network and Information Integration, Ministry of Education, Nanjing 211189, China*
[3]*Beijing Institute of Spacecraft System Engineering, Beijing 100094, China*

Correspondence should be addressed to Junchi Ma; bjbzmjc@126.com

Received 20 April 2016; Revised 20 September 2016; Accepted 10 October 2016

Academic Editor: Christopher J. Damaren

Soft error caused by single event upset has been a severe challenge to aerospace-based computing. Silent data corruption (SDC) is one of the results incurred by soft error. SDC occurs when a program generates erroneous output with no indications. SDC is the most insidious type of results and very difficult to detect. To address this problem, we design and implement an invariant-based system called Radish. Invariants describe certain properties of a program; for example, the value of a variable equals a constant. Radish first extracts invariants at key program points and converts invariants into assertions. It then hardens the program by inserting the assertions into the source code. When a soft error occurs, assertions will be found to be false at run time and warn the users of soft error. To increase the coverage of SDC, we further propose an extension of Radish, named Radish_D, which applies software-based instruction duplication mechanism to protect the uncovered code sections. Experiments using architectural fault injections show that Radish achieves high SDC coverage with very low overhead. Furthermore, Radish_D provides higher SDC coverage than that of either Radish or pure instruction duplication.

1. Introduction

A single event upset (SEU) is a change of state caused by one single ionizing particle (ions, electrons, photons, etc.) striking a sensitive node in a microelectronic device [1, 2]. The error in device output or operation caused as a result of SEU is called soft error. Because this type of error does not reflect a permanent failure, it is termed soft [3]. The first reports of failures attributed to cosmic rays emerged in 1975 when space-borne electronics malfunctioned [4]. In 1993, neutron-induced soft errors were even observed in airborne computers at commercial aircraft flight altitudes [5]. Soft error has emerged as a key challenge in aerospace-based computing [6, 7].

The raw error rate per device (e.g., latch, SRAM cell) in a bulk CMOS process is projected to remain roughly constant or decrease slightly; thus soft error rate per processor will grow with Moore's law in direct proportion to the number of devices added to a processor in the next generation [8]. Unless we develop and apply more effective soft error mitigation techniques, the trend is inevitable.

The result of soft error is categorized into four types [9], benign, crash, hang, and silent data corruption (SDC), shown in Figure 1. Benign means the error is masked and the program gets the right output; crash means the error causes the program to stop execution; hang means that resource is exhausted but the program still cannot finish execution; silent data corruption means that the program generates erroneous output. When crash or hang occurs, the system is aware that the program is executed abnormally. Compared with the others, SDC is more insidious since it occurs without any indications. Applying the erroneous output incurred by SDC may lead to loss of properties and even casualties. Erroneous output is definitely more dangerous than none, since users cannot be aware of errors until a serious consequence occurs. This paper mainly focuses on eliminating SDC.

Symptom-based fault detection mechanisms provide low-cost solutions [10, 11]. These mechanisms treat anomalous software behavior as symptoms of hardware faults and detect them by placing very low-cost symptom monitors in hardware or software. However, faults incurring SDC escape

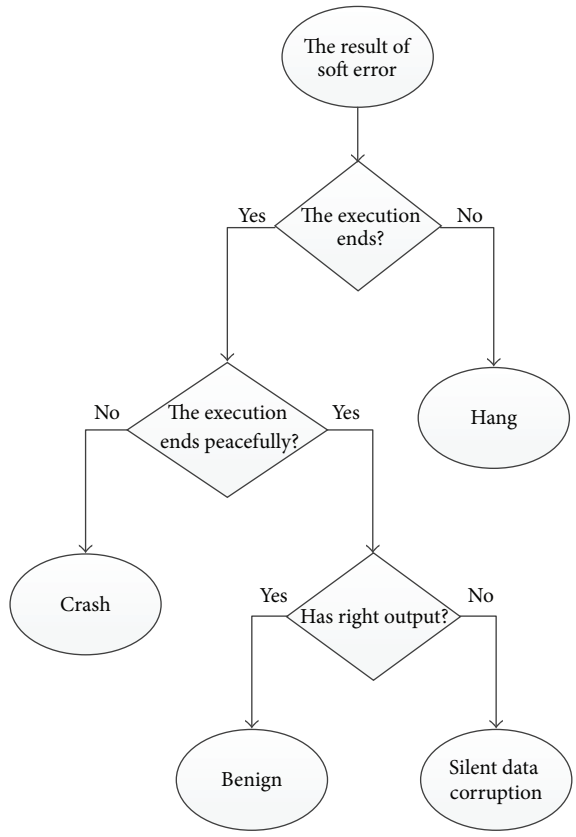

Figure 1: Classification of the result of soft error.

detection since they do not cause symptoms at all. To address this limitation, software-based instruction duplication is a possible alternative. With this approach, instructions are duplicated and their results are validated within a single thread of execution [12–15]. This solution has the advantage of being purely software-based, requiring no specialized hardware, and can achieve high coverage. However, the overheads in terms of performance and power are quite high since a large fraction of the program is replicated. Future missions will require much greater computational power than is available in today's processors [4]; thus low-cost fault detection solution is desired by future aerospace-based computing.

To address the problem of detecting SDC, this paper proposes an assertion-based detection mechanism. An assertion is a statement with a predicate (boolean-valued function, a true-false expression). If an assertion is found to be false at run time, an assertion failure rises, which typically causes the program to throw an assertion exception. Assertions in this paper are based on program invariants [16], which are properties that are true at a particular program point or points. For example, $x = 2y$ is an invariant about the variables x and y, which represents that they satisfy a linear relationship. This invariant is satisfied whenever the program is executed normally but seldom satisfied if a soft error affects the value of x or y. Based on this principle, we design and implement the system Radish which can harden the program against soft errors. Radish can extract invariants from a C program and insert invariant-based assertions back into the source code. Once an assertion is found to be false, it suggests that a soft error is detected. Then the execution is stopped and a warning is given.

Radish merely adds a few lines of code to original source code and thus it is easy to implement. Besides, it does not need to modify the underlying hardware and hardly increases the complexity of the system. Furthermore, the overhead of Radish turns out to be very low since the overhead of a single assertion is low and the number of assertions in a program is small.

To further increase the SDC coverage, we extend Radish by incorporating the mechanism of software-based instruction duplication. The code sections that are not covered by Radish are protected by deploying instruction duplication. Experimental results show that Radish achieves high coverage with low cost, and Radish_D even achieves higher coverage than that of Radish or pure instruction duplication. The techniques of Radish and Radish_D offer new solutions to soft error mitigation.

2. Definitions and Models

This section describes important definitions and models used in this paper.

Definition 1. A program is defined as $\langle F, E, \text{IN}, \text{OUT}\rangle$. F represents the functions in the program. E is the set of edges

Table 1: Relationships of invariants considered in this paper.

Category	Expression
Unary	$x = a;\ x > a;\ x < a;\ x\%a = 0;\ x \neq 0;$
	$x \in \{a, b, c\};\ x[k] < a;\ x[k] > a$
Binary	$y = ax + b;\ x < y;\ x \neq y;\ x = y^2;$
	$x[k] < x[k+1];\ x[k] > x[k+1];\ x[\,] \subset y[\,];$
	$x[k] < y[k];\ y[k] = ax[k] + b;\ x \in y[\,]$
Ternary	$z = ax + by + c;\ x = y \wedge z;\ x = y \vee z;$
	$x = \text{Lshift}(y, z);\ x = \text{Rshift}(y, z);\ x = \max(y, z);$
	$x = \min(y, z);\ x = y \times z;\ x = y \div z$

that denote dependencies between functions, s.t. $E = \{e_{xy} \mid f_x \text{ call } f_y,\ f_x \in F,\ f_y \in F\}$. IN and OUT denote the input and the output. Soft computation [17] is not considered in this paper; therefore, if F, E, and IN are determined, OUT can be uniquely determined.

Definition 2. A function F is composed of a set of basic blocks B and variables V; thus $F = \{B, V\}$. A basic block is a single entrance, single exit sequence of instructions. For a single instruction i_j, $i_j = \langle \theta, S, D \rangle$, where j denotes the sequence number of the dynamic instruction during the execution. θ denotes the program point, which equals the offset from the start position of the assembly file. S and D denote the source operands and the destination operands.

Definition 3. $\forall i_m \in f_y$, if $\exists i_k \in f_x, e_{yx} \in E \wedge i_k \cdot S = i_m \cdot D$, also $\nexists i_l \in f_x, l < k \wedge i_l \cdot D = i_k \cdot S$, then $i_{\max\{m\}}$ in f_y is defined as the connector instruction. Literally, the connector instruction transmits data from one function to another. The variable that a connector instruction writes is defined as the connector variable $\text{CV} = \{v \mid v \in i_{\max\{m\}} \cdot D\}$. Connector variables include function argument variables, function return variables, and global variables. By definition, the connector instruction is the last to write a connector variable in the function.

Definition 4. Execution profile is denoted by Γ, which is given as a tuple $\Gamma = \langle \theta, V, L \rangle$. Execution profile defines the values of the variables at given program points. θ represents the given program point. L is the acquired value set of variables that appears in V.

Definition 5. Invariant Q is defined as $Q = \langle \theta, \psi, r \rangle$, where θ represents the program point, $\psi = \langle v_1, \ldots, v_j, \ldots, v_n \rangle$ is the ordered set of variables, and r represents the relationship of variables that appear in ψ. $r \in R$, where R is the relationship set considered in the paper, shown in Table 1. R can be categorized into unary, binary, and ternary.

For instance, suppose an invariant $q_1 = \langle \theta_1, \psi_1, r_1 \rangle$, where $\theta_1 = 0\text{x}10$, $\psi_1 = \langle \text{tmp}_1, \text{tmp}_2 \rangle$, and $r_1 = \{\langle x, y \rangle \mid y = x + 1\}$. q_1 represents that at the program point 0x10 the ordered set $\langle \text{tmp}_1, \text{tmp}_2 \rangle \in r_1$, that is, tmp_1, tmp_2, satisfies the condition of $\text{tmp}_2 = \text{tmp}_1 + 1$.

The fault model we assume is a single bit flip within the register file. Most faults in other portions of the processor eventually manifest as corrupted state in the register file [18]. Moreover, we assume that at most one fault occurs during a program's execution.

3. Radish

This paper implements Radish, a system which can harden program against soft error. Radish enhances the resilience of the program to soft error by inserting assertions to the source code. The assertions are based on program invariants. If the statement of an assertion is not satisfied during the execution, the execution is stopped and a warning reports the occurrence of soft error.

The input of Radish is C source file and the output is a new C source file. The new source file can be compiled and executed just as the original source file. They are identical in functionality but vary in reliability.

This section introduces the workflow of Radish, which can be divided into three phases, that is, preprocessing, detecting, and selecting. Figure 2 shows the details of each phase. In the preprocessing phase, we extract the execution profiles Γ of the critical program points. Then Γ is used to extract potential invariants in the detecting phase. After that, invariants Q_{pot} are obtained and a fraction of them are converted to assertions in the selecting phase. In the end, hardened source code is outputted. We will describe each phase below.

3.1. Preprocessing Phase. In the phase of preprocessing, we find the critical program points and extract their execution profiles. The profiles are used to extract invariants in the next phase. Finally, assertions will be placed in those program points to prevent faults from propagating. The SDC coverages vary due to the program points of assertions, and therefore we analyze the propagation of SDC and find the critical program points for propagation. A fault may propagate through data flow or control flow to incur SDC. Due to the distinction of the two categories of propagation, we analyze and search for their critical program points separately.

When a fault propagates through data flow, the same static instructions are executed just as the fault-free execution, but the data that the instructions read or write are corrupted. To incur SDC, the corrupted data need to be transmitted to other functions, especially the output function. Only connector instructions can perform this operation; thus they must be executed and the data they transmit are corrupted. This makes the connector instructions efficient for fault detection and therefore they are selected as the critical program points against data flow propagation.

Next we discuss fault propagation in control flow. The compare instruction performs a comparison between two values and the result of the comparison impacts the bits of the flag register, which determines the consequent jump performed by a branch instruction. Propagation through control flow means that an erroneous jump is performed by a branch instruction. Assume that in the fault-free execution i_k is a branch instruction and the next instruction is $i_{k+1} \in b_u$, which

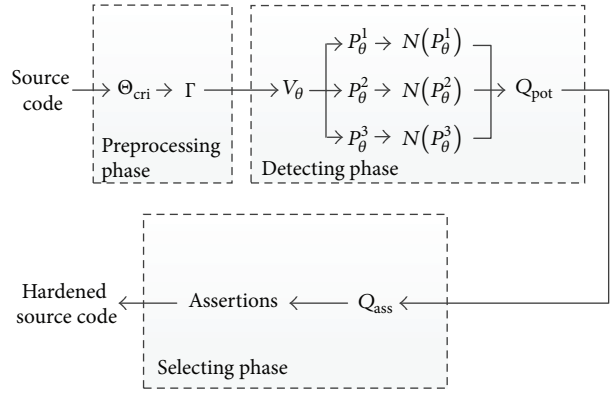

Figure 2: The workflow of Radish.

means i_k chooses b_u as the next basic block. When the flag register is corrupted in the presence of soft error, then $i_{k+1} \in b_w$, which means i_k chooses the erroneous branch b_w instead of b_u. To avoid this, we should check if the right branch is taken after the execution of i_k. Therefore branch instructions are selected as the critical program points of control flow propagation.

According to the analysis above, the critical program points of data flow and control flow propagation refer to connector instructions and branch instructions. It takes two steps to extract the execution profiles of the critical program points.

Step 1. We compile the source code and translate it into assembly file and then locate connector instructions and branch instructions in the assembly file. Their program points are recorded and added to the program point set Θ_{cri}.

Step 2. The execution profile is acquired by using Kvasir [16]. Kvasir executes C and C++ programs and creates data trace files of variables and their values by examining the operation of the binary at runtime. Using Kvasir makes it possible to interrupt program's execution and read the values of all variables manifest at the program points of interest. Once it finishes executing, we get the profiles Γ at the target program points in Θ_{cri}.

3.2. Detecting Phase. In the detecting phase, the ordered set of variables and the corresponding ordered set of values are generated according to the execution profiles Γ. We check if the values satisfy any relationship of R listed in Table 1. The detecting phase has 4 steps in total.

Step 1. For each program point θ of Θ_{cri}, we get the set of accessible variables V_θ from the execution profile Γ. Then the unary, binary, and ternary ordered sets P_θ^1, P_θ^2, and P_θ^3 are created. The superscript digits refer to the number of variables of the ordered set. For example, P_θ^2 is an arrangement of two variables in V_θ; that is, $P_\theta^2(v_k, v_j) = \{\langle v_k, v_j\rangle \mid v_k \in V_\theta \wedge v_j \in V_\theta\}$.

Step 2. Find the corresponding values of the variables appearing in P_θ^1, P_θ^2, P_θ^3 and generate the ordered sets of value $N(P_\theta^1)$, $N(P_\theta^2)$, $N(P_\theta^3)$. For example, $N(P_\theta^2(v_k, v_j))$ is the ordered value set of $P_\theta^2(v_k, v_j)$. $N(P_\theta^2(v_k, v_j)) = \{\langle l_u, l_w\rangle \mid \langle\theta, v_k, l_u\rangle \in \Gamma \wedge \langle\theta, v_j, l_w\rangle \in \Gamma\}$.

Step 3. For the relationships that have undetermined parameters, we use a part of the ordered set of values to calculate those parameters. Thus the entire expression is determined.

Step 4. Test if each element of the ordered set of values satisfies the condition of the relationship. If so, then create a new invariant and put it into the potential invariant set Q_{pot}.

Take a binary relationship $r_{lin} = \{\langle x, y\rangle \mid y = ax + b\}$ as example. We shall show each step of detecting phase. Since it is a binary relationship, only the binary ordered sets of P_θ^2 and $N(P_\theta^2)$ are considered in this example.

In the first step, we get $V_\theta = \{v_k \mid \exists l, \langle\theta, v_k, l\rangle \in \Gamma\}$ by searching the execution profile Γ. Then $P_\theta^2(v_k, v_j) = \{\langle v_k, v_j\rangle \mid v_k \in V_\theta \wedge v_j \in V_\theta\}$ is obtained by creating the arrangement of every two variables in V_θ.

In the second step, $N(P_\theta^2(v_k, v_j)) = \{\langle l_u, l_w\rangle \mid \langle\theta, v_k, l_u\rangle \in \Gamma \wedge \langle\theta, v_j, l_w\rangle \in \Gamma\}$ is obtained by finding the values of v_k and v_j in the execution profile Γ. There may be many value pairs of v_k and v_j because certain code sections can be invoked for many times in a single execution and each invoking produces one value instance.

In the third step, we calculate the parameters a, b in r_{lin}. To this end, we need to use at least 2 elements of $N(P_\theta^2(v_k, v_j))$. Assuming the two elements are $\langle l_1, l_2\rangle$ and $\langle l_3, l_4\rangle$, it could be easily obtained that $a = (l_4 - l_2)/(l_3 - l_1)$ and $b = (l_2 l_3 - l_1 l_4)/(l_3 - l_1)$.

In the last step, all elements in $N(P_\theta^2(v_k, v_j))$ are checked whether they satisfy $r_{lin} = \{\langle x, y\rangle \mid y = ((l_4 - l_2)/(l_3 - l_1))x + (l_2 l_3 - l_1 l_4)/(l_3 - l_1)\}$. If all of them pass this validation, the invariant $\langle\theta, \langle v_k, v_j\rangle, r_{lin}\rangle$ holds and it is added to the potential invariant set Q_{pot}.

3.3. Selecting Phase. It is often observed that the number of elements of the potential invariant set Q_{pot} is very large. If all of them are converted into assertions and inserted into source file, it will incur very high performance overhead. In the selecting phase, proper invariants are selected according to their capability of detecting SDC. Heuristics about selection

criteria are formulated on the basis of propagation of SDC. These heuristics are generic and can be applied to any invariants. We list the heuristics first and then describe the selecting steps.

Heuristic 1. There are certain types of variables that should be monitored at each target program point.

A fraction of variables are capable of telling if the execution is going well, and thus monitoring these variables is able to detect SDC. The target program point set Θ_{cri} can be categorized into program points of connector instructions and branch instructions. At the program points of connector instructions, it is the connector variables that should be paid special attentions to since they reflect whether results of functions are correct. At the program points of branch instructions, branch-controlling variables, which appear in the statement of *if*, *while*, or *for* structure, reflect the status of these structures and thus should be noticed. Therefore for all target program points we find certain variables to monitor.

Heuristic 2. The likelihood of detecting SDC increases if the number of valid values defined by an assertion decreases.

Invalid values cannot pass the examination of assertions in the presence of soft error. Therefore having more invalid values (less valid values) means the likelihood of detecting SDC increases. The number of valid values of an invariant is determined by its relationship. Equality relationship, using "=" as operator, only has one valid value, then come inclusion ($\in$, $\subseteq$), range ($>$, $<$), and inequality relationship ($\neq$) in order of ascending number of valid values.

Heuristic 3. The likelihood of detecting SDC increases if more variables are included by an assertion.

The more the variables appearing in an assertion, the more the variables it can monitor. If any of the variables gets corrupted due to soft error, the assertion will be able to catch the error. Thus having more variables in an assertion leads to higher coverage of SDC. So far, the largest number of variables is 3, which refers to ternary relationships.

Utilizing these heuristics, we are able to reduce the number of invariants and obtain more effective assertions. The selecting phase has three steps.

Step 1. The invariants which contain connector variables at the program points of connector instructions or branch-controlling variables at the program points of branch instructions are selected based on Heuristic 1.

Step 2. The invariants with the relationship that has fewer valid values are picked up according to Heuristic 2.

Step 3. The invariants which contain the largest number of variables are selected due to Heuristic 3.

The selecting process stops until there is only one invariant left or all the steps have been performed. Then we convert the chosen invariants into assertions, which is basically a string conversion problem. For brevity's sake, we do not talk about it in this paper. Finally we include the assertion header file at the beginning of the new source file to make sure assertions can work.

4. Radish_D

The assertions generated by Radish cannot fully monitor all the variables and program points; thus certain faults might propagate through unprotected code sections. To further increase the coverage of SDC, we introduce software-based instruction duplication mechanism to protect the code sections that are not covered by Radish.

This paper utilizes instruction duplication mechanism of SWIFT [15] for comparison and also for our own duplication in Radish_D. SWIFT duplicates all computation instructions along the path of replication and the replica instructions use different registers and different memory locations. At certain synchronization points, comparison instructions are inserted to check if the original instructions and their replica have identical values.

Rather than deploying full instruction duplication mechanism of SWIFT, Radish_D applies selective instruction duplication mechanism. Because a portion of instructions have been protected by assertions, we only need to duplicate the others.

Before deploying duplications, we need to determine which variables are safe under the protection of assertions. An assertion is capable of protecting the variables which appear in its statement. However, the protection does not last for the entire lifetime of those variables. Only the fraction from the beginning of the local function till the variable's host assertion is considered safe, since the variable's value is checked during the execution of the assertion.

We partition each variable's lifetime by assertions and identify the safe periods. Then duplications are deployed in the instruction level. The targets of duplications are the instructions which do not contain a variable in the safe period as operand. A replica instruction is created by copying the opcode and operands of the original instruction. The destination operand is changed into an unused register, the copy of the original destination operand. Next we decide if there is a need to change the source operands of the replica instruction. If there has already been a copy of the source operand, which means this source operand was some instruction's destination operand and thus got a copy, we replace the replica instruction's source operand with its copy. The replica instruction is inserted before the original instruction in the same basic block.

Besides, store, branch, and call instructions are chosen as the synchronization points. If any source operand of these instructions has a copy, we compare its value with that of its copy by inserting a compare instruction. According to the type of the operand (int or float), the compare instruction can be either icmp instruction or fcmp instruction. After the compare instruction, a branch instruction using the predicate of neq is inserted into the code. If the two values show a discrepancy, it will jump to a function called faultDetected; if otherwise, it will continue to execute the previous store, branch, or call instruction. The function of faultDetected outputs error messages and returns with an exit code, which will inform the system of soft error and end the execution.

We use an example to show the distinction between our method and full instruction duplication mechanism in

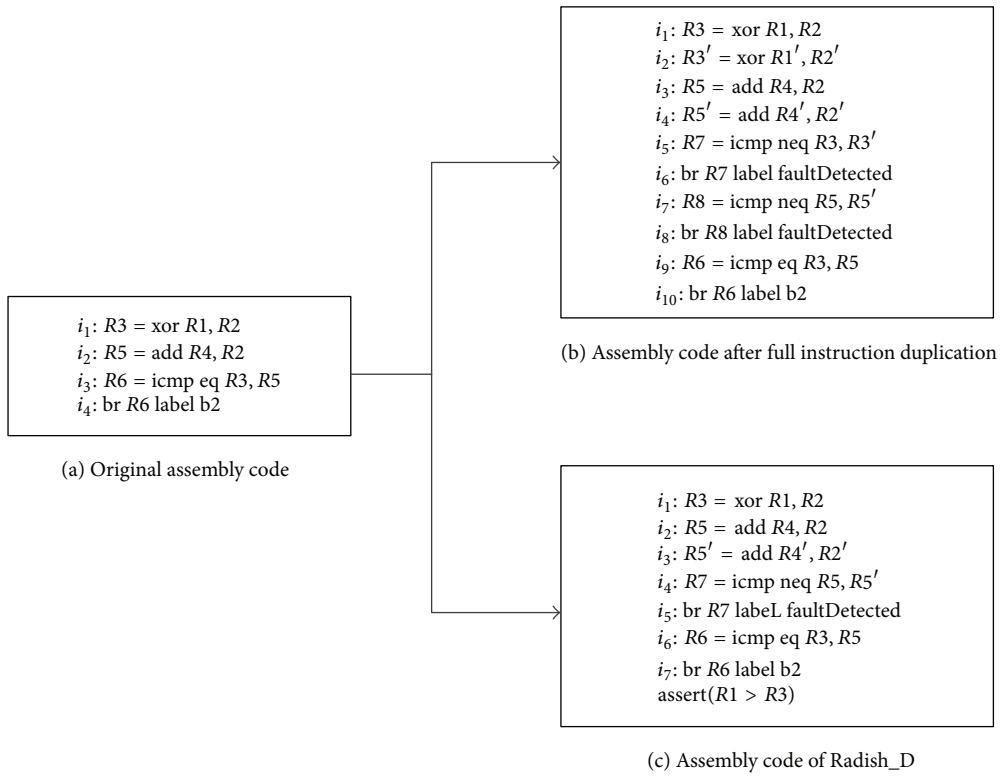

Figure 3: A sample assembly code before and after transformation of full instruction duplication and Radish_D.

Figure 3. For consistency, we make use of the LLVM [19] assembly language to present the assembly code. Figure 3(a) shows the original assembly code and Figure 3(b) shows the assembly code after full instruction duplication. It can be found in Figure 3(b) that $R3'$ is the replica of $R3$ and the duplication is accomplished by the instruction i_2. Similarly, $R5'$ is the replica of $R5$ through the duplication by i_4. i_9 is the synchronization point and the source operands of i_9, $R3$ and $R5$, need to be examined. i_5 and i_7 compare $R3$ and $R5$ with their replicas separately. If the values of $R3$ and $R3'$ are not equal, i_6 will call faultDetected to report a soft error.

The assembly code generated by Radish_D is shown in Figure 3(c). Assume that we have already obtained an assertion about $R1$ and $R3$ by utilizing Radish, which is shown in the line of code "assert($R1 > R3$)." Due to the assertion, $R1$ and $R3$ are considered safe during the execution of this example. $R1'$ and $R3'$ are no longer necessary and the instructions used for their duplication are eliminated. Variables except $R1$ and $R3$ still need to be duplicated and checked; thus $R5$ is duplicated by i_3 and checked at the synchronization point i_6. The efficiency of Radish_D and full instruction duplication mechanism will be exploited in the next section.

5. Experiment

This paper applies fault injection experiments to validate the effectiveness of Radish and Radish_D. The fault injection experiment is performed on the original executive first. The hardened executives using Radish, Radish_D, and full instruction duplication are targeted subsequently. We compare the results of the fault injection experiments and calculate the SDC coverage and performance overhead. To ensure a fair comparison among these mechanisms, we use a metric called the SDC detection efficiency, which is defined in prior work [9] as the ratio between SDC coverage and overhead for a detection mechanism.

The platform for validation is Ubuntu 14.04 (AMD64 architecture). LLFI [20] is applied to perform fault injections. LLFI is an LLVM-based fault injection tool. The source code is translated into an intermediate representation (IR) and the IR code is then injected. The faults can be injected into specific program points, and the effect can be easily tracked back to the source code. LLFI is configured to inject destination register. In a single fault injection, LLFI randomly picks up one instruction and injects 1 soft error to the destination operand. One fault injection experiment continues until the fault injection has been repeated for 1000 times. The injected faults may affect data flow or control flow. We take the following LLVM IR code to explain the effect on control flow:

(1) %judge1=icmp ne i32 %1, %2

(2) br i1 %judge1, label %BB1, label %BB2.

%judge1 determines the outcome of branch. If %judege1 is injected, the branch instruction may choose the wrong branch and thus affect control flow. The mechanism of full instruction duplication is implemented by developing a new pass under LLVM infrastructure. The pass is also used by

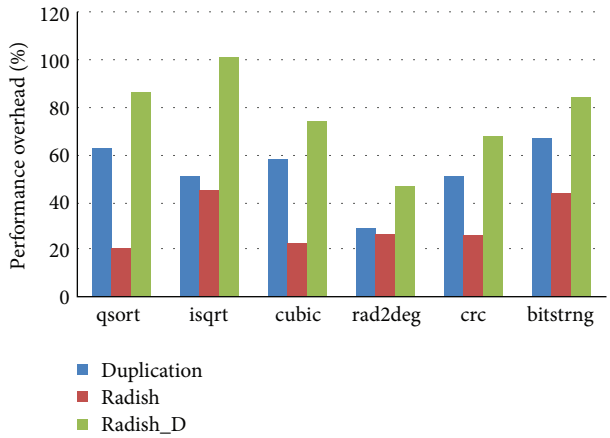

Figure 4: The comparison of performance overheads among full instruction duplication, Radish, and Radish_D.

Radish_D for the operation of instruction duplication by modifying certain conditions for duplication.

The programs used for evaluation are from MiBench benchmark suite [21]. These programs are qsort (which performs the algorithm of quick sort), isqrt (which is base two analogue of the square root algorithm), cubic (which solves a cubic polynomial), rad2deg (which converts between radians and degrees), crc (which computes 32-bit crc to detect accidental changes to raw data), and bitstrng (which prints bit pattern of bytes formatted to string). These are C programs consisting of a few hundred lines of C code. We use 25 inputs to extract invariants and randomly choose one input for the injection.

5.1. Comparison between Radish and Full Instruction Duplication. Figure 4 shows the performance overheads of Radish and full instruction duplication. We use the execution time of the original program as baseline for comparison. Compared with the baseline, the average overhead incurred by Radish is 30.4%, and the overhead incurred by full instruction duplication is 52.8%. The overhead of full instruction duplication mechanism is 22.4% higher than the overhead of Radish for the studied programs.

Figure 5 shows the SDC coverages. The average SDC coverage of Radish is 77.1% and that of full instruction duplication is 84.3%. The average SDC coverage of full instruction duplication is 7.2% higher than that of Radish. Among most of the benchmarks, the SDC coverages of full instruction duplication and Radish are very close.

Full instruction duplication does not achieve nearly 100% coverage since it does not check the result of store and branch instruction. For example, in Figure 3(b) which denotes the full instruction duplication, if $R6$ in i_9 is injected, i_{10} is affected and may choose the wrong branch. SWIFT [15] raises the coverage to nearly 100% since it assumes that the hardware applies ECC and it adds control flow checking mechanism. The SDC detection efficiency can be observed in Figure 6. Radish has higher SDC detection efficiency, which is 1.6 times as much as that of full instruction duplication. This is because the mechanism of full instruction duplication protects all instructions executed, which incurs high SDC coverage with very high overhead. However, Radish obtains relatively high SDC coverage with much lower overhead. Radish achieves this by curbing the number of program points that generate assertions. Further, the execution cost of assertions is relatively low and assertions have good SDC coverage since they are seldom satisfied when soft errors occur.

5.2. The Experimental Results of Radish_D. The average SDC coverage of Radish_D is 92.5%, which is 8.2% higher than that of full instruction duplication and 15.5% higher than that of Radish. It can be validated that instruction duplication of Radish_D protects unsafe code sections that are not covered by assertions. Radish_D may generate assertions that check the variable which is stored in the memory after the store instruction (see Heuristic 1). Moreover, at the program points of branch instructions, branch-controlling variables are checked. Therefore the assertions of Radish_D catch some of faults that escape the detection of duplication mechanism and the coverage of Radish_D is higher than that of full instruction duplication.

The average overhead of Radish_D is 76.3%, lower than the sum of the overhead of full instruction duplication and Radish, because we eliminate the duplication deployed to the instructions that have already been protected by assertions.

The average SDC detection efficiency of Radish_D is lower than that of full instruction duplication or Radish. For Radish_D, there are overlapping soft errors that can be detected by both instruction duplication and assertions. To these soft errors, the overhead increases by deploying instruction duplication but the SDC coverage does not increase. The SDC detection efficiency is the ratio between SDC coverage and overhead, and thus it is lowered.

5.3. False Positives of Invariants. A false positive for an input can occur when the values at the assertion points for this input do not satisfy the condition of the assertion learned from the training inputs. We use 25 inputs for training and 100 inputs for testing. No faults are injected in these runs. We test all the programs that were used to evaluate SDC coverage in the fault injection experiment. The result shows that the averaged false positive rate of the studied programs is 4.8%.

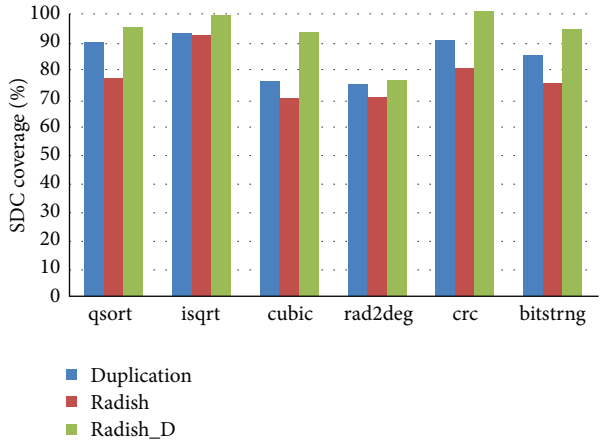

Figure 5: The comparison of SDC coverages among full instruction duplication, Radish, and Radish_D.

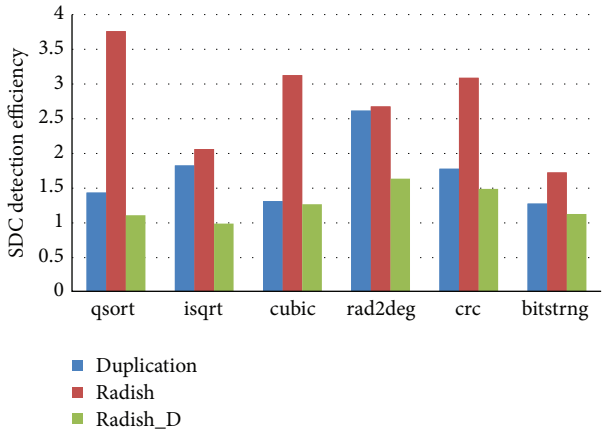

Figure 6: The comparison of SDC detection efficiencies among full instruction duplication, Radish, and Radish_D.

We also conduct the experiment to exam the effect of training set size. The result of qsort is shown in Figure 7. The training set consists of 25, 50, and 75 inputs and false positives are computed across 100 inputs.

The false positive rate decreases from 5% to 3% as the training set size is increased from 25 to 50 and to 2% for 75 inputs. The SDC coverage also decreases as the training set increases from 25 to 75 inputs. The impact on both SDC coverage and false positive rate from increasing the training set size is significant. Hence we should choose the training set size according to the user target. If user specifies the bound of SDC coverage and overhead by turning false positive rate into overhead we can choose a training set size to achieve the target.

Besides, reexecution can reduce the overhead incurred by fault positive. When an assertion raises an alarm, we can determine if it is a false positive by reexecuting it. If the assertion raises an alarm again, it is a false positive. In this case, the alarm can be ignored, and the program can continue.

From the discussion above, it can be concluded that Radish_D has higher SDC coverage than that of Radish or full instruction duplication. But its overhead is also higher, which suggests that Radish_D should be used in the situation where SDC coverage is considered to have more priority than overhead. Further, the SDC detection efficiency of Radish is far higher than that of Radish_D or full instruction duplication, which means it is more cost-effective. But Radish may incur overhead due to false positives. Users can choose Radish or Radish_D according to their consideration of tradeoff between the SDC coverage and performance overhead.

6. Related Work

Prior research [8, 22, 23] applies invariants with a single variable and most of the invariants are based on bounded range. We apply invariants with more variables, which can achieve better coverage in many occasions. For example, we can always extract an invariant $n - k + 1 = 0$ from a typical loop structure shown as follows:

```
for (k = 1; k <= n; k++)
{
    ...
}
← Extracting invariant from here
```

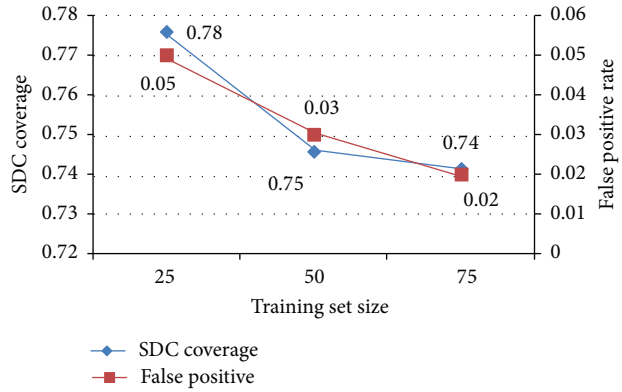

Figure 7: The SDC coverage and false positive rate for varied training set sizes.

It is found that assert($n - k + 1 = 0$) is often better than the bounded-range-based invariant assert($k_{\min} \leq k \leq k_{\max}$) at detecting errors since assert($n - k + 1 = 0$) checks both n and k while assert($k_{\min} \leq k \leq k_{\max}$) only checks k.

A typical criterion for selection of detectors defined in [22], the tightness, is the probability that the detector detects an error given that there is an error in the value of the variable that it checks. The notion of tightness is based on the value of a single variable. The invariant in this paper may include 2 or 3 variables and the notion of tightness cannot be used to describe an invariant with more than one variable. For example, if x is flipped in the invariant $x < y$, since there are multiple possible values of y, it cannot be decided whether the invariant is still satisfied and thus the tightness cannot be calculated. Since the tightness cannot be used, we apply certain heuristics to choose invariants and it is proved to be effective.

7. Conclusion

To address the problem of detecting SDC, we propose an approach which applies invariant-based assertions and implement a system called Radish. Radish neither requires any hardware modifications to add error detection capability to the original system, nor needs to acknowledge the semantics of the program and thus possesses a good scalability. Experiments show that Radish achieves high SDC coverage with very low overhead.

Furthermore, we propose Radish_D by adding instruction duplication to the unsafe code sections which are not covered by assertions. Radish_D achieves higher SDC coverage than that of Radish or full instruction duplication mechanism. Both Radish and Radish_D offer feasible alternatives for soft error mitigation.

Competing Interests

The authors declare no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the National Basic Research Program of China ("973" Project).

References

[1] H. Schirmeier, C. Borchert, and O. Spinczyk, "Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pp. 319–330, IEEE, Rio de Janeiro, Brazil, June 2015.

[2] A. O. Daniel, L. P. Laércio, S. Thiago et al., "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, 2016.

[3] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, pp. 243–247, San Francisco, Calif, USA, February 2005.

[4] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975.

[5] J. Olsen, P. E. Becher, P. B. Fynbo, P. Raaby, and J. Schultz, "Neutron-induced single event upsets in static RAMS observed at 10 km flight altitude," *IEEE Transactions on Nuclear Science*, vol. 40, no. 2, pp. 74–77, 1993.

[6] J. P. Walters, K. M. Zick, and M. French, "A practical characterization of a NASA SpaceCube application through fault emulation and laser testing," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pp. 1–8, June 2013.

[7] S. Mittal and J. S. Vetter, "A survey of techniques for modeling and improving reliability of computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1226–1238, 2016.

[8] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based fault screening," in *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 169–180, Scottsdale, Ariz, USA, February 2007.

[9] Q. Lu, K. Pattabiraman, M. S. Gupta et al., "SDCTune: a model for predicting the SDC proneness of an application for configurable protection," in *Proceedings of the Compilers, Architecture and Synthesis for Embedded Systems*, pp. 1–10, Uttar Pradesh, India, 2014.

[10] N. J. Wang and S. J. Patel, "ReStore: symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.

[11] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1, pp. 265–276, 2008.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[13] M. Shafique, S. Rehman, P. V. Aceituno, and J. Henkel, "Exploiting program-level masking and error propagation for constrained reliability optimization," in *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*, pp. 1–17, Austin, Tex, USA, June 2013.

[14] S. Rehman, M. Shafique, P. V. Aceituno, F. Kriebel, J.-J. Chen, and J. Henkel, "Leveraging variable function resilience for selective software reliability on unreliable hardware," in *Proceedings of the 16th Design, Automation and Test in Europe Conference and Exhibition (DATE '13)*, pp. 1759–1764, Grenoble, France, March 2013.

[15] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 243–254, IEEE Computer Society, San Jose, Calif, USA, 2005.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo et al., "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, 2007.

[17] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, pp. 1–12, IEEE Computer Society, June 2013.

[18] F. Shuguang, G. Shantanu, A. Amin et al., "Shoestring: probabilistic soft error reliability on the cheap," in *Proceedings of the ASPLOS*, pp. 385–396, Pittsburgh, Pa, USA, 2010.

[19] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*, pp. 75–86, San Jose, Calif, USA, March 2004.

[20] A. Thomas and K. Pattabiraman, "LLFI: an intermediate code level fault injector for soft computing applications," in *Proceedings of the Workshop on Silicon Errors in Logic System Effects (SELSE '13)*, pp. 1–8, Palo Alto, Calif, USA, 2013.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst et al., "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization*, pp. 3–14, Austin, Tex, USA, 2001.

[22] K. Pattabiraman, S. Giacinto, C. Daniel et al., "Dynamic derivation of application-specific error detectors and their implementation in hardware," in *Proceedings of the 6th European Dependable Computing Conference (EDCC '06)*, pp. 97–108, Coimbra, Portugal, October 2006.

[23] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou, "Using likely program invariants to detect hardware errors," in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pp. 70–79, IEEE Computer Society, Anchorage, Alaska, USA, June 2008.