Low-cost on-line fault detection using control flow assertions

Rajesh Venkatasubramanian,
John P. Hayes,
Brian T. Murray

Advanced Computer Architecture Laboratory
University of Michigan

Delphi Automotive Systems
Brighton Technical Center

Proceedings of the 9th IEEE International On-Line Testing Symposium (IOLTS'03)

Presented by Yeh Tsung-Yu

Outline

- Introduction
- Control Flow Faults
- Fault Detection
- Fault Injection Tool
- Experiment Results
- Conclusion

Introduction

- Why we need on-line fault detection ?
  - Faulty behavior of embedded systems may lead to mishaps, so they should detect faults as early as possible.
- Faults can be classified into permanent, and transient faults.
  - Author propose a mechanism to detect control flow fault due to transient fault.
  - At last, we evaluate this mechanism by fault injection, like simulation, or heavy-ion radiation.

Introduction

- On-line fault detection can be done using hardware or software redundancy.
  - Hardware:
    - Two identical processors to execute the same program, their outputs are compared pin-by-pin to detect faults.
    - Disadvantage: It's impractical in the cost-sensitive markets
  - Software:
    - A simple technique to detect transients is to re-execute the same program on the same processor and compare the results.
    - Disadvantage: The technique requires around 100% performance overhead

Introduction

- As a result, a low-cost software-based technique is considered.
  - Assertion checking – insert check code in target program.
- Fault model due to transient fault.
  - Data fault : For ex, processor subtracts two numbers wrongly.
  - Control flow fault : Processor jumps to an incorrect next instruction.
- Here we propose a systematic technique to detect such transient-induced control flow faults.
  - Data faults is application-dependent, it is difficult to use a systematic way to detect it.

Introduction

- Overview of this control flow fault
  - Don’t check control transfer between subroutine call, and between library function call.
  - Don’t check intra-block faults because the probability is not so high.
  - We just focus on inter-block faults within subroutine.

Introduction

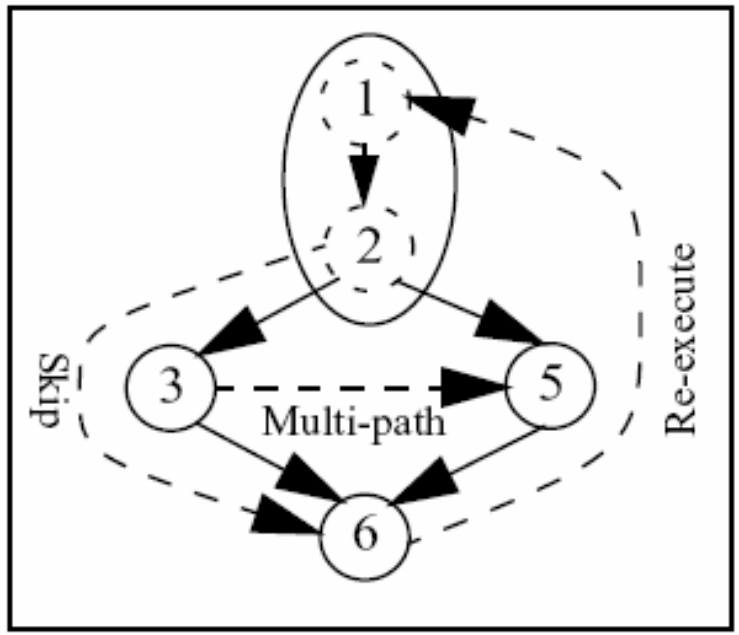
- What's definition of basic block?
  - A sequence of instruction which will be executed one by one sequentially.
  - Head : locate next to conditional branch instruction.
  - Tail : conditional branch instruction.

Control Flow Faults

- Our classification of control flow fault
  - Fault type : skip, re-execute, multi-path.
  - We insert XOR operation in every basic block, every block “only” modify their own bit.
  - These fault will make ES word reset to initial state.(ES: execution state, assigned to subroutine)
- Detection : Finally compare the ES word.
- What information could help mapping detected fault to fault type? See next slide.
  - ES word sometimes can’t exactly help mapping fault type.

Control Flow Faults

```
1    Speed = 50;
2    if (brake_applied == 1)
3        New_Speed = Speed - 5;
4    else
5        New_Speed = Speed - 3;
6    Accl = New_Speed - Speed;
```




```
1    ES_1 = ES_1 ^ 01;
2    Speed = 50;
3    if (brake_applied == 1) {
4        ES_1 = ES_1 ^ 010;
5        New_Speed = Speed - 5;
6    } else {
7        ES_1 = ES_1 ^ 010;
8        New_Speed = Speed - 3;
9    }
10   ES_1 = ES_1 ^ 0100;
11   if (ES_1 != 0111) error();
12   Accl = New_Speed - Speed;
```

Control Flow Faults

```
1   i = n - 1;
    while (i > 0) {
2       j = 0;
        while (j < i) {
3           if (arr[j] > arr[j+1]) {
4               tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
5           }
            j = j + 1;
6       }
        i = i - 1;
7   }
```

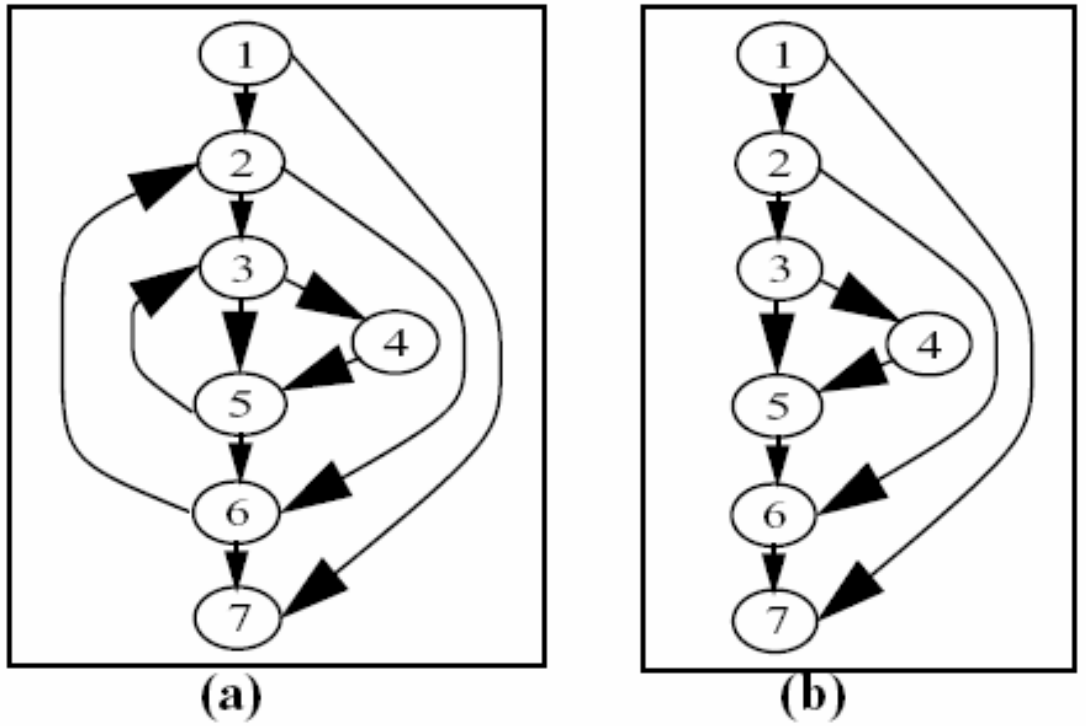


Figure 2. (a) A CFG and (b) the corresponding DAG

- CFG : Control Flow Graph
- DAG : Direct Acyclic Graph
- We could transform CFG to DAG by removing the loop feedback edge.

Control Flow Faults

- How DAG help us to map detected fault to appropriate type?
  - DAG means the normal execution flow. for ex, block 2 execute before block 5. (2>=5)
  - We call a fault that results in a jump (*a → b)* a *skip fault* if a>= b.
  - jump (*a → b) if b>=a : re-execute fault.*
  - jump (*a → b) if no order relationship : multi-path fault.*

Fault Detection – switch structure

- Besides the “if-else” discussed before, we still have other control structures to solve.
- Here we discuss switch (as same structure as nested if-else).
- In Figure 4, a multi-path fault can result in execution of blocks *E1* and *X2* (or *X3*).
  - We force a parity error in such faults by complementing *X2*’s and *X3*’s parity bits in the block *E1*.

Fault Detection – switch structure

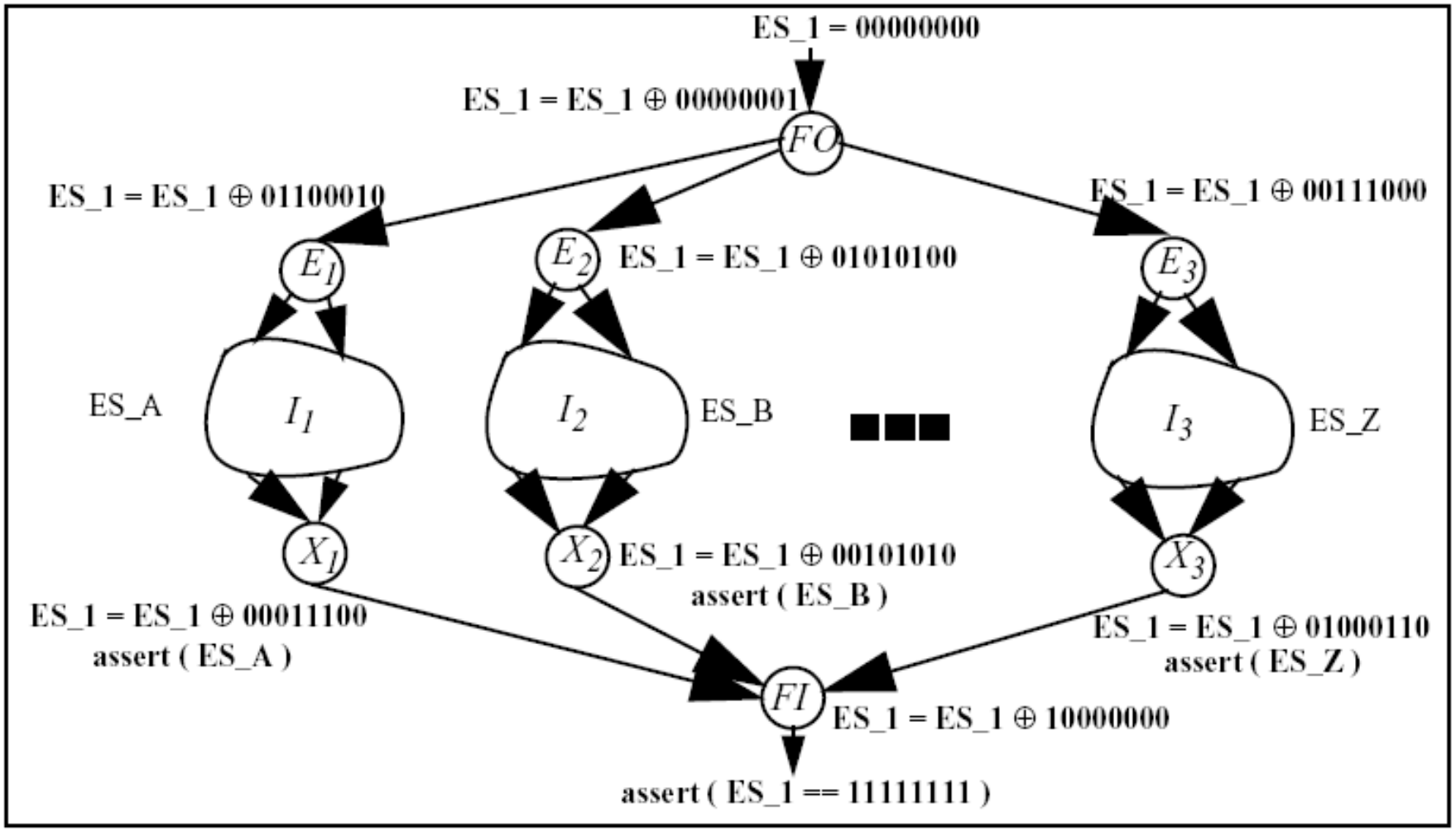


Figure 4. Generic CFG of a nested *if-then-else* construct with proposed instrumentation and assertions

Fault Detection - loop

- Now we discuss detection within loop.
- Since we assign only one bit to a basic block, the bit of the block should be destroyed(re-initial) during loop execution.
- Therefore, we insert assertions at the end of loop constructs and reset the execution status variables.

Fault Detec

```
1   i = n - 1;
    while (i > 0) {
2       j = 0;
        while (j < i) {
3           if (arr[j] > arr[j+1]) {
4               tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
5           }
            j = j + 1;
6       }
        i = i - 1;
7   }
```

(a)

```
1       ES_1 = ES_1 ^ 01;
        i = n - 1;
        while (i > 0) {
2           ES_1 = ES_1 ^ 010;
            j = 0;
            while (j < i) {
3               ES_1 = ES_1 ^ 0100;
                if (arr[j] > arr[j+1]) {
4                   ES_1 = ES_1 ^ 01000;
                    tmp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = tmp;
4a              } else
                    ES_1 = ES_1 ^ 01000;
5               ES = ES_1 ^ 010000;
                if (ES_1 != 011111) error();
                ES_1 = 011;
                j = j + 1;
6           }
            ES_1 = ES_1 ^ 0100000;
            if (ES_1 != 0100011) error();
            ES_1 = 01;
            i = i - 1;
7       }
        if (ES_1 != 01) error();
```

(b)

Fault Injection Tool

- we have developed a software-based fault injection tool SFIG (Software-based Fault Injection using *gdb*).
- SFIG is written in Python.
- SFIG takes a target program, an instruction address, an iteration number and a fault type as inputs.
- Here fault types is presented in the “FERRARI fault injection system”.

Experiment Results

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

Experiment Results

- Fault coverage :
  - Percentage of some type of fault that can be detected during the test of an electronic system, usually an integrated circuit.

Experiment Results

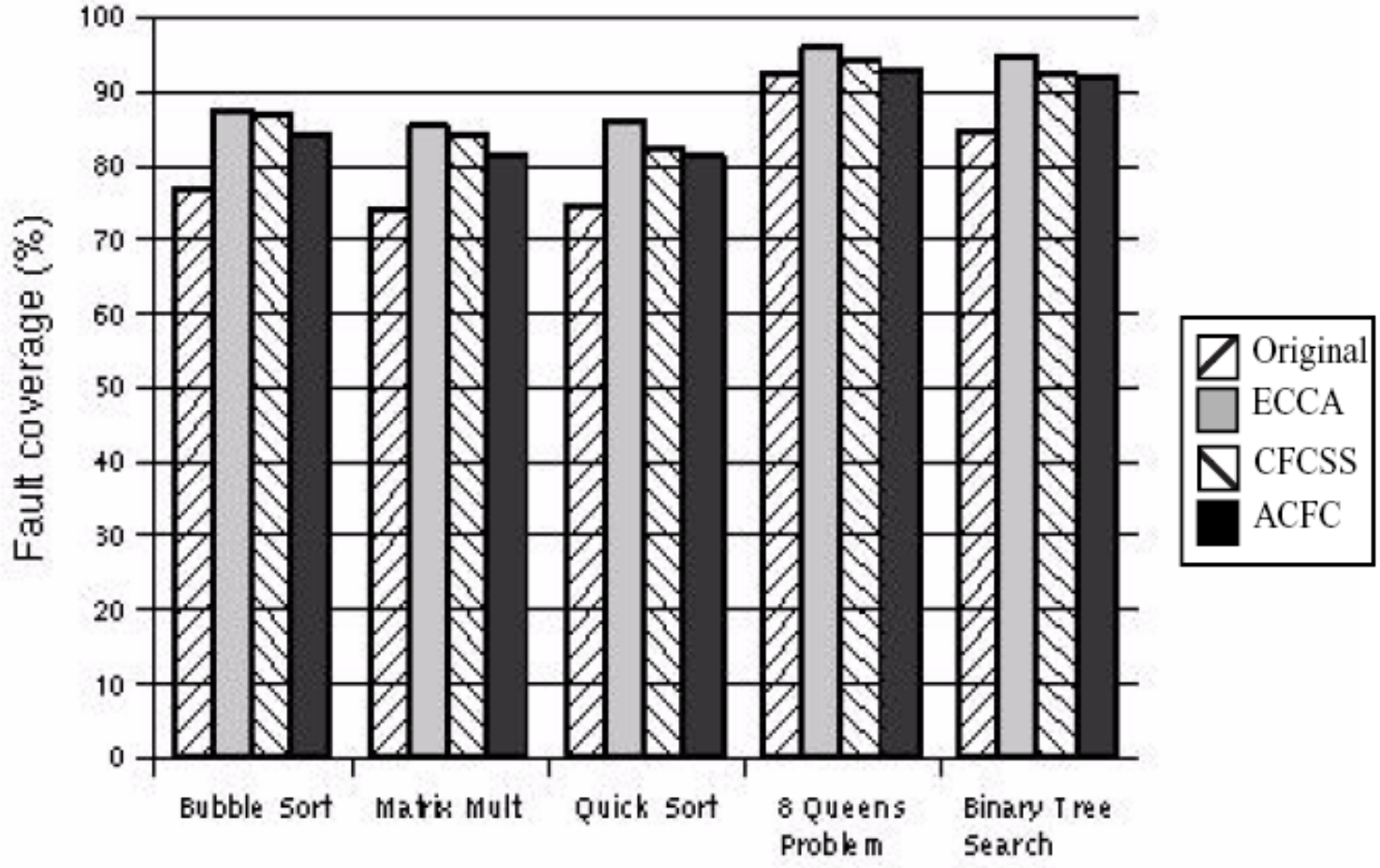


Figure 9. Comparison of fault coverage results of the test programs

Experiment Results

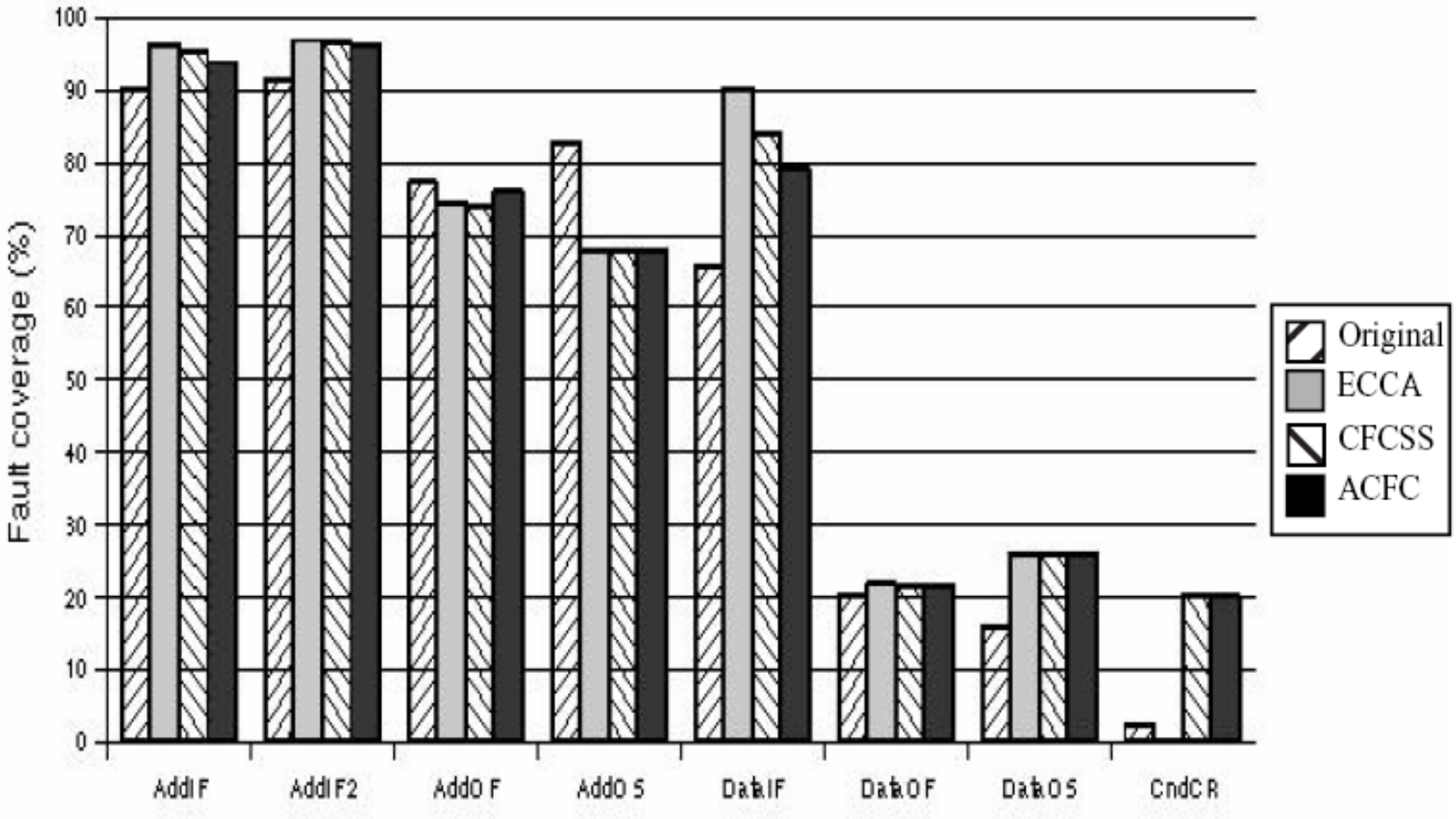


Figure 10. Comparison of fault coverage with respect to fault types

Conclusion

- Key contribution:
  - Classification of control flow fault.
  - Improve performance but only incur less fault coverage.
  - Systematic : preprocessor automatically patch the source code.