

Low-Cost On-Line Fault Detection Using Control Flow Assertions

Rajesh Venkatasubramanian[†], John P. Hayes[†] and Brian T. Murray[‡]

[†]*Advanced Computer Architecture Laboratory
University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109, U.S.A.*

[‡]*Delphi Automotive Systems
Brighton Technical Center
12501 Grand River
Brighton, MI 48116, U.S.A.*

ABSTRACT^{*}

A control flow fault occurs when a processor fetches and executes an incorrect next instruction. Executable assertions, i.e., special instructions that check some invariant properties of a program, provide a powerful and low-cost method for on-line detection of hardware-induced control flow faults. We propose a technique called ACFC (Assertions for Control Flow Checking) that assigns an execution parity to a basic block, and uses the parity bit to detect faults. Using a graph model of a program, we classify control flow faults into skip, re-execute and multi-path faults. We derive some necessary conditions for these faults to manifest themselves as execution parity errors. To force a control flow fault to excite a parity error, the target program is instrumented with additional instructions. Special assertions are inserted to detect such parity errors. We have developed a preprocessor that takes a C program as input and inserts ACFC assertions automatically. We have implemented a software-based fault injection tool SFIG which takes advantage of the GNU debugger. Fault injection experiments show that ACFC incurs less performance overhead (around 47%) and memory overhead (around 30%) than previous techniques, with no significant loss in fault coverage.

1 Introduction

Embedded computer systems are employed very extensively in safety-critical applications such as spacecraft, airplanes and automotive control. Faulty behavior of such systems may lead to mishaps, so they must be designed to detect faults as early as possible. Faults can be classified into permanent, intermittent and transient faults. Permanent faults result from manufacturing defects or residual design errors in hardware and software components. Intermittent and transient hardware faults are due to environmental effects such as electromagnetic interference (EMI) and alpha particle hits [15]. Embedded systems often operate in harsh environments and encounter inter-

mittent and transient faults [8]. We investigate on-line testing and detection of intermittent and transient hardware faults via their impact on program behavior.

On-line fault detection can be done using hardware or software redundancy. A common hardware technique employs two identical processors to execute the same program; their outputs are compared pin-by-pin to detect faults [10, 15]. Hardware redundancy incurs high cost, and is impractical in the cost-sensitive markets of interest here. In such cases, software-based fault detection is preferred. A simple software technique to detect transients is to re-execute the same program on the same processor and compare the results. The technique requires around 100% performance overhead. Assertion checking [17] i.e., special code that checks some invariant properties of a program, is a low-cost software-based technique. An example is range checking, where the assertions perform boundary checks on the values of program variables in order to detect transients. While such mechanisms are widely used, they are application-specific and their fault coverage is not well understood.

If due to a fault a processor subtracts two numbers instead of adding them or one of the numbers being added is corrupted, then a *data fault* is introduced. The detection of such data faults is application-dependent and it is difficult to develop a systematic technique to detect them. Therefore, we focus on a special class of faults called *control flow faults* that occur when a processor jumps to an incorrect next instruction. Fault injection experiments using heavy-ion radiation [4] and simulation [14] suggest that in embedded controllers control flow faults account for about half of all the errors caused by transients. In this paper, we propose and evaluate a systematic and low-cost testing technique to detect such transient-induced control flow faults.

Many control flow checking techniques proposed in the literature use a *watchdog processor* [10] which is a simple coprocessor that performs concurrent system-level error detection. In a modern pipelined microprocessor with internal caches, a watchdog external to the processor cannot observe the instructions executed by the processor [16]. Most commercial off-the-shelf (COTS) processors

^{*} *This research was supported in part by a contract from Delphi Automotive Systems.*

do not have a built-in watchdog. The high cost of custom-designing a processor to include a watchdog motivates us to consider purely software-based control flow checking techniques.

Alkhalifa et al. [2] propose a technique called Enhanced Control Flow Checking Using Assertions (ECCA) which assigns a unique prime number identifier (BID) to each basic block (a sequence of instructions with a single entry and exit) of a program. When the processor executes a new block, special assertions check the control flow using BIDs. Our experiments suggest that ECCA results in over 150% storage overhead. CFCSS (Control Flow Checking by Software Signatures) [13] is another software-based checking technique that assigns a unique signature s_i to each basic block. During execution, a global variable (or register) G is initialized with the signature of the first block of a program. When control transfers from one basic block to another, CFCSS computes the signature of the destination block from the signature of the source block and the XOR difference between the signatures of the source and destination blocks. Control flow is checked by comparing the computed signature with the assigned signature. The authors of [13] insert control flow checking assertions by hand and their experimental results show around 97% transient fault coverage with 30% to 60% memory overhead.

The key contributions of this paper are a classification scheme for control flow faults and a new control flow checking technique that does not depend on the predecessor-successor relationships between basic blocks. We propose a technique (ACFC) that uses the execution parity of a basic block for fault detection. ACFC adds special variables to a routine to check control flow. We call such variables *execution status* (ES) words. Each basic block is assigned a single parity bit in an ES word. A routine with many basic blocks may require multiple ES words resulting in memory overhead. However, our technique inserts fewer instructions than previous techniques. For example, in Figure 1(c), ACFC instruments blocks 3, 5 and 6 with one instruction, whereas CFCSS [13] inserts two extra instructions in each block. Our experiments with benchmark programs show that ACFC results in significantly less performance and memory overhead -around 47% and 30%, respectively- than previous techniques, with no significant loss in fault coverage. Our fault injection experiments show that ACFC detects around 95% of control flow faults.

In Section 2, we classify control flow faults and discuss their properties. Section 3 discusses the proposed assertion-based control flow checking and presents a systematic way to insert assertions in a program. Section 4 describes a software-based fault injection tool. In section 5, we

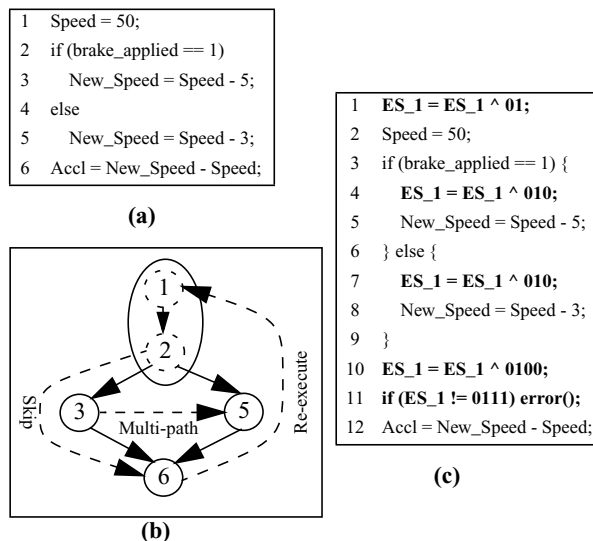


Figure 1. The proposed ACFC method: (a) fragment of a C program, (b) control flow graph (CFG) of the program with various control flow fault types, and (c) the program with extra instructions (boldface) and assertions (boldface)

present fault injection experiment results for various case studies. Section 6 draws some conclusions, with pointers to future research.

2 Control Flow Faults

A program may be divided into a set of (sub)routines. In a language like C, a routine is expressed with a set of variable declarations, assignments, calls to other routines, and control statements (*if-then-else*, *while*, *continue*, *break*, etc.). We do not consider *goto* since it complicates the presentation. However, it is straightforward to extend our technique to incorporate *goto* constructs. The structure of a routine R can be represented by a directed graph with unique entry and exit nodes, where nodes represent basic blocks and arcs represent the flow of control. This graph is called the *control flow graph* (CFG) of R . Figure 2(a) shows CFG corresponding to the C routine in Figure 5(a).

In general, a *control flow fault* occurs when a processor fetches and executes an incorrect next instruction. Faults that cross routine boundaries can be detected by assigning a unique identifier to a function. On entry to a function a global variable is assigned the identifier, and the variable is checked on exit [11]. Hereafter, we consider only control flow faults that do not cross routine boundaries, i.e., the fault is associated with a jump within the current routine. An *inter-block* fault is a jump from one basic block to another, whereas an *intra-block* fault either skips or re-executes a set of instructions within the current basic block. Since basic blocks typically consist of 5 instructions [5], inter-block faults occur with higher probability than intra-block faults. For example, let us assume 50

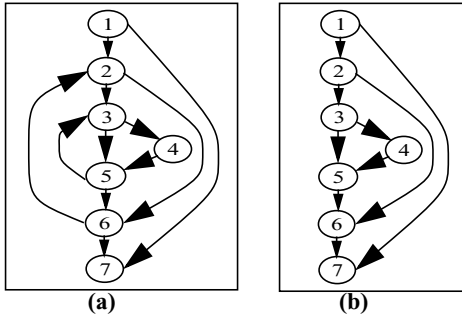


Figure 2. (a) A CFG and (b) the corresponding DAG

basic blocks with 5 instructions each. The probability of a fault resulting in a jump to an instruction in the current basic block is $5/(50*5) = 1/50$, whereas an inter-block jump has a probability of $49/50$. Therefore, we focus on inter-block faults.

If a routine has no loops, then the CFG of the routine's structure is a directed acyclic graph (DAG). For example, the CFG in Figure 1(b) is a DAG. If a routine employs loop constructs such as *while* or *for*, then removing the feedback edges corresponding to the loops from the CFG of the routine results in a DAG. For example, Figure 2(a) shows a CFG of a bubble sort routine shown in Figure 5(a), and Figure 2(b) shows its corresponding DAG. Every DAG represents a partial ordering relation denoted by $a \geq b$. In straight-line code, i.e., code without loops, $a \geq b$ means that the block a is executed before block b when both of them are executed. For example, in Figure 2(a), $3 \geq 6$ and if both blocks 3 and 6 are executed, then block 3 will be executed before block 6.

We now classify the control flow faults with the help of the partial ordering. We use the notation $a \rightarrow b$ to indicate that the control flows directly from the last instruction of block a to the first instruction of block b . We call a fault that results in a jump $a \rightarrow b$ a *skip fault* if $a \geq b$ and $a \neq b$. For example, in Figure 2(a), $2 \geq 5$ and the fault $2 \rightarrow 5$ is a skip fault. A fault resulting in a control transfer from block a to block b is called a *re-execute fault* if $b \geq a$. In Figure 2(a), the fault $4 \rightarrow 1$ is a re-execute fault. If there is no ordering relation between block a and block b , then we call a fault $a \rightarrow b$ or $b \rightarrow a$ a *multi-path fault*. In Figure 1(b), we cannot order blocks 3 and 5, so the fault $3 \rightarrow 5$ is a multi-path fault. It is easily seen that the classification of control flow faults into skip, re-execute, and multi-path faults is complete.

3 Fault Detection

Figure 3 shows the method for instrumenting a C routine to implement ACFC fault detection. We first demonstrate how to check control flow in a routine with no loops.

```

Procedure ACFC(routine  $R$ )
foreach (basic block  $X$  of  $R$ )
  Insert an instruction to complement  $X$ 's parity bit;
  if ( $X$  is an entry or exit block of an if-then-else or case statement)
    Instrument  $X$  to force a parity error on a multi-path fault;
  end;
if ( $X$  is an exit block of a while, for, or do statement)
  Insert assertions to check the values of the ES words;
  Reset the ES words to their values when the loop was entered;
end;
if ( $X$  is an exit block of  $R$ )
  Insert assertions to check all ES words;
end;
end;

```

Figure 3. Procedure to instrument a C routine with ACFC assertions

Then we describe instrumenting loop constructs such as *while*, *do*, *for*, *break*, and *continue*.

A skip fault bypasses the execution of at least one basic block, whereas a re-execute fault executes at least one basic block twice. For example, in Figure 1(b), the skip fault $2 \rightarrow 6$ bypasses the execution of block 3 or 5, and the re-execute fault $6 \rightarrow 1$ executes the block 6 twice. If every block sets a unique variable to a new value, then a skip fault shows up as one or more data faults. Similarly, if a faulty re-execution of a block sets the variable to a wrong value, then a re-execute fault manifests itself as a data fault. Instead of using dedicated variables in each block, ACFC assigns a parity bit to each block. A basic block is assigned a single bit in an ES word. For example, in Figure 1(c), bits 1 and 3 of the ES_1 word are assigned to blocks 1 and 6, respectively. Since only one of the blocks 3 and 5 is executed, we assign the same parity bit (bit 2 of ES_1) to both the blocks. At the start of the program the ES_1 word is initialized to all 0's.

Each basic block is instrumented with an XOR instruction to calculate its execution parity. A fault that results in skipping a block will not set the block's execution parity, whereas a faulty re-execution resets the block's execution parity to its initial value. For example, in Figure 1(c), a skip fault that bypasses the execution of block 1 will not set the first bit of the ES_1 word, whereas a faulty re-execution of block 1 resets the first bit of ES_1 to its initial value. Such faults are detected by special assertions, for example, the assertion in line 11 of Figure 1(c). To prevent compiler optimizations from removing the inserted XOR instructions from a block, we instrument the C program using the inline assembly instructions supported by the GNU C compiler.

Typically, a multi-path fault transfers control from the *then* branch of an *if-then-else* statement to the *else* branch, or vice versa. It can be easily seen that a multi-path fault executes the *then* branch's entry block (denoted by E_t) and the *else* branch's exit block (denoted by X_e), or vice versa.

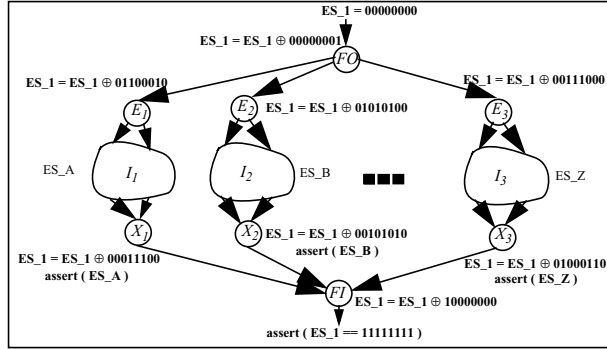


Figure 4. Generic CFG of a nested *if-then-else* construct with proposed instrumentation and assertions

ACFC forces an execution parity error under such multi-path faults by complementing the block X_e 's parity bit in the block E_i , and vice versa. When both blocks E_i and X_e are executed due to a multi-path fault, block X_e 's bit is reset to its initial value resulting in an error.

A generic CFG of a nested *if-then-else* construct is shown in Figure 4. Note that nested *if-then-else* can also be implemented by a *switch* construct with a *break* statement after each of its *case* sections. We assume the *switch* statement has a *default* section, otherwise we add a dummy *default* section. The graph has a fan-out node FO and a fan-in node FI . Each fan-out branch has an entry block E_i , an exit block X_i , and intermediate blocks which are grouped together in Figure 4. The intermediate blocks are denoted by I_i . The blocks FO , E_1 , E_2 , E_3 , X_1 , X_2 , X_3 , and FI are assigned bits 1, 2, ..., 8 of ES_1 , respectively. A single multi-path fault results in execution of the entry block of a fan-out branch and the exit block of a different fan-out branch. For example, in Figure 4, a multi-path fault can result in execution of blocks E_1 and X_2 (or X_3). We force a parity error in such faults by complementing X_2 's and X_3 's parity bits in the block E_1 . In Figure 4, the instruction inserted in a block is shown next to the block. The assertion on ES_1 detects a control fault involving blocks FI , FO , E_i , and X_i . However, a control flow fault can execute the intermediate nodes in an incorrect sequence. Figure 4 assumes that the intermediate nodes I_1 , I_2 , and I_3 are assigned bits of the status words ES_A , ES_B , and ES_Z , respectively. In the corresponding exit blocks, we insert assertions to check the values of such status words. These assertions detect a control fault that affects the execution of intermediate blocks.

Now we discuss the detection of control faults in a routine with loops. Since we assign only one bit to a basic block, the execution parity of the block can be destroyed during loop execution. Therefore, we insert assertions at the end of loop constructs and reset the execution status variables. Figure 5(a) shows a fragment of a bubble sort routine which has nested *while* constructs and an *if-then-*

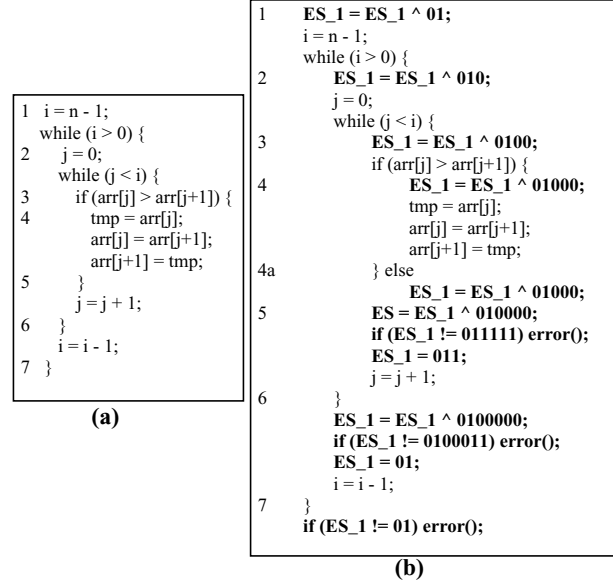


Figure 5. (a) Fragment of bubble sort routine, and (b) the routine with proposed ACFC assertions (boldface)

else. To simplify instrumentation, we convert the *if-then* construct in Figure 5(a) into an *if-then-else* in Figure 5(b) by adding a dummy *else* section. Each basic block is instrumented with XOR instructions as discussed, and assertions are inserted in the last block of each *while* loop to detect control faults. After the check, the ES word is re-initialized to the value it had when the execution entered the *while* loop. If a *break* or *continue* statement is used, then we insert assertions before the *break* (or *continue*) statement to check the ES words, and re-initialize the words to the values expected by the destination block.

Now we show instrumenting a *switch* statement that has no *break* statements after its *case* sections. Such statements are very rarely used. Figure 6 shows the generic control flow graph of such a *switch* statement. The bits 1, 2, ..., 7 of the ES_1 word are assigned to the blocks FO , E_1 , X_1 , E_2 , X_2 , E_3 , and X_3 , respectively. We assume that E_3 is the entry node of the (dummy) *default* section. The blocks FO , E_3 , and X_3 are executed once, independent of the inputs to the target program. Therefore, we instrument blocks FO , E_3 and X_3 with a single XOR instruction. The instrumented instructions are shown next to the blocks in Figure 6. The execution of all other blocks depends on the inputs to the target program. The exit block of a fan-out branch is executed if and only if the entry block of the branch has been executed already. Similarly, if the entry block a fan-out branch is executed, then the entry block of the next branch is also executed. For example, in Figure 6, if block E_1 is executed, then block E_2 is also executed. However, note that block E_2 can be executed even when E_1 is not executed. We instrument the CFG as shown in Figure 6.

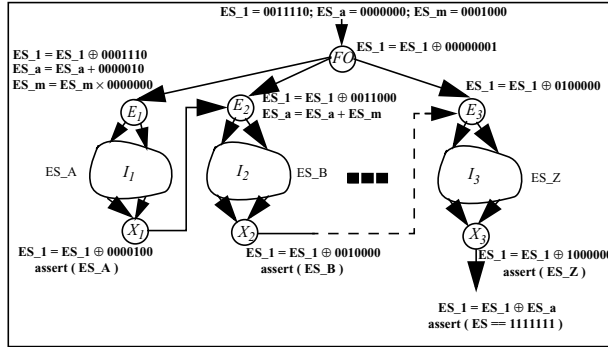


Figure 6. Generic CFG of a *switch* construct with no *break* statements, and the proposed assertions (boldface)

Type	Description
AddIF	Address error resulting in executing a different instruction
AddIF2	Address error resulting in executing two different instructions
AddOF	Address line error when a data operand is fetched
AddOS	Address line error when an operand is stored
DataIF	Data line error when an opcode is fetched
DataOF	Data line error when an operand is loaded
DataOS	Data line error when an operand is stored
CndCR	Errors in condition code flags

Figure 7. Fault types in the FERRARI fault injection system [7]

We developed a preprocessor that takes a C program as input and automatically inserts ACFC assertions as outlined in Figure 3. The preprocessor parses the program and generates a parse tree [1] to analyze the nested control constructs and instrument the program. We use a modified breadth-first search (BFS) algorithm [1] to traverse the parse tree, and assign a parity bit to each block. Then, the preprocessor traverses the blocks in program order and instruments them.

4 Fault Injection Tool

To evaluate the fault coverage provided by the ACFC method, we have developed a software-based fault injection tool SFIG (Software-based Fault Injection using *gdb*). Although several fault injection toolkits have been reported [3, 7], either they are not suitable for our purpose or they are no longer available, so we decided to develop our own. SFIG takes a target program, an instruction address, an iteration number and a fault type as inputs. The fault is injected when the program counter reaches the given instruction address. If that address is within a loop, then the program counter may reach the same address many times during execution. So, the iteration number specifies the iteration at which the fault should be injected. Our injector uses the GNU debugger *gdb* to inject faults and is capable of injecting the extensive set of transient

Program (optimized)	Memory overhead %			Performance overhead %		
	ECCA	CFCSS	ACFC	ECCA	CFCSS	ACFC
Bubble sort	490.2	141.5	112.2	622.7	185.8	136.2
Matrix mult	303.2	96.0	54.0	260.1	119.2	49.0
Quick sort	409.2	85.3	53.2	402.0	111.4	41.2
8-Queens problem	427.7	109.1	80.5	545.0	155.2	120.6
Binary tree search	372.5	64.9	48.0	515.9	102.2	90.9

(a)

Program (unoptimized)	Memory overhead %			Performance overhead %		
	ECCA	CFCSS	ACFC	ECCA	CFCSS	ACFC
Bubble sort	178.7	56.5	47.2	125.7	37.5	32.0
Matrix mult	148.4	54.4	30.2	36.6	18.6	4.3
Quick sort	132.2	29.6	18.8	131.0	40.3	13.7
8-Queens problem	208.4	59.9	45.6	200.7	61.9	48.7
Binary tree search	213.8	40.7	30.3	276.8	60.2	53.9

(b)

Figure 8. (a) Overhead comparison with and (b) without compiler optimization

fault types described in FERRARI [7] and summarized in Figure 7. SFIG is written in Python [9] and currently works on Sun Sparc machines with the Solaris operating system.

An injected fault can sometimes result in an infinite loop, so the *alarm* system call is used to set a predetermined deadline. If the program does not finish before the deadline, the operating system terminates the program by sending the signal *SIGALRM*. SFIG collects statistics of various types of signals (interrupts) received after fault injection, and determines the method used for detecting the injected faults. Faults are detected by the following four mechanisms.

- **Operating system:** This detects faults that result in memory access violations, illegal instructions, and the like.
- **Time-out:** This detects faults resulting in infinite loops.
- **User checks:** These are programmer-inserted debugging checks and help to detect some transients.
- **Assertions:** These detect the faults that result in an invalid control flow.

Exhaustively applying all possible faults is impractical. The fault injection experiments reported in the literature [4, 7] choose random subsets of the possible faults. We employ a systematic pseudo-exhaustive technique that injects faults in every basic block of the target program. First the target program is executed with the given input, and the execution counts of each block are determined. Then, a fault list is generated for each block. If a block is executed multiple times, then different iterations of the block are considered. For all chosen iterations of the block, a user-specified number of faults are injected. For example, the user can specify that two AddIF faults should be injected in a block.

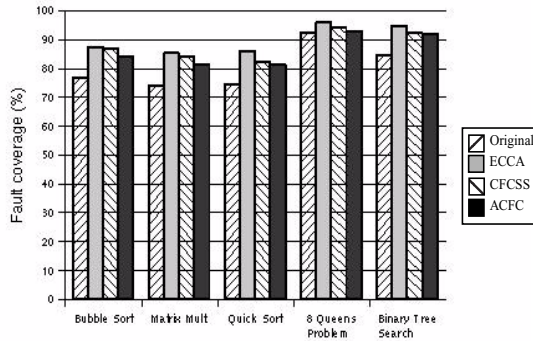


Figure 9. Comparison of fault coverage results of the test programs

5 Experimental Results

To evaluate our proposed assertion-based technique for control flow checking, we carried out a series of fault injection experiments using the SFIG tool. We compare ECCA, CFCSS, and ACFC using five Stanford integer benchmark programs [6]: bubble sort, matrix multiplication, quicksort, 8 queens problem, and binary tree search manipulation. Three different versions of each program with ECCA, CFCSS, and ACFC assertions were written. The programs were compiled with and without compiler optimization (using the *-O2* option of *gcc*). Figure 8 compares the memory and performance overhead. The overhead numbers are calculated by counting the number of extra instructions used by the checking technique. We can see that compiler optimization results in high overhead numbers. Typically, optimization does not affect the number of basic blocks significantly, but it reduces the number of instructions required to implement a basic block. Since the average number of extra instructions inserted in a basic block by a checking method remains the same with and without optimization, the overhead seems to increase significantly due to compiler optimization.

From the results in Figure 8 it can be observed that with compiler optimization ACFC incurs less performance (around 47%) and memory (around 30%) overhead than the CFCSS method. The ECCA method has high overhead because assertions are inserted in every basic block, whereas CFCSS and ACFC reduce the overhead by using a single assertion for several blocks.

For each test case faults are first injected in the unmodified target program to evaluate the normal fault detection capabilities of the operating system. Then, faults are injected in the modified versions of the program with ECCA, CFCSS and ACFC assertions. We used optimized executables for fault injection. In all, we injected around 100,000 faults. Some of the injected faults do not result in an error; such faults are not considered in Figures 9 and

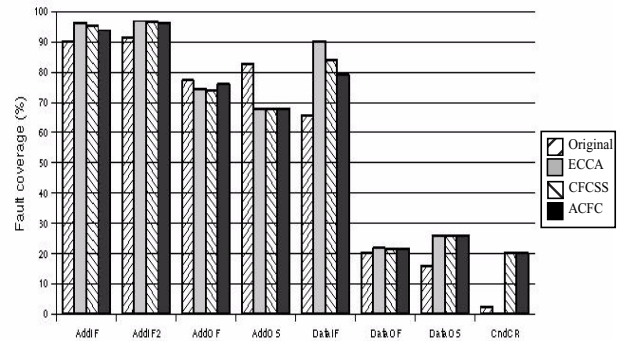


Figure 10. Comparison of fault coverage with respect to fault types

10. Figure 9 shows that ACFC improves the overall fault coverage by around 6% from the original program (which has no control flow checking assertions). Figure 10 compares the fault coverage for the various injected fault types. ACFC covers around 95% of AddIF and AddIF2 faults. These fault types result in execution of incorrect next instructions, so they directly map to control flow faults.

From Figures 9 and 10, we see that ECCA achieves the highest fault coverage. But, as the overhead results in Figure 8 suggest, it has significantly higher overhead than the other two techniques. The coverage of the CFCSS technique is slightly more (around 1.5%) than that of ACFC. The reason is the following. Assume that the average size of a basic block is 5 instructions [5]. The CFCSS method normally instruments a block with 3 extra instructions, whereas ACFC normally adds only 1 extra instruction. A fault during the fetch of an extra instruction is easily detected because it corrupts the execution status word. Since the probability of injecting such a fault is higher for CFCSS (3/8) than it is for ACFC (1/6), CFCSS has slightly higher coverage.

To further demonstrate the capabilities of ACFC, we used our preprocessor to insert assertions into relatively big benchmark programs such as fast fourier transform (FFT) and compress, and measured fault detection using the SFIG tool. The ACFC assertions result in around 20.3% and around 10.6% performance overhead in the FFT and compress programs, respectively. Figure 11 shows the coverage provided by various detection mechanisms discussed in Section 4. It can be observed that address and data bus faults during instruction fetch are detected effectively by ACFC assertions. ACFC detects all inter-block faults that jump within a routine except a jump to the entry block of the routine. Typically, the entry block initializes all ES words to a valid state, so the fault may go undetected. However, the probability of occurrence of such faults are very low.

Fault type	Fault detection mechanism				Undetected faults	Overall coverage
	OS	Time-out	User checks	ACFC		
AddIF	79.3%	0%	0.7%	12.1%	7.9%	92.1%
AddIF2	79.6%	0%	0.3%	10.4%	9.7%	90.3%
AddOF	72.9%	0%	0%	0%	27.1%	72.9%
AddOS	73.3%	0%	0%	0%	26.7%	73.3%
DataIF	40.6%	0.2%	0.3%	17.3%	41.6%	58.4%
DataOF	21.3%	0%	0%	0%	78.7%	21.3%
DataOS	10.0%	0%	0%	0%	90.0%	10.0%
CndCR	0%	0%	22.3%	0%	77.8%	22.3%

Figure 11. Coverage results of FFT and compress programs with ACFC assertions

6 Conclusions

We have presented a new assertion-based control flow checking technique ACFC (Assertions for Control Flow Checking) that instruments a program block with extra instructions to calculate the block's execution parity. Control flow faults are detected by comparing the calculated parity of a block with the pre-computed parity. The proposed technique is relatively low in cost since it does not require any special hardware, and has much less performance and memory overhead than the previously proposed software-based techniques. We developed a preprocessor that automatically inserts ACFC assertions into a C program. Our fault injection experiments show that ACFC detects around 95% of all control flow faults, and improves the overall coverage by around 6% compared to the original program. We have introduced a software-based fault injector SFIG that implements a comprehensive (pseudo-exhaustive) technique to inject faults into every basic block. We have also presented a new classification of control flow faults, and necessary conditions for such faults to excite parity errors. We believe that these conditions can be generalized for systematically determining the control flow coverage provided by arbitrary user-inserted assertions.

7 References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1988.
- [2] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham, "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, pp. 627-641, June 1999.
- [3] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for Experimental Evaluation of Dependability in Modern Computers", *IEEE Trans. on Software Engineering*, vol. 24, pp. 125-136, Feb. 1998.

- [4] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", *Proc. of Intl. Symposium on Fault Tolerant Computing*, pp. 340-347, 1989.
- [5] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, 1996.
- [6] J. Hennessy, *Stanford Integer Benchmarks*, Personal Communications, 1988.
- [7] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a flexible software-based fault and error injection system", *IEEE Trans. on Computers*, vol. 44, pp. 248-260, Feb. 1995.
- [8] J. H. Lala and R. E. Harper, "Architecture Principles for Safety-Critical Real-Time Applications", *Proc. of the IEEE*, vol. 82, pp. 25-50, Jan. 1994.
- [9] M. Lutz, *Programming Python*, O'Reilly, Sebastopol, CA, 1996.
- [10] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb. 1988.
- [11] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection", *Proc. of Intl. Symposium on Fault Tolerant Computing*, pp. 328-335, 1992.
- [12] N. Oh, S. Mitra, and E. J. McCluskey, "ED⁴I: Error Detection by Diverse Data and Duplicated Instructions", *IEEE Trans. on Computers*, vol. 51, pp. 180-199, Feb. 2002.
- [13] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control Flow Checking by Software Signatures", *IEEE Transactions on Reliability*, vol. 51, pp. 111-122, Mar. 2002.
- [14] 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", *Proc. of Intl. Symposium on Fault Tolerant Computing*, pp. 316-325, 1992.
- [15] D. K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [16] M. Z. Relá, H. Madeira, and J. G. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks", *Proc. of Intl. Symposium on Fault Tolerant Computing*, pp. 394-403, 1996.
- [17] S. H. Saib, "Executable Assertions - An Aid to Reliable Software", *Proc. of 11th Asilomar Conference on Circuits, Systems and Computers*, pp. 277-281, 1978.