CCDA: Correcting Control-flow and Data Errors Automatically

Navid Khoshavi, Mohammad Maghsoudloo, Hamid R. Zarandi
Department of Computer Engineering and Information Technology
Amirkabir University of Technology
E-mails: {navid.khoshavi, m.maghsoudloo, h_zarandi}@aut.ac.ir

Abstract—This paper presents an efficient software technique to detect and correct control-flow and data errors through addition of redundant codes in a given program. This technique is performed using both control-flow and data-flow graphs. Based on this method, most of control-flow errors in the program are, first detected, and next corrected automatically, so that not only errors in the control flow are detected / corrected but also, errors in the data of program because of control flow errors are repaired. In order to evaluate this technique, we use a simulator which can simulate an assembly code on the 80X86 microprocessors. We use three benchmarks: quick sort, matrix multiplication and linked list, and 1000 transient faults were injected on several executable points in the program code. The experimental results demonstrate that on average 93.5% of control flow errors are detected and corrected by CCDA automatically without any data error generation. It also shows that the performance and memory overheads of CCDA are noticeably less than previous related works.

I. INTRODUCTION

In applications where reliability and safety are of major concerns such as critical applications, occurring transient and permanent faults during run time can cause considerable corruption in systems. Cost in these applications is the second goal that can be considered after safety and reliability. In these cases, hardware-based redundancy methods can be applied, which is not feasible in such systems, instead in these systems software implemented hardware fault tolerance is more suitable.

A Control Flow Error (CFE) will occur when the processor executes an incorrect sequence of instructions. A CFE can be caused by transient or permanent faults in hardware components such as program counter (PC), address circuits, or the memory system [2]. On average between 33 to 77% of transient faults have been shown to result in the CFEs by previous works [1]. Control Flow Checking (CFC) is a regular method that is used for monitoring the flow of a program which partitions the program code into basic blocks and then adds redundant instructions for checking the flow of the program. Basic block is a branch free group of instructions terminated by a branch. The instructions in a basic block should be executed sequentially. CFEs are divided into two types (inter node and intra node). Intra node CFE is an illegal jump within a node (basic block), and inter node CFE is an illegal jump between two nodes, or between a node and the unused spaces of the program which is called partition block.

Several methods have been designed for detecting each type of the CFEs in a program code. From viewpoint of implementation forms, approaches divided into software-based and hardware-based. Often hardware-based methods use an external watchdog (checker) processor for state and performance monitoring of the main (master) processor [5]. On the other hand, software-based methods mostly use a signature for each basic block. That signature is calculated at runtime, and is compared with the one which was calculated at compile time, without using additional hardware components [5].

Almost all the software-based techniques such as BSSC [5], ECI [5], CFCSS [6] and CEDA [2] are handled only for CFE detection through signature assignments to basic blocks. CFEs detection has been studied widely in literature, but for CFEs correction, a few published works exist [1], [3]. After the CFE is detected, the control should be transferred back to the block, which illegal jump has occurred in. Consequently the CFE is detected and corrected. But correcting the CFE is not sufficient and we cannot be sure about the accuracy of final results, because of the data errors that are generated by the CFE. For more assurance, the generated data errors should be corrected in correction phase or after it.

For correcting the data errors, the program needs to be check-pointed at some locations in the code, and then reloaded and re-executed from the previous checkpoint, for obtaining correct results in output. But this may not be possible in some systems, because of high overhead and high latency for check-pointing, loading the program state and re-executing [1].

In this paper, we present a technique for automatic CFEs and data errors correction. In order to CFEs detection, we use signature for each node, and then specify some instructions and insert them at the beginning and at the end of the nodes for calculating and checking signatures at run time. For automatic correction of CFEs and data errors, a function is defined at design time. This function is
implemented through considering Data Flow Graph (DFG) of the program.

First to evaluate our technique, we have used three well-known benchmarks. These benchmarks are generated and simulated in EMU assembler and simulator [16] that can simulate and run assembly codes on the 80X86 microprocessors. The results of injecting 1000 transient faults on the program code reveal that 93.5% of control flow errors are detected and then corrected without any data error generation. Also the measurements show that performance and memory overheads of CCDA are considerably less than previous related works like ACCED [1] which is based on duplication methods to correct the data errors.

The structure of this paper is as follow: the next section gives related works proposed previously and introduces terminologies. Section 3 gives a brief overview of our technique. Section 4 describes the detection technique which is used in our work. Section 5 explains our correction approach. Section 6 describes the tool and presents the experimental results, and also contains a discussion about results. Finally Section 7 concludes the paper.

II. RELATED WORK

This section first, introduces software-based techniques for CFEs detection (BSSC, ECI and CEDA), then describes a technique for CFEs correction (ACCE).

The Block Signature Self Checking (BSSC) technique presented by [5] is purposed to check the flow of a program between branch free blocks (basic blocks) by defining a signature for each basic block and check it at the end of the block. In this technique, at the beginning of the blocks, a subroutine is called and address of the first instruction in the node is pushed into the top of the stack as the signature, or it is stored in a static variable. A subroutine at the end of the basic block compares the embedded signature with the signature stored by the entry routine. Therefore it can detect an illegal jump between two nodes caused by a CFE.

In the Error Capturing Instructions (ECI) technique [5], errors which cause erroneous execution in the unused spaces and data spaces are detected. This technique fills unused space (partition block) with branch instructions to an infinite loop or with software interrupt instructions. So with execution of these instructions, we can indicate that a CFE has occurred.

These two techniques (BSSC and ECI) can be combined in order to increase the error detection coverage, because the BSSC technique can only detect the CFEs in the program space and the ECI can only detect the CFEs in the unused and data spaces.

The Control-flow Error Detection through Assertions (CEDA) technique is presented by [2]. In this technique like previous techniques, extra instructions are automatically embedded into the program at compile time in order to update run-time signatures continuously, and to compare them with pre-assigned value. Previous techniques insert multiple instructions in the basic blocks whenever they want to update the run-time signature, but this technique inserts fewer instructions than the previous works, because of calculating signatures differently. Therefore this technique has less overhead and is more efficient than the prior works. Decreasing overheads and increasing effectiveness are main purposes of this approach.

The Automatic Correction of Control-flow Errors (ACCE) technique [1] partitions the program code into functions which include one or more basic blocks. ACCE uses CEDA technique for CFEs detection. After the detection phase, a predefined function called error-handler is automatically executed, and the program control is transferred to the function and then to the basic block in which the illegal jump has been occurred. An extension of ACCE, called Automatic Correction of Control-flow Errors with Duplication (ACCED), is used for data error detection and correction through duplicating computations. The ACCED takes advantage of the property of the ACCE that variables which may be wrong due to CFE correction are limited to a single node with high probability, and further increases the probability of a correct output.

III. OVERVIEW OF OUR TECHNIQUE

In this section we first describe some definitions and then give a brief overview of our technique.

Control Flow Graph (CFG): CFG is a graph with sets of nodes (basic blocks) and directed edges which show the right transmission between basic blocks.

Data Flow Graph (DFG): DFG is a graph with sets of nodes (operations and variables) and directed edges which present data dependencies between variables.

Control-Flow Error (CFE): A control flow error occurs if the sequence of instructions which are executed in the presence of a fault is different from the fault free sequence.

Source Signature Register (SSR): is a run-time register which is continuously updated, and finally stores the signature of the basic block in which a CFE has occurred.

Destination Signature Register (DSR): is a run-time register which is continuously updated, and finally stores the signature of the basic block that the control is transferred to it incorrectly due to a CFE.

Data errors in so many applications such as safety critical applications may cause destructive results in output. Therefore in these situations, only focusing on correction of CFEs is not sufficient and applicable. Fig. 1 illustrates how the data errors can be generated due to a CFE. When a CFE occurs (like Branch1 in Fig. 1) it can be detected at the beginning of the basic block or at the end of it by added instructions, then the function called CFE handler can transfer control to the block which the CFE has occurred in it (Branch3). But this type of correction can generate data errors.
With regards to Fig. 1, the data errors can happen because of executing the set of instructions resided in Region 1 (which are not executed at regular time), or the set of instructions resided in Region 2 (that are performed wrongly) or the set of instructions resided in Region 3 (which are executed additionally). So we cannot be sure that the data errors will not happen in the output, by returning exactly to the block which the CFE has occurred in.

IV. DETECTION OF CONTROL FLOW ERRORS

Fig. 2. Shows how a CFE can be detected. If an illegal jump occurs before added instructions at the beginning of the another basic block (CFE1), then it can be detected by conditional branch in line two by comparing the stored value in the SSR (as the signature of the node) with another one calculated in compile time. If they are not equal, the CFE is detected and the function that is used for correction is called. For CFE2, which is an illegal jump to the middle of another basic block, added instructions at the end of the node can detect CFE2 similar to the case of the additional instructions at the beginning of the node.

The signature of the destination basic block is stored in DSR at the beginning and at the end of the node. It will be used in correction phase for storing the signature of the basic block that the control is transferred to it illegally. If the destination signature is stored only at the beginning, when an illegal jump occurs to middle of the node, we cannot rely on stored value in DSR. For example in Fig. 2 when CFE2 occurs, the signature of the previous basic block (source) was stored in DSR. This value can mislead the CFE handler function. The added instructions at the end of the basic block solve this problem and store the true value in it.

V. AUTOMATIC CORRECTION PHASE

When a CFE is detected through added instructions, the control is transferred to CFE handler function. This function can relocate control to the basic block which not only corrects CFE, also guarantees that corrects the data errors in the program. This is implemented by considering DFG of the program at the design time.

In the previous section we described some problems of prior methods for correction. And we showed that in critical applications this type of correction is not applicable. For handling this problem, the data errors should be considered and finally corrected. Techniques for correcting the data errors by duplicating instructions are presented in [1], [8], [9], [10] among others. The basic concepts of them are duplicating computations and comparing their results in order to detect data errors. But this type of data errors correction has high overhead because of duplicating and comparing. In the rest of this section we explain our correction technique that is more efficient and has less overhead than previous works.

At the design time, we can make DFG and CFG from program code. When the CFE is detected, the signature of the source basic block is already available in SSR and the signature of the destination one is already available in DSR. These two values are given to CFE handler function as inputs. Finally the function can transfer control to the nearest basic block which initialize modified registers between source and destination. We have this information at the design time and can implement the function by considering them.

Fig. 3. Shows three basic blocks from the set of basic blocks in program code, and shows the DFG generated from data dependencies between registers in these basic blocks. Suppose that register2 and register3 initializations are done in basic block1, and register1 initialization is done in basic block2. If CFE1 occurs and the control is transferred from basic block2 to basic block3, then we cannot be sure about accuracy of values in register1 and register3, because of the problems which were previously explained in section3. We have this information at design time and can define the CFE handler function with respect to them. For example suppose that the source basic block is basic block2 and the
destination one is basic block3. Also suppose that the modified registers by the CFE (register1 and register3) are initialized in basic block1 and basic block2. For CFE and data errors correction the control should be transferred to basic block1 therefore the modified registers are re-initialized and computation on them are re-executed after this transmission.

By re-executing the code from basic block1 the first value which was stored in register3, is re-loaded to it again. Also after completing basic block1 and in basic block2 the first value of register1 is re-loaded to it.

If CFE2 occurs, the source basic block is basic block3 and the destination one is basic block1. We can find them from stored signatures in SSR and DSR. The affected registers by this CFE are register1, register2, and register3. Also we know that initializing of register1 is done in basic block2. And the initializing of register2 and register3 are done in basic block1. So we should return to basic block1 in order to load the first values to registers and re-execute computations that had used them.

Fig. 4 shows an implementation sample of the CFE handler function. Conditional branches are defined by considering the DFG at design time. Branch addresses are addresses of the first instruction in the basic blocks of the program code, and we want to transfer the control to them. When an illegal jump occurs to signature update statements, stored values in the SSR and DSR may be wrong and out of the predefined range of the signatures. The last line of the function is defined in order to correct these CFEs.

In our technique for detection and correction of illegal jumps to unused space (partition block), we fill up the partition block with instructions shown in Fig. 5. We use zero as the destination signature value for the partition block in order to distinguish it from other blocks in program code. If the destination of the illegal jump is partition block, then in the CFE handler function the destination does not need to be considered, because no computation related to the program computation is done in. So, at the design time, when we want to implement the CFE handler function, we can ignore the destination basic blocks if its signature value is equal to zero (by checking the signature with conditional branches).

VI. EXPERIMENTAL RESULTS

For evaluation of our technique, we have used the EMU simulator [16] to simulate the program on the 80X86 microprocessors. This simulator can generate console programs that can be executed on any computer that runs x86 machine code (Intel/AMD architecture). We have utilized there well-known benchmarks: Quick Sort and Matrix Multiplication and Linked List for evaluation. Simulation-based fault injection method has been used and
1000 transient faults have been injected on several points in basic blocks of the program. The fault models considered were:

- Branch creation in which a jump instruction to the other basic blocks was inserted in the code.
- Branch creation in which a jump instruction to the partition block was inserted in the code.

In order to detect the generated CFEs which affect the results, we compare the results of fault free execution with faulty ones.

Table 1 shows the correctability results of the programs. It is found that around 93.5% of the control flow errors result in correct output with CCDA compared with around 95.1% with ACCED and 7.2% without them. About 7% of faults on an average resulted in segmentation faults even with CCDA. Segmentation faults are generated due to illegal jumps to signature update statements.

A variable is defined for efficiency estimation of methods in (1). Correction coverage is the percentage of the injected control flow errors which are corrected by the correction techniques, and cost is the average of the memory and performance overheads.

\[
\text{Efficiency} = \frac{\text{Correction Coverage}}{\text{Cost}}
\]

Table 2 depicts comparison among CCDA and ACCED in terms of memory and performance overheads and methods efficiency. Memory overhead consists of added instructions to all basic blocks and added instructions which are used to implement CFE handler function. Performance overhead is generated due to the latency of the CFE handler function and the latency of the effects of it on the execution flow.

According to Table 2, the performance and memory overheads of CCDA are 76% less than ACCED on average, and the efficiency of proposed technique is more than twice the efficiency of ACCED.

Matrix multiplication program has many computational operations and the basic blocks of its code is larger than the other benchmarks. Also the number of the basic blocks is fewer in compare to other two benchmarks. According to it, for matrix multiplication benchmark, memory overhead percentage is noticeably less than quick sort and linked list benchmarks. In contrast to matrix multiplication, linked list program has the fewest computational operations. Therefore the number of its basic blocks is higher than other two benchmarks, and the memory overhead percentage is the highest.

**VII. CONCLUSIONS**

In this paper, we have presented an efficient software technique which can correct control flow errors and data errors automatically. We have explained that only focusing on CFEs correction is not applicable. And also we have described how the data errors can occur and can generate destructive results in output. Therefore the data errors should be considered especially in critical applications. In order to develop this type of detection and correction, firstly for CFEs detection, we have used signature for each node, and then have specified instructions for calculating and checking them at run time. We have implemented a function by considering the control flow graph and the data flow graph of the program code at the design time for automatic
correction of the control flow errors and the data errors. Fault injection experiments show that CCDA, when applied on benchmarks, produces correct results in over 93% of the cases. The latency and the additional memory required for correcting the CFEs and the data errors are considerably less than the duplication based methods which have been recently published.

REFERENCES

[16] 80X86 microprocessor simulator and emulator.
http://www.emu8086.com