CFEDR: Control-Flow Error Detection and Recovery Using Encoded Signatures Monitoring

Lanfang Tan
School of Computer
National University of Defense Technology
Changsha, China
Email: tlf1022@126.com

Ying Tan
School of Computer
Sichuan University
Chengdu, China
Email: tinapgaara@126.com

Jianjun Xu
School of Computer
National University of Defense Technology
Changsha, China
Email: jjxu@163.com

***Abstract*—The incorporation of error detection and recovery mechanisms becomes mandatory as the probability of the occurrence of transient faults increases. The detection of control flow errors has been extensively investigated in literature. However, only few works have been conducted towards recovery from control-flow errors. Generally, a program is re-executed after error detection. Although re-execution prevents faults from corrupting data, it does not allow the application to run to completion correctly in the presence of an error. Moreover, the overhead of re-execution increases prominently. The current study presents a pure-software method based on encoded signatures to recover from control-flow errors. Unlike general signature monitoring techniques, the proposed method targets not only interblock transitions, but also intrablock and inter-function transitions. After detecting the illegal transition, the program flow transfers back to the block where the error occurred, and the data errors caused by the error propagation are recovered. Fault injection and performance overhead experiments are performed to evaluate the proposed method. The experimental results show that most control flow errors can be recovered with relatively low performance overhead.**

I. Introduction

As the dimensions and operating voltages of computer electronics are reduced to satisfy the demand for higher density, functionality, and lower power, their sensitivity to radiation increases dramatically [1]. A plethora of radiation effects in semiconductor devices varies in magnitude from data disruptions to permanent damage, ranging from parametric shifts to complete device failure [2], [3]. Transient faults, which are intermittent faults that do not cause permanent damage but may result in incorrect program execution by altering signal transfers or stored values, are one of the primary concern of commercial terrestrial applications.

In terms of effects, transient errors can be classified into data errors and control flow errors (CFEs). Data errors appear when the contents of variables in memory or in microprocessor registers are altered. CFEs refer to deviations from the program's normal instruction execution flow, such as upsets to the target addresses of branch instructions or the program counter (PC). Studies have shown that CFEs account for 33% [4] to 77% [5] of all transient errors. Therefore, equipping computer systems with CFE detection and recovery mechanisms is necessary.

A multitude of CFE detection techniques has been proposed [6]–[21]. CFE detection techniques can be divided into three classes, namely, hardware-based, software-based, and hybrid techniques. The use of a watchdog processor, which is a small and simple coprocessor that detects errors by monitoring the behavior of the main processor, is a typical hardware-based technique [6]–[12]. In general, hardware-based techniques are very efficient for a single, fixed reliability policy. However, hardware-based techniques are expensive because they require extra hardware modules, such as watchdog processors. Moreover, hardware-based techniques are infeasible for a number of commodity systems. Compared with hardware-based techniques, software-based techniques are more flexible and less costly. A signature monitoring mechanism [15]–[22], [25] appears to be the most universal and effective software-based technique. A control-flow checking by software signatures (CFCSS) [16] can detect most CFEs with a code-size overhead of 25% to 60%. Dynamic signature monitoring (DSM) [17] allows full protection against all illegal interblock transitions. However, the code-size of DSM increases approximately four times and system performance degrades thrice. Researchers have proposed hybrid software-hardware solutions, in which error detection mechanisms are implemented in hardware but are controlled by software running on the processor [13], [14] to gain advantages of both hardware-based and software-based techniques.

However, detecting errors is only part of the path to full fault tolerance. A truly reliable system must be able to recover from errors. Most known current techniques have only addressed CFE detection, but few studies have been conducted on CFE recovery. Generally, a program is re-executed when a CFE is detected. Although re-execution prevents faults from corrupting data, it does not allow application to run to completion correctly in the presence of an error.

Few studies on CFE recovery have been published [23], [24]. In [23], the automatic correction of control-flow errors (ACCE) algorithm for correcting CFEs was proposed. The ACCE adds checking instructions for error detection during compilation time, and a detected error is corrected at the node (basic block) level. After error detection, the program transfers back to the node where the error occurred and continues execution. However, the algorithm exhibits two main drawbacks as follows: (a) The program output may be incorrect because of a number of unrecovered data errors caused by

978-1-4799-1585-9/13/$31.00

the CFE propagation; and (b) If the destination of the illegal transition is a checking instruction, the recovery will fail. In ACCE, at least four instructions are added to each block. The probability of illegal transitions to the checking instructions remains high. Therefore, ACCE is insufficient. In [24], two techniques for automatic CFEs detection and correction were presented. Although data errors caused by CFE propagation are considered, the memory may be corrupted by illegal transitions to store instructions. Furthermore, the overhead of shadow variable duplication increases dramatically to recover data errors, and illegal jumps between two functions are not handled.

The current study presents a pure-software technique, namely, control-flow error detection and recovery (CFEDR) using encoded signatures monitoring, to remove the drawbacks mentioned above. The CFEDR aims to detect the occurrence of a CFE and rolls back to the block where the CFE occurred after the error detection. Moreover, the data errors caused by the CFE propagation are recovered, and the memory is protected from corruption by the CFE propagation. In CFEDR, the program is divided into functions that are made up of storeless basic blocks (SBBs). A unique identification number is assigned for each function. For each SBB, five kinds of encoded signatures (function identification, detection, recovery, transition and mask signatures) are associated during compile time. Based on the associated signatures, checking and recovery instructions are added. During program execution, errors can be detected by comparing the run-time signature with a pre-computed signature. A CFE-handler is designed for each function and a global CFE-handler is designed for the program, respectively. After error detection, CFE-handlers search all blocks to find the one where the error occurred and transfer the program control back to the matched block. Then the recovery instructions are executed to correct the data errors caused by the CFE propagation. Therefore, the program continues execution without being affected by the error.

The rest of the paper is organized as follows. Section II outlines the fault model. Section III discusses the details of the CFEDR technique. Section IV illustrates an example of automatic error recovery. Section V presents the experiments. Finally, conclusions are drawn in Section VI.

II. FAULT MODEL

The current study adhered to the single event upset (SEU) model [29], [30], which states that only one fault may occur during an execution. In this paper, only the CFEs were targeted. Thus, SEUs can be modeled as illegal transitions to other instructions that differ from the expected ones. CFEs can be grouped into the following two categories according to the destinations.

1) Illegal intra-function transitions: they are incorrect jumps within in function. Moreover, they can be classified into interblock and intrablock transitions. Illegal interblock transitions refer to incorrect jumps to other blocks. Illegal intrablock transitions denote incorrect jumps within a block. In general, signature monitoring techniques are only designed to detect illegal interblock transitions, whereas CFEDR technique attempts to detect both interblock and intrablock transitions.

2) Illegal inter-function transitions: they are incorrect jumps between two functions. To our knowledge, this is the first time anyone proposes a signature monitoring technique to handle illegal inter-function transitions.

TABLE I
NOTATION IN CFEDR

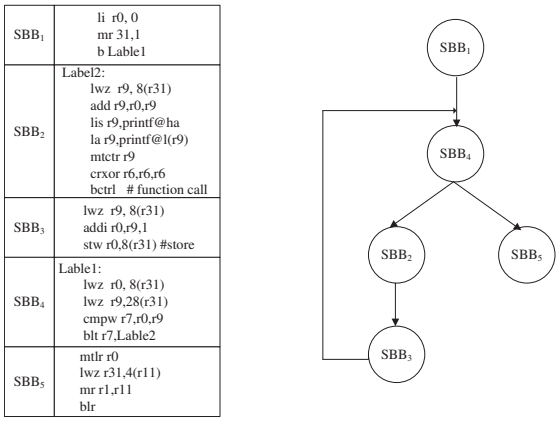
Notations	Definitions
SBB_i	the ith storeless basic block in a function
SBB_{ij}	the j th sub-block of SBB_i divided by the intrablock-check interval. For instance, an SBB has m instructions and the intrablock-check interval is n instructions, SBB can be diveded into $\lceil m/nd \rceil$ sub-blocks.
Suc(SBB_i)	the set of successors of SBB_i
Pre(SBB_i)	the set of predecessors of SBB_i
Num	the number of SBBs in a function
Fn	the number of functions in a program
br_{ij}	the transition from SBB_i to SBB_j
$des(i)$	the identification number of the destination block for transition from SBB_i during an execution. For a transition br_{ij}, $des(i) = j$
D_i	the detection signature of SBB_i
R_i	the recovery signature of SBB_i
B_{ij}	the transition signature of br_{ij}, which is calculated by $(D_i \oplus R_i)$
FID_f	the function ID of the fth function
M_{fi}	a mask flag for function id defined as $\{M_{fi} = 111100..0\}$, where the length of bits with "1" is equal to the length of the function id field.
G	a dedicated register, which contains the run-time signature for error detection, with initial value of the detection signature of the entry block.
P	a dedicated register, which contains the run-time signature for error recovery, with initial value of the recovery signature of the entry block.
T	a temporary register used during the error recovering process.
O	a flag register representing whether a CFE has been detected, with initial value of "0".

III. CFEDR TECHNIQUE

Similar to general signature monitoring techniques, the steps of the CFEDR technique are as follows: 1) Partitioning the program into functions, and dividing the functions into SBBs. 2) Assigning a unique signature for each function, and associating encoded signatures to each SBB. 3) Inserting added instructions. 4) Designing a CFE-handler for each function and a global CFE-handler for the program. The rest of this section discusses the steps above in detail. Notations used in this paper are listed in Table I.

A. Partitioning the Program Code

In the current study, the program code was first partitioned into functions to check the inter-function CFEs. Then, the functions were divided into SBBs to protect the memory uncorrupted by the CFE propagation. An SBB is a sequence of instructions in which no store instruction or branch instruction



(a) a function source code (b) the control flow graph

Fig. 1. A typical source code and its CFG

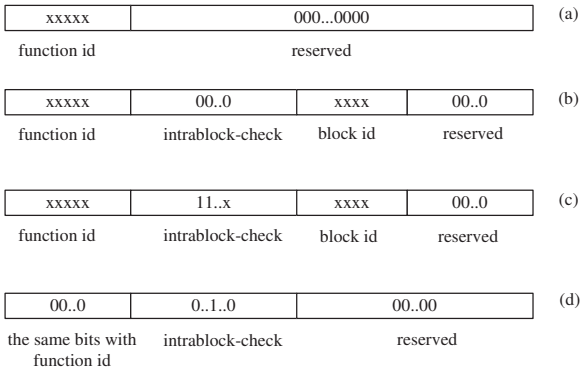


Fig. 2. The structure of signatures

(including function call instructions) exists, except the last instruction [27]. The consecutive store instructions belong to an SBB to reduce the number of SBBs. The structure of a function can be represented using a directed graph, where nodes represent SBBs and arcs represent the control flow transfers between the blocks. This directed graph is called the control flow graph (CFG). Figure 1 shows an example of a CFG.

B. Assigning Encoded Signatures

In CFEDR, five kinds of static signatures are introduced during the compilation phase. These signatures are either 32-bit or 16-bit numbers according to the machine word.

1) FID: each function of a given program is associated with a function identification (FID). Figure 2(a) shows the structure of an FID, where the first field specifies the function id and the remaining bits represent the reserved field. A function id is numbered from 1, and its length can be calculated using $\lceil \log_2^{Fn+1} \rceil$. The reserved field has a value of "0" and has no effect on the signatures.

2) D: in each SBB, a unique signature is introduced for error detection. As shown in Figure 2(b), the detection signature contains four fields, three of which have predefined values and the other one is reserved. The first field represents the FID of the current function. The second field is defined to check illegal intrablock transitions. The length of the second field depends on the size of the max block in the function and the intrablock check interval. Suppose that the intrablock CFEs are checked at an interval of n instructions and the max block has Max instructions. Thus, the length of the intrablock-check field can be defined as $\lceil Max/n \rceil$ and all bits are set to "0". The third field is introduced to identify the block, and the fourth field is reserved.

3) R: in each SBB, a unique signature R is associated for error recovery. As shown in Figure 2(c), R has the same structure as D except for the assignment of the intrablock-check field. If a block has m instructions and the intrablock CFEs are checked at an interval of n instructions, the first $\lceil m/n \rceil$ bits of the intrablock-check field are set to "1" and the remaining bits keep "0".

4) M: a set of mask signatures $\{M_h = 00..100..|h = 1, ..., K\}$ are introduced, where K is the length of the intrablock-check field. Figure 2(d) shows the M_h with three fields. The first field contains the same bits as the function id with value "0". The second field has the same length as the intrablock-check field, where the h-th bit is set to "1" and the remaining bits are set to "0". The third field is reserved with the value "0." In an SBB_i, a set of $\{M_1, M_2, ..., M_{(k,i)}\}$ is associated, where (k, i) denotes the number of associated mask signatures for SBB_i. (k, i) is calculated by $\lceil m/n \rceil$, m denotes the number of instructions and n denotes the intrablock-check interval. According to the definitions above, (1) is satisfied in an SBB_i, where "$\oplus$" is a bitwise XOR operation.

$$R_i \oplus M_i ... \oplus M_{(k,i)} = D_i \tag{1}$$

5) B: for each legal interblock transition, a static signature B is introduced. Then value of B can be calculated as follows:

$$B = D_{source} \oplus R_{destination} \tag{2}$$

Three constraints must be satisfied for the allocation, where $i, k, j = 1, 2, ..., Num$:

$$D_i \neq D_j, \quad if \quad i \neq j \tag{3}$$

$$R_i \neq R_j, \quad if \quad i \neq j \tag{4}$$

$$D_i \oplus D_j \oplus R_{Suc(SBB_j)} \neq R_k, \quad if \quad i \neq j \tag{5}$$

According to the definitions, (3) and (4) are satisfied for the allocation. Equation (5) is used to ensure that when the recovery signature is corrupted, the control flow will not transfer to the incorrect block.

C. Inserting Added Instructions

Three types of instructions are added to detect and recover CFEs. Checking instructions and recovery instructions are inserted in each block. Checking instructions are designed to detect CFEs while recovery instructions are designed to recover the data errors caused by the CFE propagation. Moreover,

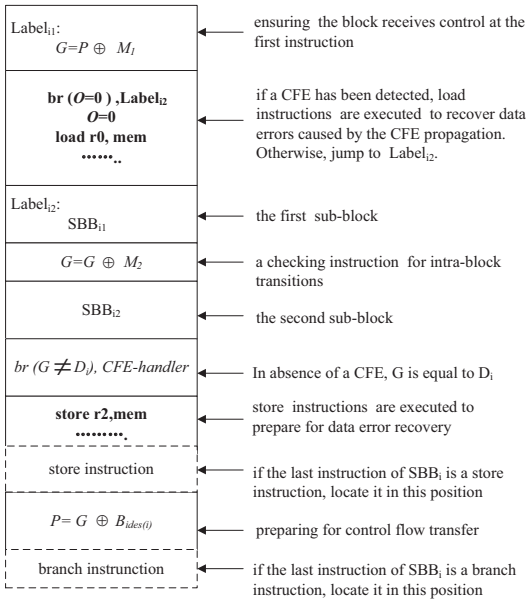


Fig. 3. An SBB with added instructions

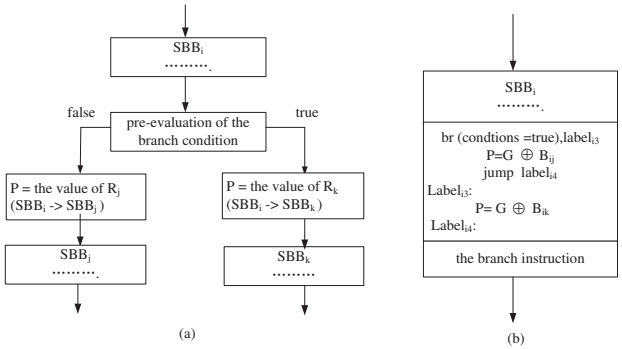


Fig. 4. Branch pre-evaluation in CFEDR

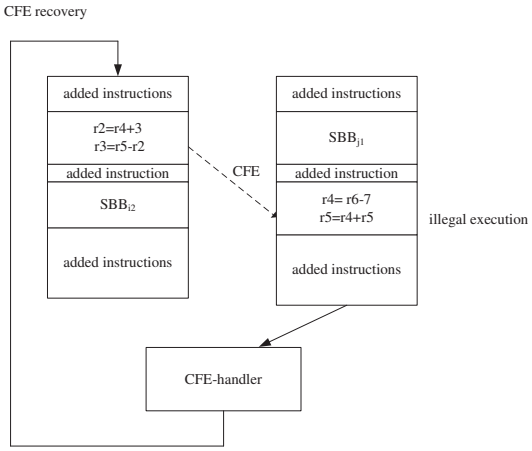


Fig. 5. Data errors caused by CFE

in each function, special instructions are inserted at the entry and exit blocks to check the inter-function CFEs.

Figure 3 shows how instructions are added and located given a block that is divided into two sub-blocks by the intrablock-check interval. The instructions in Figure 3 are the PowerPC-like instructions. A detailed discussion follows.

1) Checking Instructions: For an SBB_i, where i denotes the SBB number, the following three parts of checking instructions are added, as indicated by the italic instructions in Figure 3: 1) Signature generation instructions, such as $G = P \oplus M_1$, $G = G \oplus M_2$,..., $G = G \oplus M_{(k,i)}$ where $M_1, ..., M_{(k,i)}$ denote the mask signatures associated to SBB_i. 2) The compare instruction "br ($G \neq D_i$), CFE-handler" compares the run-time signature with the static detection signature. In the absence of error, G is equal to D_i by (1). 3) The instruction "$P = G \oplus B_{ides(i)}$" prepares the interblock transition from SBB_i to its destination block. If a block has more than one successor, the branch pre-evaluation technique is adopted, in which the condition of the branch instruction is evaluated before the branch is taken. The branch pre-evaluation technique permits the knowledge of which block will receive the program control flow. Therefore, the register is generated as the recovery signature of the destination block by $P = (G \oplus B_{ides(i)}) = (D_i \oplus (D_i \oplus R_{des(i)})) = R_{des(i)}$. Figure 4 (a) illustrates how branch prediction is implemented for a branch block whose last instruction is a branch instruction. Figure 4(b) illustrates how instructions and labels are added and located for the pre-evaluation in CFEDR.

2) Recovering Instructions: CFEs can also result in data errors with error propagation. Figure 5 shows how data errors are generated because of CFE propagation. Suppose an illegal transition occurs from SBB_i to SBB_j. Some instructions in SBB_j are executed until the error is detected by the compare instruction. Then, the program control transfers to the CFE-handler. The program recovers from the CFE and transfers back to the beginning of SBB_i via the CFE-handler. However, returning to SBB_i is not sufficient. Illegal execution instructions in SBB_j and re-execution instructions in SBB_i result in corrupted data flow. For instance, the variables (r4 and r5) are corrupted because of the wrong execution in SBB_j. When the control flow transfers back to SBB_i, corrupted variables (r4 and r5) are used as source operands in the next execution. Therefore, wrong output may be produced. A truly reliable CFEDR should recover data errors caused by CFEs, and thus an application-level checkpointing technique is adopted.

The variables in the assembly program can be grouped into two classes, namely, register variables and memory variables. The division of the program into SBBs protects the memory from the corruption of data errors caused by CFE propagation. Thus, only register variables are included in the checkpoint. Suppose the memory is protected by error detecting /correcting codes [28]. The checkpoint is saved in memory for the rolling back of the system whenever an error is detected. For each register variable, assume that a fixed mapping occurs from the content to an address in the memory, $r_i \xrightarrow{store} mem + i \times 4$, where mem denotes the start address of the checkpoint saved

in the memory.

Since the CFE is recovered at the SBB level, the checkpoint for data errors recovery should be placed at the same granularity. However, the overhead will increase prominently if all registers are included in the checkpoint. Thus, a live-variable analysis [26] is used in CFEDR to reduce the overhead of the checkpoint technique. A live-variable analysis identifies whether the value of variable x could be used along some path in the CFG. If so, x is considered live; otherwise, x is considered dead. Including dead variables in the checkpoint is unnecessary.

In the current study, the data-flow equations are defined directly in terms of $In(SBB)$ an $Out(SBB)$, which represent the set of live variables at the entrance and exit of the SBB, respectively. To derive these equations, two definitions are given. $Def(SBB)$ is the set of variables that are definitely assigned values in the block. $Use(SBB)$ is the set of variables whose values may be used in the block prior to any of their definitions. For the exit block, $Out(SBB_{exit}) = \emptyset$. For each SBB other than the exit block SBB_{exit} [26]:

$$In(SBB) = Use(SBB) \cup (Out(SBB) - Def(SBB)) \quad (6)$$

$$Out(SBB) = \bigcup_{suc(SBB)} In(SBB) \quad (7)$$

These equations can be solved using the algorithm in [26]. The solution is neglected in the current study.

Two notations, namely minimal sufficient rollback data (MSRD) and minimal sufficient checkpoint data (MSCD) are introduced to illustrate which variable in an SBB will be reloaded and saved for the recovery. MSRD is defined as the minimal and sufficient set of variables that needs to be reloaded from the memory to correct the data errors after error detection. Correspondingly, MSCD is defined as the minimal and sufficient set of variables that needs to be stored in memory to prepare for data error recovery during program execution. MSRD can be defined as the set $\{r \mid r \in In(SBB)\}$, where the live variables at the entrance of the block need to be reloaded from memory to correct data errors. MSCD can be illustrated by the set $\{r \mid r \in Out(SBB) \cap Def(SBB)\}$, where the intersection of the variables shared by $Out(SBB)$ and $Def(SBB)$ needs to be stored in memory.

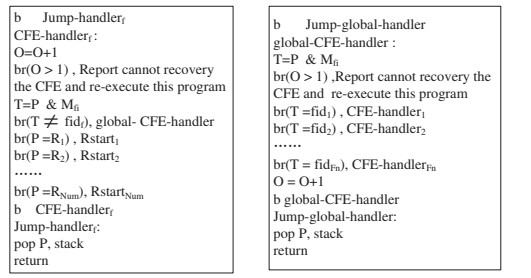
Recovery instructions that consist of load instructions and store instructions are added to implement the checkpoint technique, as indicated by the bold instructions in Figure 3. Load instructions reload variables in MSRD while store instructions store variables in MSCD. Load instructions are only executed after a CFE is detected, whereas store instructions are executed during normal execution. Moreover, some labels are added for assistance. For SBB_i, the new labels and instructions are added as follows: 1) Two new labels, namely, $Label_{i1}$ and $Lable_{i2}$, are inserted. $Label_{i1}$ is inserted before the first checking instruction "$G = P \oplus M_1$", while $Label_{i2}$ is inserted before the first original instruction of SBB_i. 2) A compare instruction "br $O = 0$, $Label_{i2}$" is inserted after the first checking instruction "$G = P \oplus M_1$". This instruction is added to "distinguish" whether the load instructions need to be executed. O is a flag register that represents whether a CFE has been detected. O with value "1" denotes that a CFE has been detected and the load instruction should be executed to recover the data errors. 3) The instruction "$O = 0$ " is inserted after the instruction "br $O = 0$, $Label_{i2}$." This instruction assigns "0" to O to prevent the load instructions from executing in the next block. 4) For each variable r in MSRD, the load instruction "load r, $mem(r)$" is inserted after "$O = 0$", where mem(r) denotes the mapped address calculated by (6). 5) For each variable r in MSCD, store instruction "store r, $mem(r)$" is inserted after checking instruction "br ($G \neq D_i$), CFE-handler".

3) Added Instructions for Inter-function CFEs: In each function, two extra instructions are added at the beginning of the entry block and one extra instruction is added at the end of exit block to check the inter-function CFEs. Examples are indicated by the bold instructions in Figure 7. 1) The instruction "push P, $stack$" is inserted at the beginning of the entry block. This instruction pushes the recovery signature R of the block to which the current function returns into the stack. 2) The instruction "$P = R_{entry}$" is inserted after the instruction "push P, $stack$" "$P = R_{entry}$" assigns the recovery signature R of the entry block to P. 3) The instruction "pop P, $stack$" is inserted before the return instruction of the exit block. Before the function is returned, the recovery signature R of the block which the function returns to is popped off the stack. The design allows the error detection during the nested invocation and recursive invocation via the stack operations.

D. Defining Encoded Signatures

A separate piece of code, called the function CFE-handler, is inserted before the instruction "pop P, $stack$" of each function to handle illegal intra-function transitions. Correspondingly, a global CFE-handler is defined to handle illegal inter-function transitions. The code of the global CFE-handler is inserted before the instruction "pop P, $stack$" of the entry function (the first executable function). Figures 6(a) and 6(b) show the implementation samples of the function CFE-handler and the global CFE-handler, respectively.

In the function CFE-handler, the first jump instruction "b Jump-handler" is inserted to ensure the function CFE-handler is executed only in the following two cases: (1) When a CFE is detected, the program transfers control to the label "$CFE - handler_f$" of the current function. The flag register O is first increased by 1. Then, a compare instruction is performed to judge whether the code has been executed using the value of O. If the value of O is larger than "1", a message is reported and the program is re-executed to recover the current error. Otherwise, a temporary register T generates the FID using the bitwise operation "AND" between the current value of P and the mask flag M_{fi}. If the value of T does not match the FID of the current function, the CFE is handed to the global CFE-handler. Otherwise, branch instructions are executed to search for all the blocks and find one with the



(a) the CFE-handler for sample function (b) the global CFE-handler for sample program

Fig. 6. Implementation of the CFE-handler

recovery signature R that matches with the current value of P. If the matched block is successfully found, the program control is transferred to the beginning of the block for CFE recovery. If no matched block is found, program control transfers to the label of the beginning. Then O is increased by 1, the instruction "br ($O > 1$),..." is executed to judge whether the program continues searching the matched block or re-executes to recover the error. (2) When a CFE jumps to the piece of code directly, the code of function CFE-handler is executed from the destination instruction of the CFE. The process is similar to that of the first case. The compare instruction "br ($O > 1$),..." ensures that all blocks are searched at least once.

The execution of global CFE-handler is similar to that of function CFE-handler except it searches for the function with FID matched with the value of T. After the matched function is found, the program transfers to the function CFE-handler of the matched function to recover the error.

IV. Illustration of Automatic Error Recovery

Figure 7 shows how a typical inter-function CFE is recovered in CFEDR. The program was assumed to have three functions, namely, the $entryfunction$, $function_1$, and $function_2$. An illegal transition is transferred from the SBB_i of $function_2$ to the SBB_j of $function_1$, as indicated by the dotted line. The following four steps are involved in the recovery of CFE.

1) The checking instruction at the end of SBB_j detects that the value of G is different from the detection signature of D_j. The error is then detected and the program control transfers to the $CFE - handler_1$ of $function_1$.

2) In the $CFE - handler_1$, the function id is generated as "$P\&M_{fi}$". Since P is unchanged, it is equal to R_i. Thus, the function id is equal to the FID of $function_2$. Then, the compare instruction "br ($T \neq fid_i$), global-CFE-handler" detects the mismatch and hands the error to the global CFE-handler.

3) The global CFE-handler searches for the function whose FID is equal to the value of T, and it determines that $function_2$ is the matched function and transfers control to $CFE - handler_2$.

4) In the $CFE-handler_2$, compare instructions are executed to search for the block whose recovery signature matches with the current value of P. Since P is unchanged, it is equal

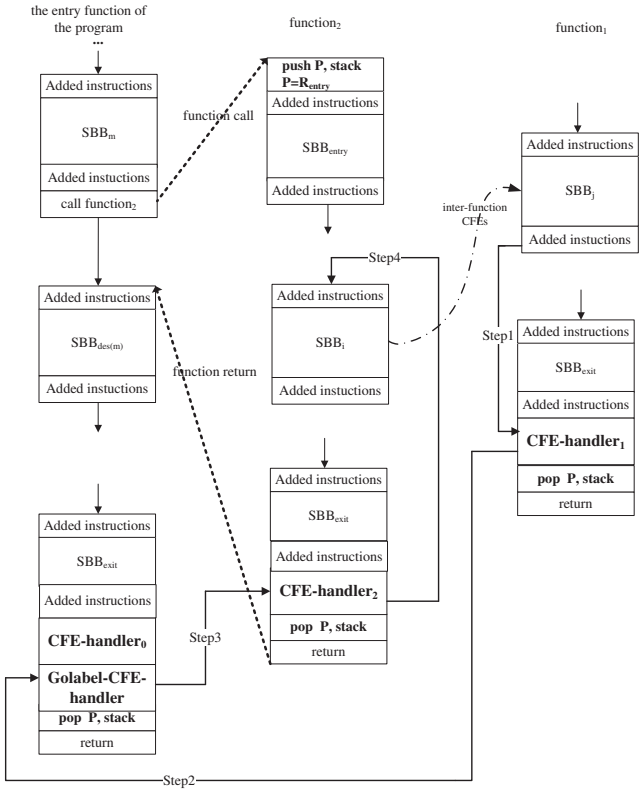


Fig. 7. An example of inter-function CFE recovery

to R_i. Therefore, the SBB_i is found and control flow transfers to the beginning of the block. Data errors caused by the error propagation are then corrected using load instructions. The program continues execution normally.

The recovery of intra-function CFEs is similar to that of inter-function CFEs, except that the intra-function CFEs are only handled by the function CFE-handler.

V. Experimental Results

In the current study, CFEDR has been implemented as an extra pass in GCC(Gnu C compiler) in order to generate the fault tolerant program automatically. The pass was inserted at the level after the register allocation and before the assembly code generation. The PowerPC architecture was the target platform. Via the extra pass, the added instructions with CFEDR are inserted automatically during compilation.Fault injection and performance overhead experiments were performed on the PowerPC 8377 microprocessor and VxWorks operating system to evaluate the CFEDR. The benchmarks SPEC CINT2006 and SPEC CFP2006 were chosen.

First, the source files were compiled to generate assembly codes for the PowerPC architecture. For each benchmark, 2,000 fault tolerant programs with CFEDR and 2,000 original programs were generated. Second, for each assembly program, a CFE was randomly picked among the following [16]: 1) a branch deletion in which a branch instruction is randomly replaced by a nop instruction; 2) a branch creation in which a

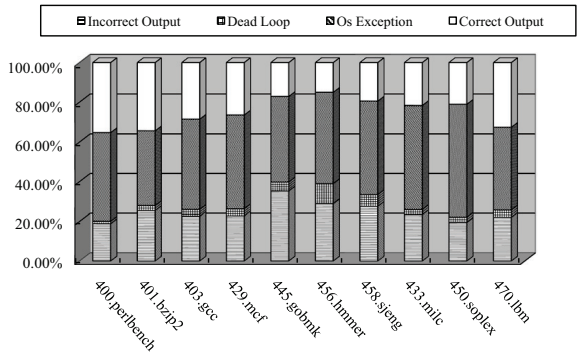


Fig. 8. Fault injection results for original programs

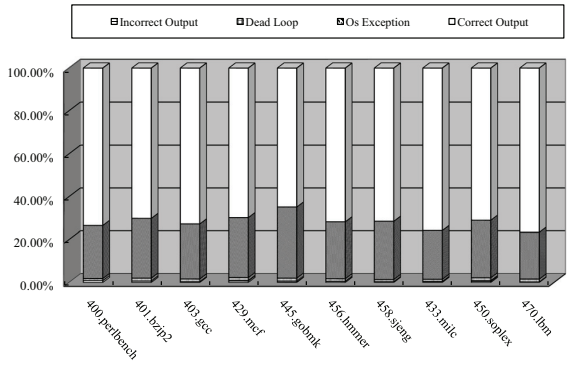


Fig. 9. Fault injection results for fault tolerant programs

jump instruction is randomly inserted into the program; 3) a branch operand change in which the immediate field of the instruction is corrupted. A translator was implemented to automatically insert the CFEs into the assembly programs. Third, for each benchmark, 2,000 fault tolerant programs and 2,000 original programs were complied and executed.

Figs 8 and 9 show the experimental results of the original and fault-tolerant programs, respectively.The results can be classified into four categories: 1) Incorrect Output: The fraction of faults that cause program exits normally with wrong output; 2) Dead Loop: The fraction of faults that make infinite loops in the program, and thus, the processor must be manually stopped; 3) Os Exception: These faults cause the operating system exception. Generally, this percentage of faults is regarded as being detected by the operating system; 4) Correct Output: These faults have no effects on program outputs. As illustrated in Figs 8 and 9, the correct outputs increase prominently with CFEDR.

Assume that the errors that produce either correct output or operating system exception are good errors. The reliability factor is computed as the sum of the percentage of the good errors. Table II compares the percentage of the correct outputs and the reliability factor. Comparing the data between row #2 and row #3, CFEDR is found to increase the correct output in the original programs from 17.8%-30.1% to 69.8%-82.3%, which indicate that CFEDR can recover the program back

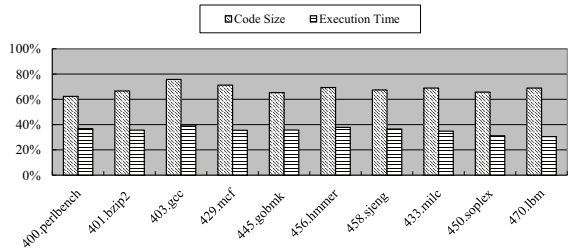


Fig. 10. Performance overhead of fault tolerant programs

into correct states and produce correct output though errors occur. And the reliability factor increases from 38.7%-47.6% to 97.3%-99.3%. We can found that only 0.7%-2.7% of the errors are not fault-safe, such as the intrablock CFEs within the interblock check interval.

Finally, for each benchmark, an original program and a fault tolerant program were compiled and repeatedly executed 500 times. The run-time was recorded, and the overheads for the benchmark programs are shown in Fig. 10.Compared to the original program, the average code-size and run-time overhead of the CFEDR technique are approximately 68.1% and 35.4%, respectively. The code-size overhead is due to the added instructions and CFE-handler codes. As explained earlier, the load instructions added at the beginning of each block and the CFE-handlers are not executed during the normal execution, and thus, they have no effects on the run-time overhead. Therefore,the run-time overhead is lower than code-size overhead.

Table III lists the performance overhead of some representative CFE detection mechanisms, such as The overhead due to CFEDR is not clearly differentiated from those of the pure CFE detection mechanisms [15]–[22]. In other words, CFEDR implements the CFE recovery with relatively low performance overhead.

VI. CONCLUSIONS

In the current study, an efficient algorithm CFEDR for detecting and recovering CFEs was presented. Based on the SEU model, the CFEs were modeled as illegal transitions. First, the program code was divided into SBBs to protect the memory uncorrupted by the CFE propagation. Then, encoded signatures were associated and added instructions were inserted. A function CFE-handler and a global CFE-handler were designed for each function and for the program, respectively. Illegal transitions were detected by comparing the run-time signature with a pre-computed signature. After CFE detection, program control flow was recovered to the SBB where the error occurred via the CFE-handlers. Moreover, the data errors caused by the CFE propagation were recovered to be truly reliable. Thus, program continues execution without being affected by the error. A complete analysis suggests that CFEDR can detect and recover most kinds of illegal transitions and the recovery latency was analyzed. Fault injection and performance overhead experiments were performed on the

TABLE II
FAULT INJECTION EXPERIMENT RESULTS

Results	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	433.milc	450.soplex
Correct(O)	18.6%	25.3%	25.8%	22.6%	28.9%	17.8%	23.5%	30.1%	25.7%
Correct(FT)	73.3%	70.4%	81.3%	77.2%	82.3%	69.8%	75.6%	81.3%	73.6%
Reliability(O)	44.2%	47.6%	38.7%	39.0%	41.8%	38.0%	42.6%	47.4%	38.9%
Reliability(FT)	98.1%	97.5%	98.5%	99.3%	97.3%	98.7%	98.4%	97.5%	98.6%

TABLE III
COMPARISONS OF THE CFE DETECTION MECHANISMS

	CFCSS	CFEDR	ACCE	SIED	DSM	RSCFC	YACCA
Run-time overhead	16.2%-69.2%	30.5%-39.6%	10%-42.6%	227% -310%	200%	110% -186%	47%-254%
Code-size overhead	26.6%- 61.9%	62.3%-75.7%	20%-200%	298%-500%	300%	172%-201%	91%-396%
capability	detect	detect and recover	detect and recover	detect	detect	detect	detect

PowerPC 8377 microprocessor to evaluate the proposed technique. Experimental results show that the benchmark programs with CFEDR produce correct outputs in 76.72% of the cases approximately. The average code-size overhead and execution time overhead were found to be approximately 73.7% and 29.9%, respectively. Compared to the pure CFE detection mechanisms, the performance overhead of CFE recovery is relatively low. The data error recovery technique and the optimization on reducing the code-size overhead and execution overhead will the focuses of future studies.

REFERENCES

[1] R. C. Baumann, *Radiation-induced soft errors in advanced semiconductor technologies*, IEEE Trans. Device and Material Reliability, vol. 5, no. 3, pp.305-316, Sep. 2003.

[2] P. E. Dodd and L. W. Massengill. *Basic mechanisms and modeling of single-event upset in digital microelectronics*,IEEE Trans. Nuclear Science, vol. 50, no. 3, pp. 583C602, Jun. 2003.

[3] F. W. Sexton, *Destructive single-event effects in semiconductor devices and ICs*, IEEE Trans. Nuclear Science, vol. 50, no. 3, pp. 603C621, Jun. 2003.

[4] J. Ohlsson, M. Rimen and U. Gunneflo, *A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog*, in FTCS, 1992, pp. 316-325.

[5] M. A. Schuette and J. P. Shen, *Processor control flow monitoring using signatured instruction streams*,IEEE Trans. Computer, vol. 36, no. 3, pp. 264-277, Mar. 1987.

[6] A. Mahmood and E. J. McCluskey, *Concurrent error detection using watchdog processors -a survey*, IEEE Trans. Computers, vol. 37, no. 2, pp.160-174, Feb. 1998.

[7] A. Benso, S. D. Carlo, G. D. Natale, and P. Prinetto, *A Watchdog Processor to Detect Data and Control Flow Errors*, in Proc. IEEE IOLTS, 2003, pp.144 - 148.

[8] Y. Y. Chen and K. F. Chen, *Incorporating Signature-Monitoring Technique in VLIW Processors*, in Proc. IEEE DFT, 2004, pp. 395-402.

[9] A. Rajabzadeh and S. G. Miremadi, *A Hardware Approach to Concurrent Error Detection Capability Enhancement in COTS Processors*, in Proc. PRDC, 2005, pp. 1-8.

[10] H. C. Lai, S. J. Horng, Y. Y. Chen, P. Z. Fan, and Y. Pan, *A New Concurrent Detection of Control Flow Errors Based on DCT Technique*, in Proc. PRDC, 2007, pp. 201-209.

[11] J. N. Mostafa, S. G. Miremadi, A. Ejlali., *Control-Flow Checking Using Branch Instructions*, in Proc. IEEE/IFIP EUC, 2008, pp. 66-72.

[12] T. Michel , R. Leveugle and G. Saucier, *A New Approach to Control Flow Checking without Program Modification*, in Proc. FTCS, 1991, pp. 334-341.

[13] G. R. Roshan and P. Sai , *A Hybrid Hardware-Software Technique to Improve Reliability in Embedded Processors*, ACM Trans. Embedded Computing Systems, vol. 10, no. 3, pp. 1-16, Apr. 2011.

[14] G. A. Reis, J. Chang, N. Vacharajani, R. Rangan, D. I. August, and S. S. Mukherjee, *Design and evaluation of hybrid fault-detection systems*, in Proc. ISCA, 2005, pp. 148-159.

[15] Y. Y. Chen, and K. L. Leu , *Signature-monitoring technique based on instruction-bit grouping*, IET Proceedings Computer Digital Techniques, vol. 152, no. 4, pp. 527-536, Jul. 2005.

[16] N. Oh, P. P. Shirvani and E.J. McCluskey, *Control-Flow Checking by Software Signatures*, IEEE Trans. Reliability, vol. 51, no. 2, pp. 111-122, Mar. 2002.

[17] B. Nicolescu, Y. Savaria and R. Velazco, *Software detection mechanisms providing full coverage against single bit-flip faults*, IEEE Trans. Nuclear Science, vol. 51, no. 6, pp. 3510-3518, Dec. 2004.

[18] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante, *Soft-Error Detection Using Control Flow Assertions*,in Proc. IEEE DFT, 2003, pp. 57-62.

[19] R. Vemu and J. A. Abraham, *CEDA: Control-flow Error Detection through Assertions*, in IEEE IOLTS, 2006, pp. 151C158.

[20] O. Goloubeva, M. Rebaudengo , R. M. Sonza and M. Violante, *Improved software-based processor control-flow errors detection technique*, in Proc. RAMS, 2005, pp. 583-589.

[21] E. Borin, C. Wang, Y. F. Wu, and Araujo, G. *Software-Based Transparent and Comprehensive Control-Flow Error Detection*,in Proc. CGO, 2006.

[22] S. Makoto , *A Dynamic Continuous Signature Monitoring Technique for Reliable Microprocessors*, IEICE Trans. Electronics, vol. 94, no. 4, pp. 477-486, Apr. 2011.

[23] R. Vemu, S. Gurumurthy and J. A. Abraham. *ACCE: Automatic Correction of Control-flow Errors*, In Proc. ITC, 2007, pp. 1-10.

[24] H. R. Zarandi, M. Maghsoudloo, N. Khoshavi. *Two Efficient Software Techniques to Detect and Correct Control-flow Errors*, in Proc. IEEE PRDC, 2010, pp. 141 - 148.

[25] Y. Sedaghat, S. G. Miremadi and M. Fazeli, *A Software-Based Error Detection Technique Using Encoded Signature*, in Proc. IEEE DFT, 2006, pp. 389-400.

[26] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986, pp. 395-404.

[27] N. Oh , P. P. Shirvani and E. J. McCluskey, *Error detection by duplicated instructions in super-scalar processors*, IEEE Trans. Reliability, vol. 51, no. 1, pp. 63-85, Mar. 2002.

[28] L. Bao , *Error-detecting/correcting-code-based self-checked/ corrected/ timed circuits*, in Proc. NASA/ESA AHS, 2010, pp. 66-72.

[29] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*, in Proc. ISCA, 2000, pp. 25-36.

[30] G. A. Reis, J. Chang, N. Vacharajani, R. Rangan, D. I. August and S. S. Mukherjee, *Design and evaluation of hybrid fault-detection systems*, in Proc. ISCA, 2005, pp. 148-159.