SoftWare Implemented Fault Tolerance (SWIFT)
Error Detection by Duplicating Instructions (EDDI) [1]

- Insert a shadow (duplicate) instruction for every master (original) instruction
  - Master and shadow instructions use different registers
- Compare the results before stores

(a) Original Code

```
ld r12=[GLOBAL]
add r11=r12, r13
st m[r11]=r12
```

(b) EDDI Code

```
1: ld r22=[GLOBAL+offset]  
   add r11=r12, r13
2: add r21=r22, r23
3: cmp.neq.unc p1, p0=r11, r21
4: cmp.neq.or p1, p0=r12, r22
5: (p1) br faultDetected
   st m[r11]=r12
6: st m[r21+offset]=r22
```
Comparing at Branches

For correctness, need to verify
- What is getting stored
- Where it's getting stored
- Is this store supposed to happen

‘What’ and ‘Where’ checks require comparing operands

Last check requires comparison at every branch as well
Error During Branch Execution

- Comparison before branch indicates no error so far
- Branch executes ...
- ERROR while executing the branch
- Result: Incorrect target

Need for control flow checking

Basic Block: Sequence of instructions without any branching in between, may be at the end

Illegal Branch
Control Flow Checking by Software Signatures (CFCSS) [2]

Aim: Avoiding illegal branching

Key Idea: Source node and destination node uniquely determine the branch

Solution:
1. Assign a unique signature to every node (basic block) at compile time.
2. Store signature and signature difference with each node.
3. Maintain a general signature register (GSR) containing signature of the current node.
4. Add the difference stored with destination node to the current GSR and compare if it matches the signature of the destination node.

\[ 1^{(1^5)} = 5 \rightarrow \text{Legal} \]
\[ 1^{(2^4)} \neq 4 \rightarrow \text{Illegal} \]

- CFCSS detects legality of the branch
- But doesn’t detect the correctness of the branch

Source block stores signature difference in RTS
Target updates GSR by adding RTS to it
SWIFT Contribution 2: Store Control Flow Optimization

Observation: Only stores are problematic

Optimization: Perform control flow checking only for those nodes that has a store. RTS and GSR computation happens in every block.

Correct path: 1->2->4
Actual path taken: 1->5->4
4 has a store but 2,5 don’t.

GSR = GSR^RTS = 1^ (1^2) = 2
RTS = 5^4

GSR = 2^ (5^4) != 4
ERROR
**SWIFT Contribution 3: Branch Optimization**

Observation: Control flow checks are super set of comparisons performed before executing branches. So latter can be eliminated.

```
... bne $s,$t,L1 ...
L1: ...
```

Without Predication

```
... cmp.ne.s p1, p0=$s,$t (p1) br L1 ...
L1: ...
```

With Predication

```
... cmp.ne.s p1, p0=$s,$t (p1) br L1 ...
L1: ...
```

block 1

block 2
Before jumping to target, RTS is evaluated; if there was an error before branching, then RTS evaluated would be incorrect, and detected later on.

Code with duplicated instructions

```c
cmp.ne.s p1, p0=$s,$t
cmp.ne.s p1', p0'=$s',$t'
// Instructions for
// comparing p1 and p1'

(p1') RTS = s1^s2
(p1) br L1

L1:
```

Can be eliminated
Results from Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>No Fault Tolerance</th>
<th>EDDI+CFCSS</th>
<th>SWIFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time</td>
<td>1.00</td>
<td>1.61</td>
<td>1.41</td>
</tr>
<tr>
<td>Static binary size</td>
<td>1.00</td>
<td>2.83</td>
<td>2.40</td>
</tr>
<tr>
<td>Fault Detection</td>
<td>0</td>
<td>100%</td>
<td>100%</td>
</tr>
</tbody>
</table>

• Optimizations helped reduce the static binary size and improve the performance over EDDI+CFCSS
• No loss in reliability
Undetected Errors

• Opcode changed to store instruction
• Multibit error – both master and shadow get similarly corrupted

References: