# Hardening Development Environment for Embedded Systems

F.Restrepo-Calle[1] A.Martínez-Alvarez[1] F.R.Palomo[2]
S.Cuenca-Asensi[1] M.A.Aguirre[2]

[1]Computer Technology Department, University of Alicante, 03690 Alicante, Spain
[2]Department of Electrical Engineering, University of Sevilla, 41092 Sevilla, Spain

Workshop on Design for Reability (DFR)
HiPEAC 2010, Pisa

# Outline

# Reliability issues ?

- Context $\rightsquigarrow$ *RENASER* project (*Radiation Effects on Semiconductors for Aerospace Systems*)
- Typically, reliability issues in mission critical embedded systems have been mitigated using redundant hardware. This method have become difficult:
  - development of a custom hardened microprocessor can be very costly!
  - electronic components more sensitive to *Single or Multiple Event Effects* induced by radiation

  During recent years. . .

  - Several proposals based on redundant software have been developed, providing detection and error correction capabilities
  - Need of low cost COTS reliable hardware become more evident

# Outline on SIHFT without recovery

*Software implemented hardware fault tolerance* (*SIHFT*) techniques, based on redundancy of instructions achieve better fault detection/correction results

- Rebaudengo et al. proposed a high level instruction redundancy reporting detection of 63% to the program data
- Oh et al. presented the *EDDI* technique (*Error Detection by Duplicated Instructions*) $\rightarrow$ better detection and overhead ...
- and *CFCSS* (*Control-Flow Checking by Software Signatures*) $\rightarrow$ faults on program flow
- Reis et al. *SWIFT* (*Software Implemented Fault Tolerance*).

# Outline on SIHFT with recovery

- Rebaudengo et al. made an approach based on high level instruction redundancy $\rightarrow$ 99.50%
- Reis et al. proposed *SWIFT-R* a technique based on triplication of low level instructions

Results from studied techniques show that low level instruction redundancy offers lower code and data overheads $\rightarrow$ a critical characteristic for embedded systems!!!

# So in this paper, we present...

- A hardening environment able to handle multiple microprocessors made up of ...
  - An extensible multi–target hardening compiler
  - An Instruction Set Simulator (*ISS*) to calculate overheads of time/memory and validate the hardened code
- As a case of study, we have developed a *Picoblaze* back-end to test the environment.
- This enviroment will allow the exploration of hybrid hardware/software solutions to obtain fault tolerant systems.
- Our environment $+$ co–design techniques $\rightarrow$ the calculation of several trade-offs between reliability, performance and device area

# According to the studied *SIHFT* techniques . . .

## . . . what are the main funcionalities a HDE must supply?

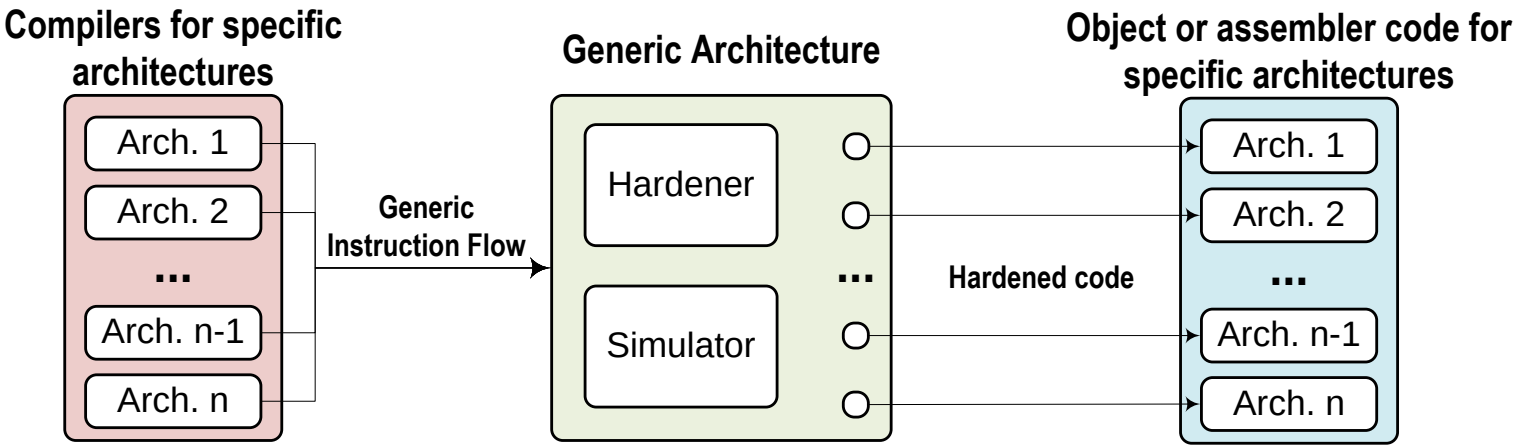
- Insertion of code transformations
- Control flow analysis
- Management of architecture's resources
- Use of *Low Level Redundancy*

# We propose . . .

## . . . a generic architecture to implement hardening tasks:

- Uniform hardening core
- Compatible with many microprocessors of interest
- Able to transform the code (at assembler level)
- Retargetable output

# Our Hardening Development Environment



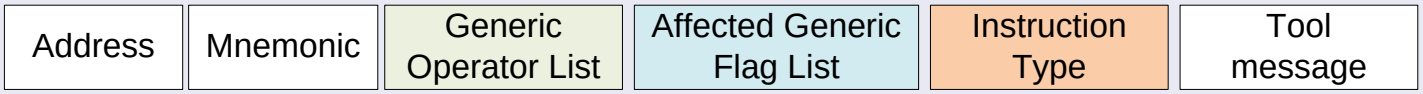
# Generic Architecture in detail

## Three main topics:

- Generic Instruction $\rightsquigarrow$ interoperability at *ISA* level
- Memory Management $\rightsquigarrow$ different set of memories
- Control Flow Management $\rightsquigarrow$ Powerful redundancy

# Generic Architecture in detail

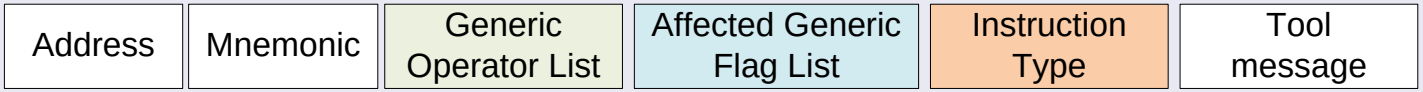
## Generic Instruction (GI) 1/2




- Address $\rightsquigarrow$ address given by the back-end compiler
- Mnemonic $\rightsquigarrow$ original nnemonic
- Generic Operator List
  - Type $\rightsquigarrow$ Register, Literal, Address, Flag
  - Addressing Mode: Absolute, PC-Relative, Register Indirect, Immediate, ...
  - Operator actual name
- Affected Generic Flag List
  - Flag type $\rightsquigarrow$ Z, not Z, C, not C, S, not S, ...
  - Flag actual name

# Generic Architecture in detail

## Generic Instruction (GI) 2/2




- Instruction Type
  - Interrupt
  - Directive
  - Control flow
  - Scalar arithmetic
  - Scalar logic
  - Scalar Input/Output
  - ...
- Tool Message $\rightsquigarrow$ to save a hardening log

# Memory Management

## Memory Management

Due to code insertions it is necessary to:

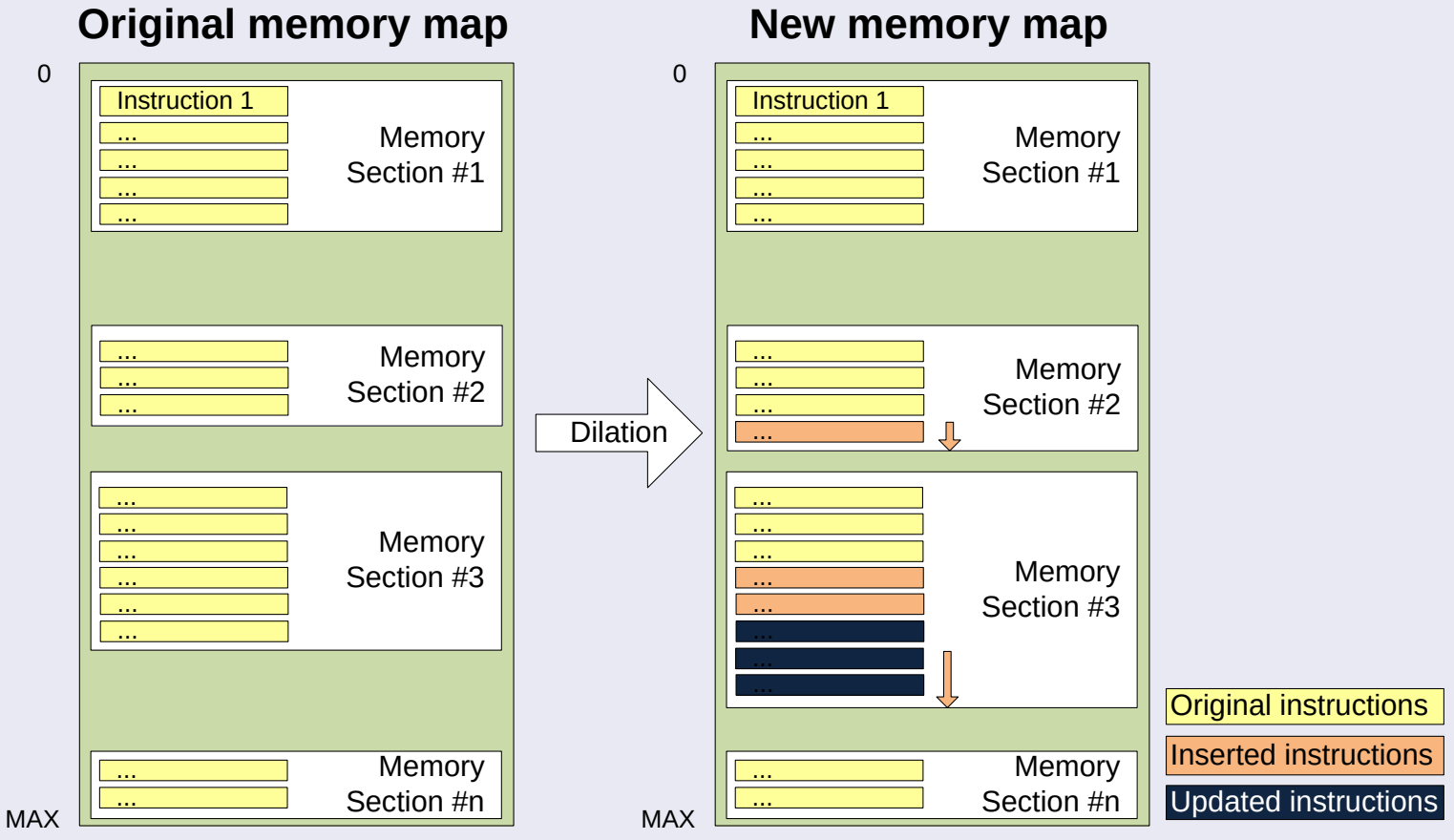
- Identify the memory map to change
- Insert the changes
- Perform a memory update

so the *HDE* offers these three possibilities:

- Dilation
- Displacement
- Reallocation

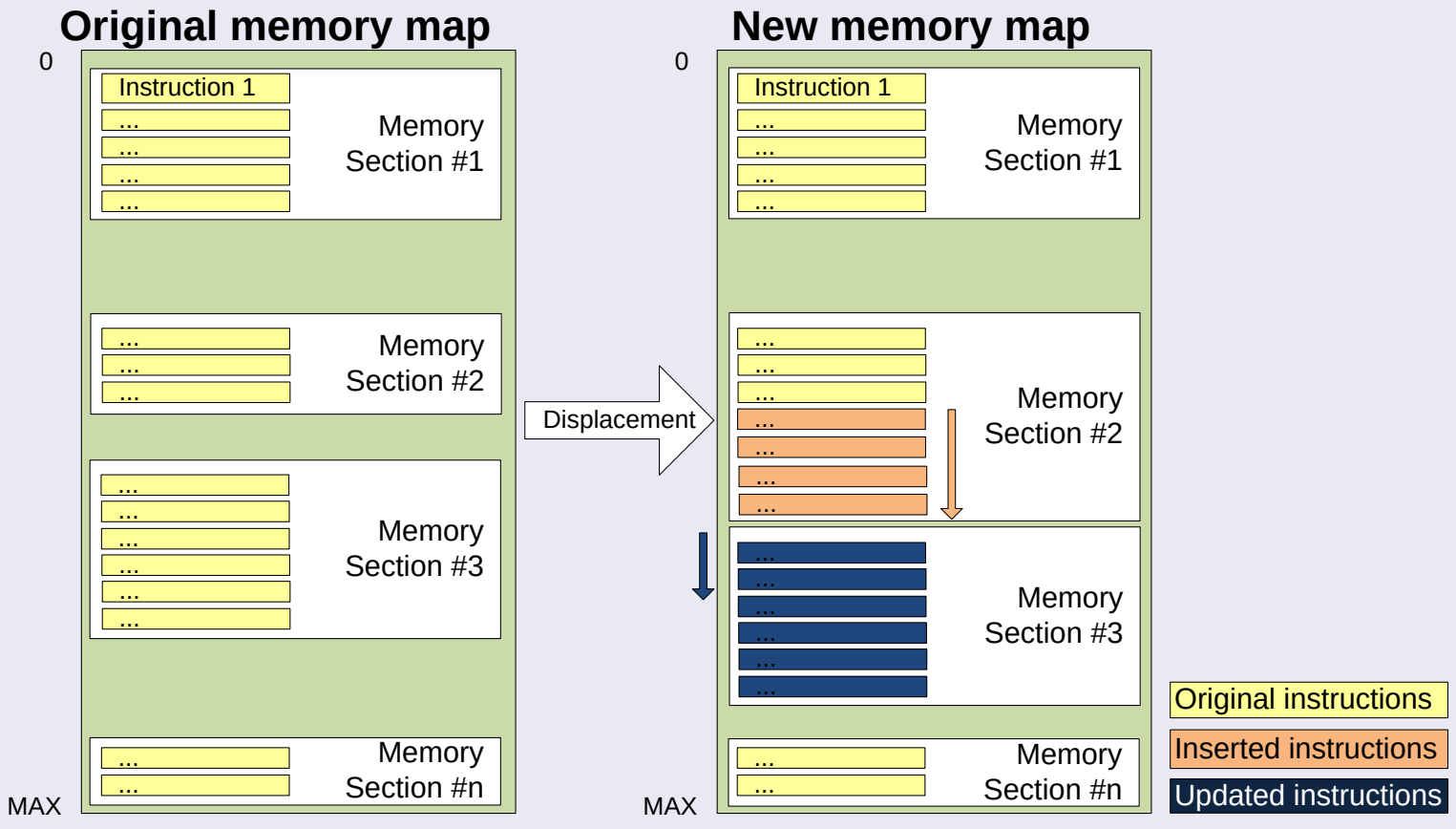
# Memory Management

## Dilation



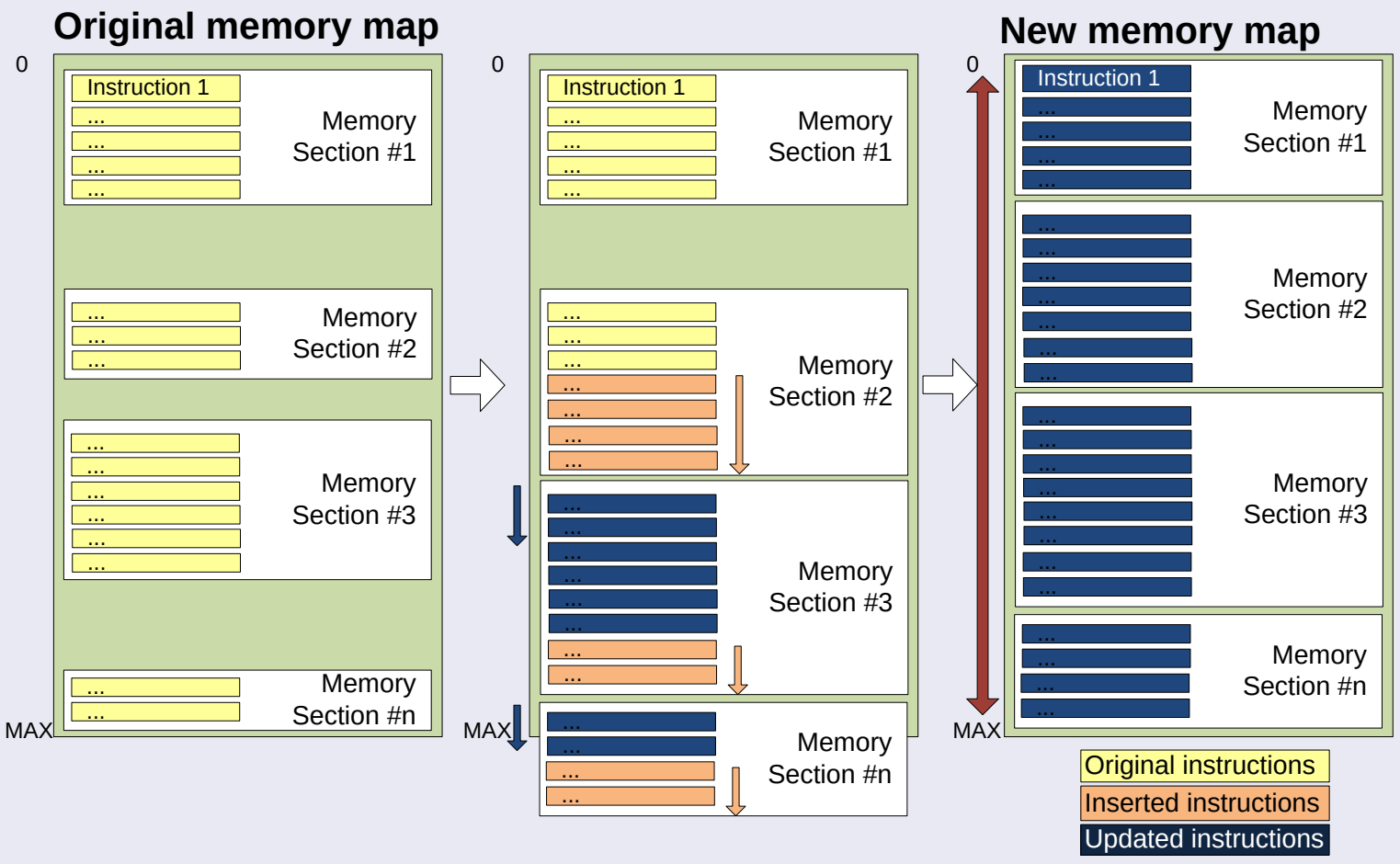
# Memory Management

## Displacement



# Memory Management

## Reallocation



# Flow Control

## Flow Control Graph

*Generic Architecture* $\rightsquigarrow$ flow control of a given *Generic Instruction Flow (GIF)*.

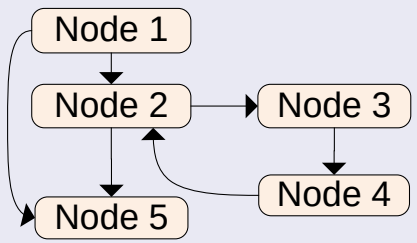
Our Flow Control Graph consists of a set of interconnected blocks conforming a directed graph:

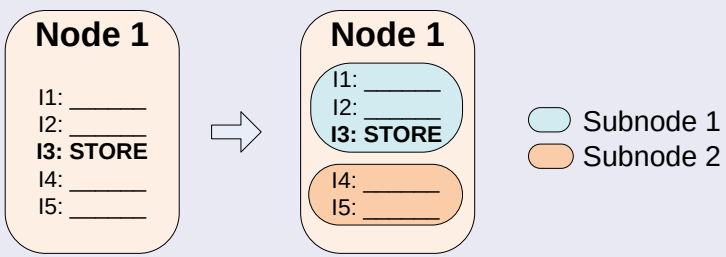
- A basic block: set of instructions sequentially executed
  - without any jump instruction nor function call (except the last instruction)
  - without any instruction being the destionation of a *call* or *jump* instruction except the first one.
  - Each one represents a node in the graph
- Every node is subdivided in a subnode if a *store* instruction is present.

# Flow Control

## Flow Control Graph and Subnodes

Node 1: {I1, I2, I3, I4, I5}
Node 2: {I6, I7, I8}
Node 3: {I9, I10}
Node 4: {I11, I12, I13}
Node 5: {I14}



# Hardening Generic Core

## Hardening Generic Core

Consists of a:

- *Hardening compiler* $\rightsquigarrow$ providing hardening methods:
  - `-method`: What *FT* technique?
  - `-mcpu` : What *CPU*
  - `-replicationRegisterLevel` : Redundancy level $\rightsquigarrow$ `add S0, S1`
  - `-replicationTimes` : Number of copies of each redundant instruction
  - `-voter` : Select the *voter* to be used
  - `-NOlookAheadAvailableRegs` : Enable/Disable advanced register search
- *...*

# Hardening Generic Core

## Hardening Generic Core

- *Instruction Set Simulator (ISS)*
  - Simulates the *GIF*
  - Outputs interesting information (time/memory overheads, statistics, . . . )
  - Checks and validates original and hardened code $\rightsquigarrow$ custom *pragmas* with the expected results
  - Can simulate *Single Event Upsets (SEUs)* faults during the simulation $\rightsquigarrow$ controlled via custom *pragmas* and/or command line options. Effects are classified as:
    - Correct results
    - Incorrect results
    - Hanged
  - Preliminary calculation of the *fault coverage FC*

# checking the hardening...

```
       ...Original Program

load s0, sa
load s1, sb
return
; Output [0]: 1,2,3,4,5
```

```
>>> Simulation file: '../../rtests_hardening/01_bubbleSort.asm'
Check succeeded - Instructions simulated: 228
Instructions in original code: 46
Single simulation result: PASSED
```

```
>>> Simulation Hardened file: '../../rtests_hardening/01_bubbleSort.asm.Hardened
Check succeeded - Instructions simulated: 400
Instructions in hardened code: 95
Hardened simulation result: PASSED
```

```
Overhead code segment    = x 2.07
Overhead time execution  = x 1.75
Dual simulation (original & hardened) result: PASSED
```

# Output from compiler and simulator...

```
--------------------------------------------------------------------------------
Summary Simulation SEU Results
--------------------------------------------------------------------------------

>>> ORIGINAL Simulation file: '../../rtests_hardening/01_bubbleSort.asm'
>>> HARDENED Simulation file: '../../rtests_hardening/01_bubbleSort.asm.Hardened'
------------------------------------------------------------------
                                        ORIGINAL          HARDENED
------------------------------------------------------------------
Total instructions excecuted          = 228               400
Directives                            = 1 (0,44%)         1 (0,25%)
Flow Control instructions             = 45 (19,74%)       71 (17,75%)
Interruption instructions             = 0 (0,00%)         0 (0,00%)
Arithmetic instructions               = 44 (19,30%)       130 (32,50%)
Logical instructions                  = 93 (40,79%)       153 (38,25%)
Shift and Rotate instructions         = 0 (0,00%)         0 (0,00%)
Storage instructions                  = 45 (19,74%)       45 (11,25%)
Input Output instructions             = 0 (0,00%)         0 (0,00%)
------------------------------------------------------------------
Correct Results in spite of SEU (OK) = 554 (55,40%)      680 (68,00%)
Incorrect Results due SEU (FAIL)      = 358 (35,80%)      239 (23,90%)
Processor Hanged due SEU (HANG)       = 88 (8,80%)        81 (8,10%)
Total Executions of the program       = 1000 (100,0%)     1000 (100,0%)
--------------------------------------------------------------------------------
```

## Case study

A compiler back-end for *Picoblaze* generating *GIF* as output. (*KCPSM3* syntax, lexical, syntactical, semantical analisys).
Two different *Triple Modular Redundancy* fault tolerant techniques implemented:

- *TMR1*
  - Identification of nodes and subnodes from the *GIF*
  - Build of the flow control graph
  - Triplication
  - Insertion of majority voters and recovery procedures on:
    - nodes
    - subnodes
    - before an instruction beeing the destination of a jump/call
  - Dynamic insertion of majority voters and recovery procedures if needed.
- *TMR2* Detect and correct faults by computing values twice, and recomputing if discrepancy.

# How looks hardened program with *TMR1* and *TMR2*...

```
load    S1, s0  ; Register copy
load    S2, s0  ; Register copy
add     s0, 3F
add     S1, 3F  ; Redundant inst
add     S2, 3F  ; Redundant inst
compare S0, S1  ; Voter
jump    Z,  00A ; Voter
compare S0, S2  ; Voter
jump    Z,  00A ; Voter
load    S0, S1  ; Recovery
store   s0, 10
```

TMR1

Original version

```
add   s0, 3F
store s0, 10
```

TMR2

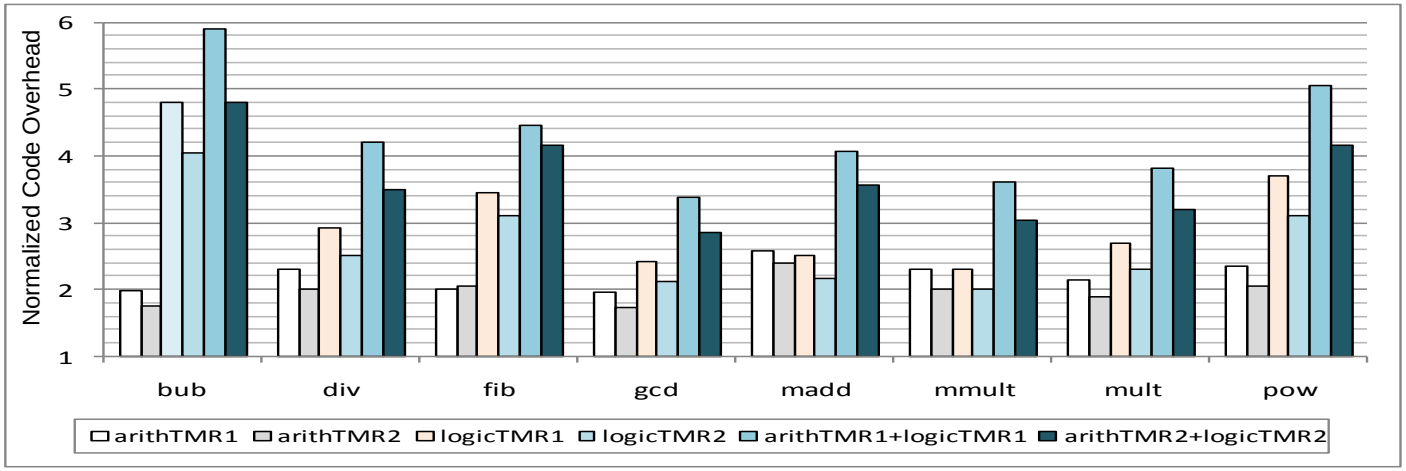
```
load    S1, s0  ; Register copy
load    S2, s0  ; Register copy
add     s0, 3F
add     S1, 3F  ; Redundant inst
compare S6, S2  ; Voter
jump    Z,  008 ; Voter
add     S2, 3F  ; Redundant inst
load    S0, S2  ; Recovery
store   s0, 10
```

## Experiments and Results

Verification of the *HDE*:

- Correctness of the compiler back-end $\rightsquigarrow$ extensive regression test (477 programs)
- Validation of correct funcionality $\rightsquigarrow$ via a `-check-hardening` simulator option
- Evaluation of the implemented hardening technique (overheads and *FC*) $\rightsquigarrow$ custom benchmark using *TMR1* and *TMR2*
  - bubble sort (*bubble*)
  - scalar division (*div*)
  - scalar multiplication (*mult*) and Matrix Multiplication (*mmult*)
  - Fibonacci (*fib*)
  - Greatest Common Divisor (*gcd*)
  - Matrix addition (*madd*)
  - Exponentiation (*pow*)

# ISS results of code and time overheads



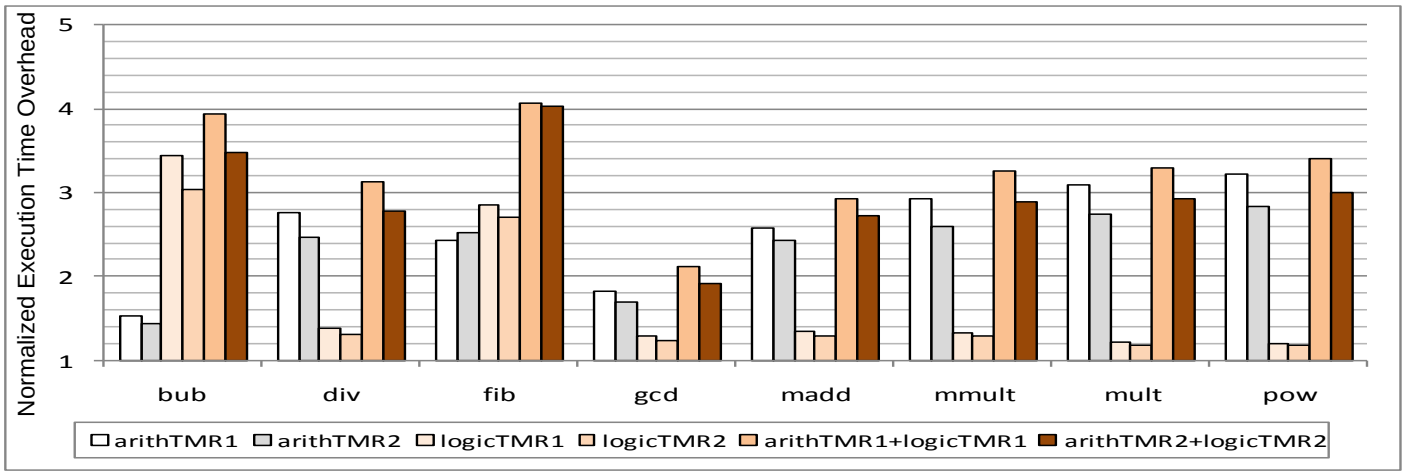




**Figure:** Execution and time overhead

# ISS results of Fault Coverage

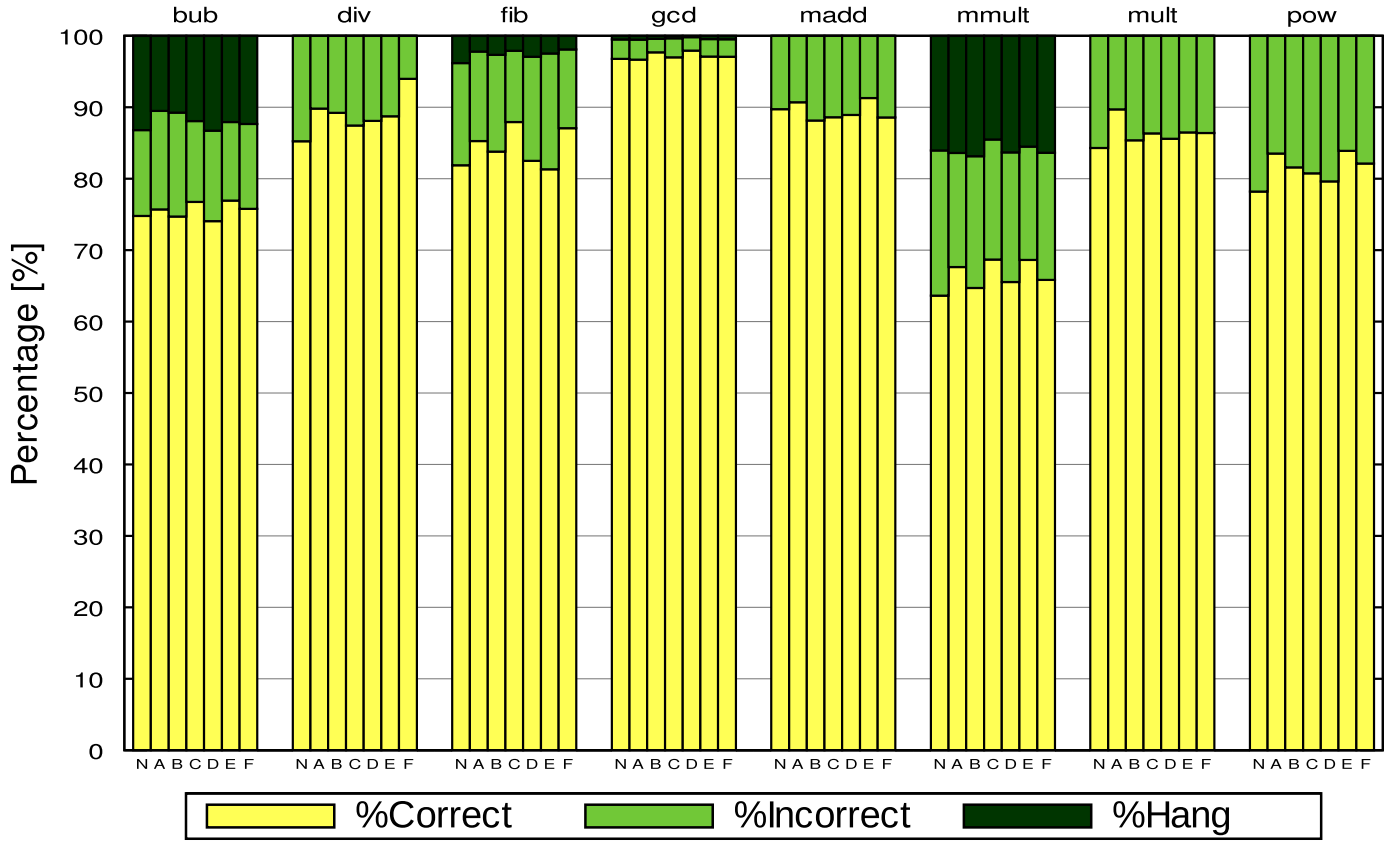



**Figure:** *FC* results for original version(N), *arithTMR1*(A), *arithTMR2*(B), *logicTMR1*(C), *logicTMR2*(D), *arithTMR1+logicTMR1*(E), *arithTMR2+logicTMR2*(F)

## Conclusions and Future Work

- We have presented a Hardening Development Environment for embedded systems.
- A revisión of the main FT techniques was been done
- A *Generic Architecture* and a *Generic Hardening Core* has been introduced
- A case study for *Picoblaze* with 2 implemented hardening strategies has been developed to test the *HDE*
- The overall system provides a low cost automatic solution to incorporate fault tolerant techniques in embedded systems
- The *HDE* will be extended to support *Microblaze* and *Leon3*
- We will use the *FTU* emulation tool to achieve more realistic statistics on *FC*

## Thank you for your attention!

Molte Grazie! Domande?