Software-implemented hardening against soft errors

Massimo VIOLANTE
Politecnico di Torino - Italy

Goal

- To present an overview of soft error mitigation techniques based “mainly” on software approaches

- In this talk we will look at the so-called Software-Implemented Hardware Fault Tolerance (SIHFT) Techniques

Outline

- Introduction: why SIHFT?
- Assumption
- A system-level view of soft errors
- SIHFT target
- SIHFT techniques
- Conclusions

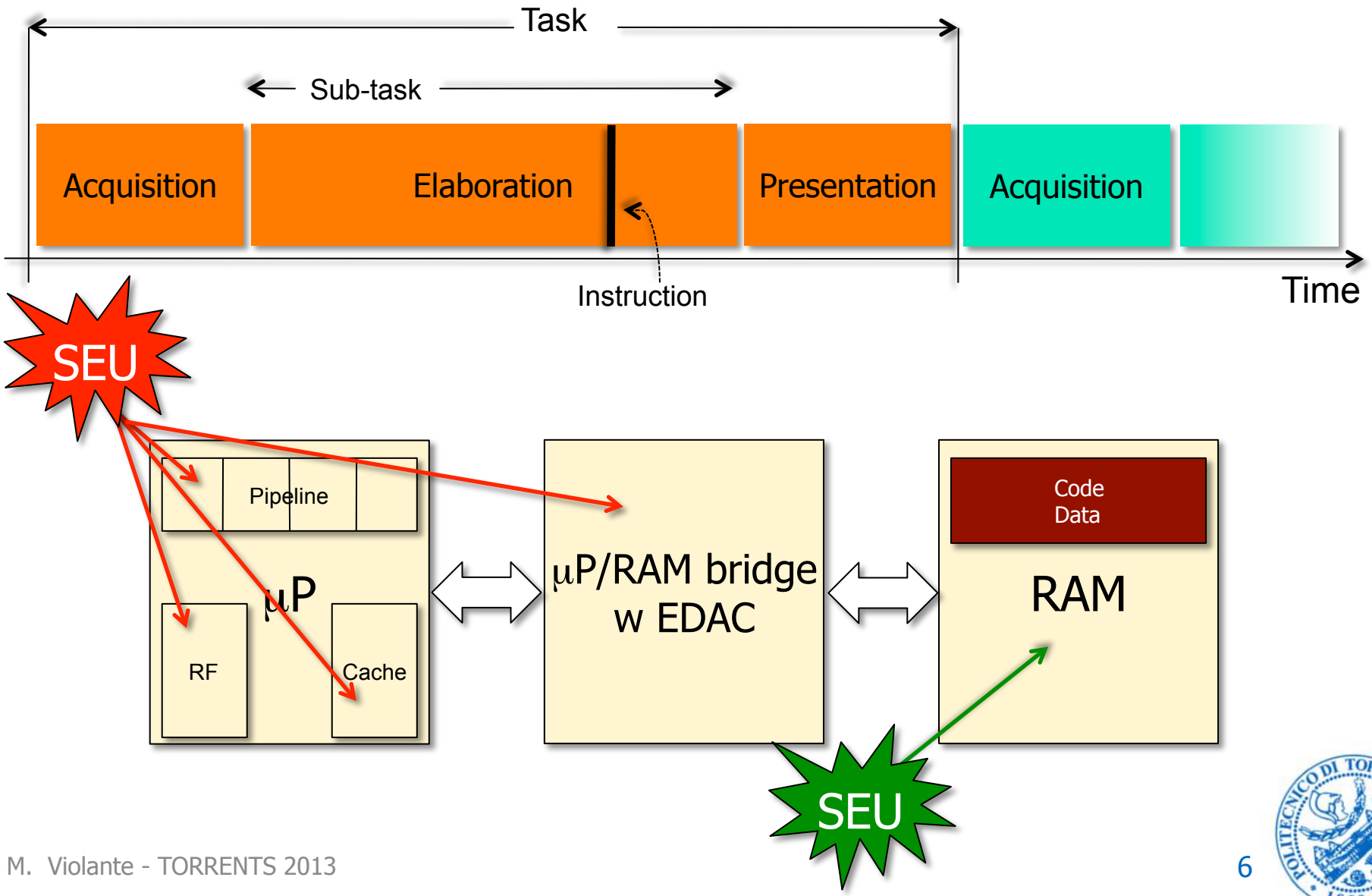
Introduction: why SIHFT?

- COTS components are growing interest due due performance reason, but
- COTS components are not intended for tolerating soft errors, and must be protected
- SIHFT provides soft error mitigation using time and information redundancies in software to avoid hardware redundancy
  - More hardware = more expenses
  - More software ≠ more expenses, iff robust software can be obtained automatically

Assumptions

- Hardware:
  - Microprocessor-based systems
  - COTS components (not rad hard)
  - Single Event Upset (SEU)
- Software:
  - It is correct (no bugs)
  - It is fully available as source code
- Target of the system:
  - Payload processing (one task, no operating system)

Assumptions (cont.)



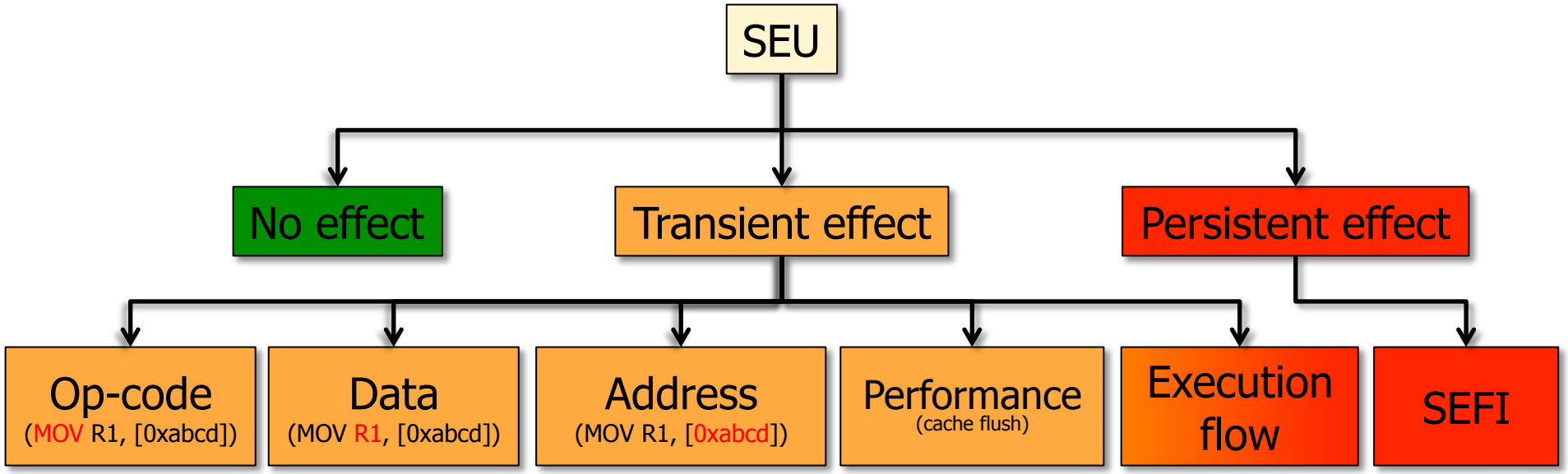
Assumptions (cont.)

- Active redundancy is enforced:
  - The fault is detected first, then
  - The fault is corrected
  - No fault masking (like TMR)

- Correction can be:
  - Forward correction
    - Correct outputs generated from faulty outputs using redundant data
  - Backward correction
    - The computation is repeated

A system-level view of soft errors

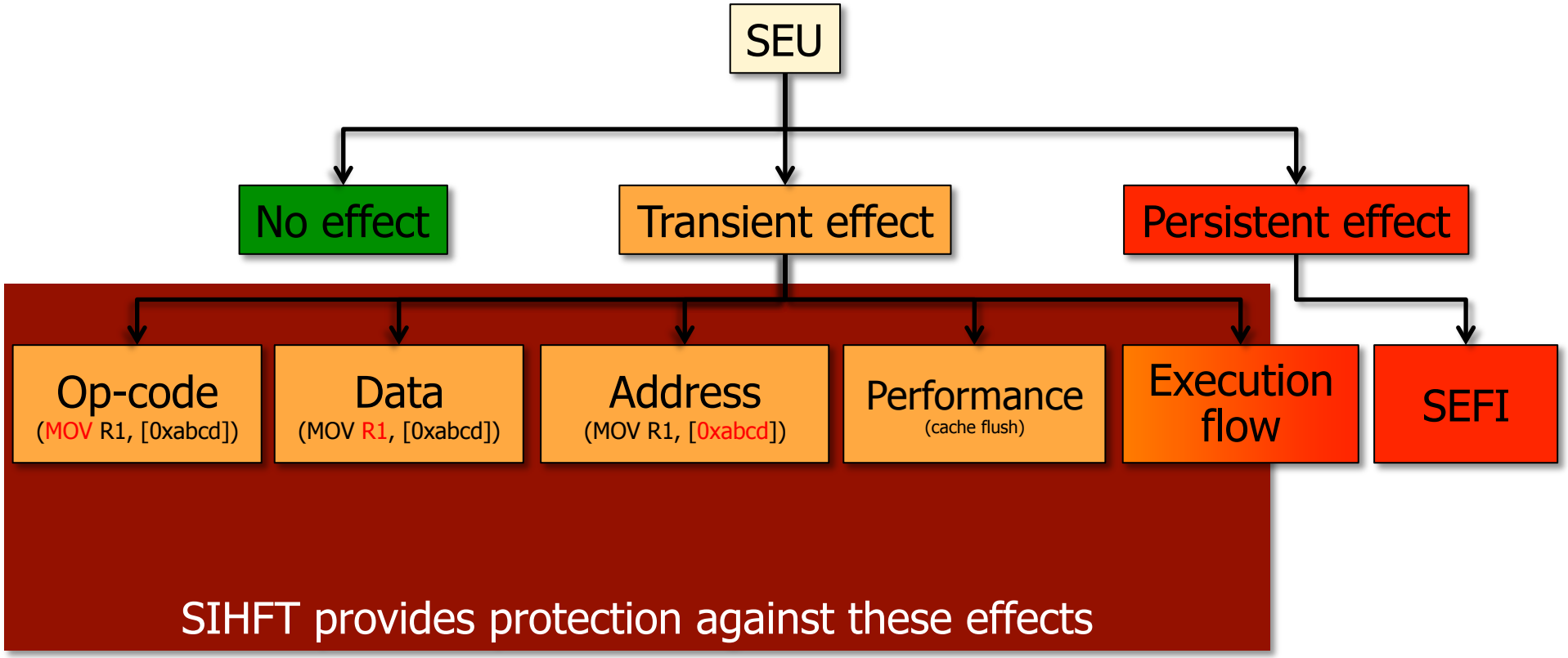
- SEU effects can be modeled according to the modifications they introduce to a running program
  - Model does not depend on the SEU location



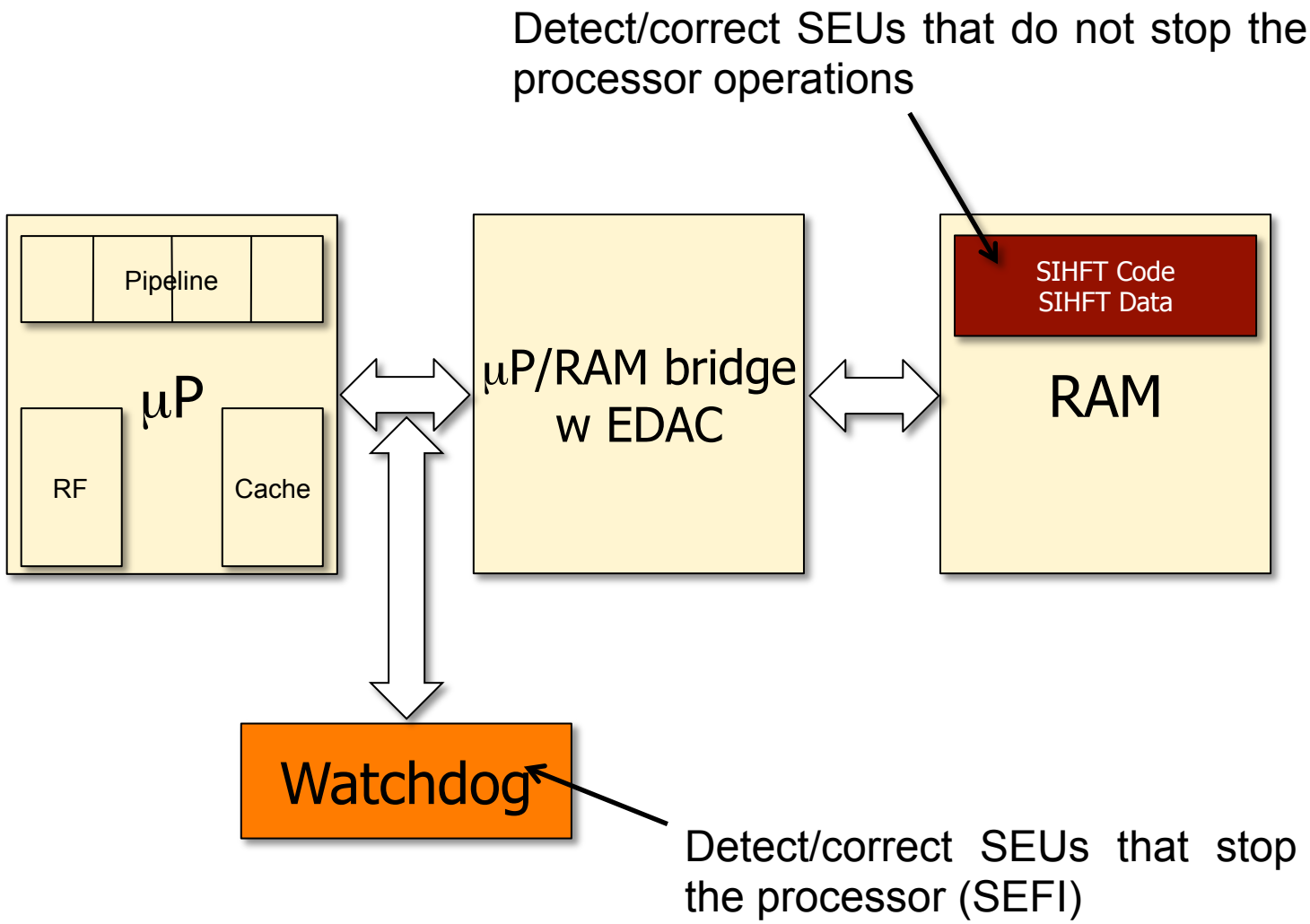
SIHFT target

- SIHFT techniques target SEU effects that does not "hang" the processor (SEFI)
  - Being based on software, SIHFT mandates a processor that is up and running
- To cope with SEFI, complementary techniques are needed
  - Low-cost hardware working in parallel with main processor (e.g., watchdog)

SIHFT target (cont.)



System architecture



SIHFT techniques

- Instruction-level time redundancy
  - Data-oriented techniques
  - Control-oriented techniques
- Task-level time redundancy

SIHFT techniques

- Instruction-level time redundancy
  - Data-oriented techniques
  - Control-oriented techniques
- Task-level time redundancy

Instruction-level time redundancy

- Based on adding instructions (at C level, assembly level, or intermediate level) to:
  - Replicate data
  - Replicate computations
  - Perform consistency checks
- Two classes of techniques:
  - Data-oriented for tackling everything but errors in the execution flow
  - Control-oriented for tackling execution-flow errors
- Exploit backward error correction
- SEFI are not detected

Data-oriented techniques

- The source code of application software is modified by:
  - Replicating each stored data
  - Replicating each operation
  - Checking the consistency of data
- Can be applied at different levels:
  - High-level (e.g., C) programs
  - Assembly programs
  - On compiler-generated intermediate code

EDDI

- Proposed in 2002 by Oh, Shirvani and McCluskey
- Based on duplicating instructions and data at the assembly level
- Reduces execution time slow down by careful instruction scheduling in superscalar architectures

Example

- Original code

```
ADD R3, R1, R2
```

- Modified code

```
ADD R3, R1, R2
ADD R23, R21, R22
BNE R3, R23, error
```


EDDI results

- Detection capability 100%
- Execution time overhead ranges from 72% to 111% for a two-way processor
- Memory overhead is about 100%

The PoliTo approach

- Proposed by Politecnico di Torino group starting from 1999

- Main characteristics:
  - Works on high-level code (e.g., C language)
  - Limited assumptions on addressed faults
  - High independence on hardware
  - High fault coverage
  - Its application can be automated
  - Originally intended for fault detection, only

Basic idea

- A set of rules have been defined
- Rules transform a high-level code into a hardened one
- Rules application can be automated

Rules

- Duplicate every variable
- Execute every operation on the two replicas
- Check for consistency after each read access

Example

- Original code

```
int a, b;
…
b = a+5;
```

- Modified code

```
int a1, b1, a2, b2;
…
b1 = a1+5;      b2 = a2+5;
assert( a1==a2 );
```


Transformation tool

- Transformation rules can be applied automatically
- A prototypical tool automating the application of the rules has been developed
  - Reads a C code
  - Produces a hardened C code

Overhead

- The application of the hardening rules on the whole code on unpipelined or RISC processor systems
  - Increases the size of the code and data areas
    - Factors from 3 to 4 have been observed
  - Reduces the program execution
    - Factors of about 3 have been observed

- Main issue
  - The need for disabling compiler optimizations, or
  - Adoption of ad-hoc compiler

Experimental results

- Several fault injection campaigns have been performed to evaluate the fault detection capabilities of the method
  - On a transputer-based system
  - On an 8051-based system
  - On a LEON processor

Experimental set up

- Some sample benchmark programs have been selected
- Hardened versions have been developed
- Fault injection and radiation experiments have been performed

Fault Classification

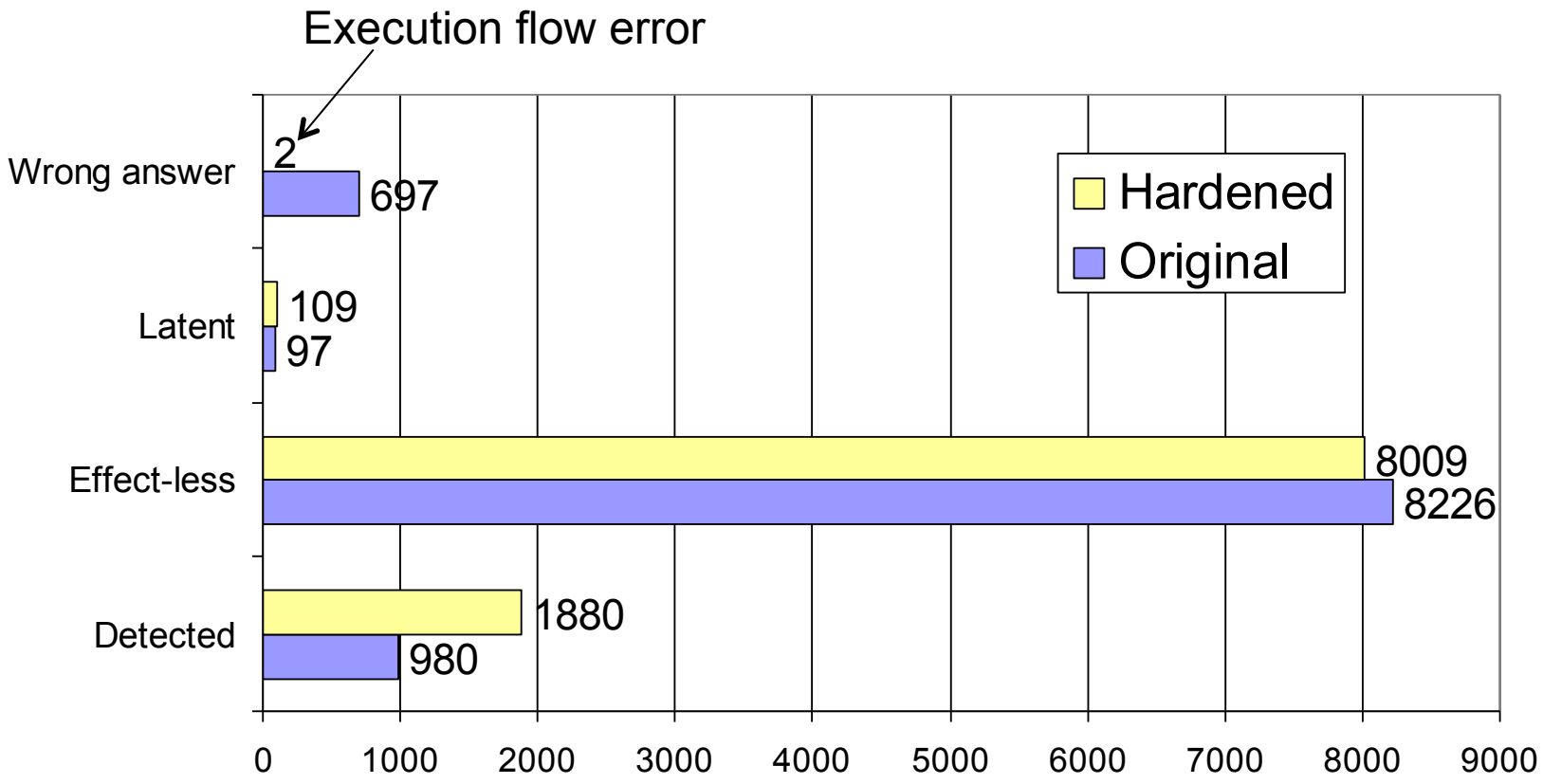
- Effect less
- No Answer
- Latent
- Wrong Answer
  - Undetected incorrect output
- Detected

LEON system: set up

- Injected faults
  - SEUs in the processor memory elements (including register file and pipeline registers)

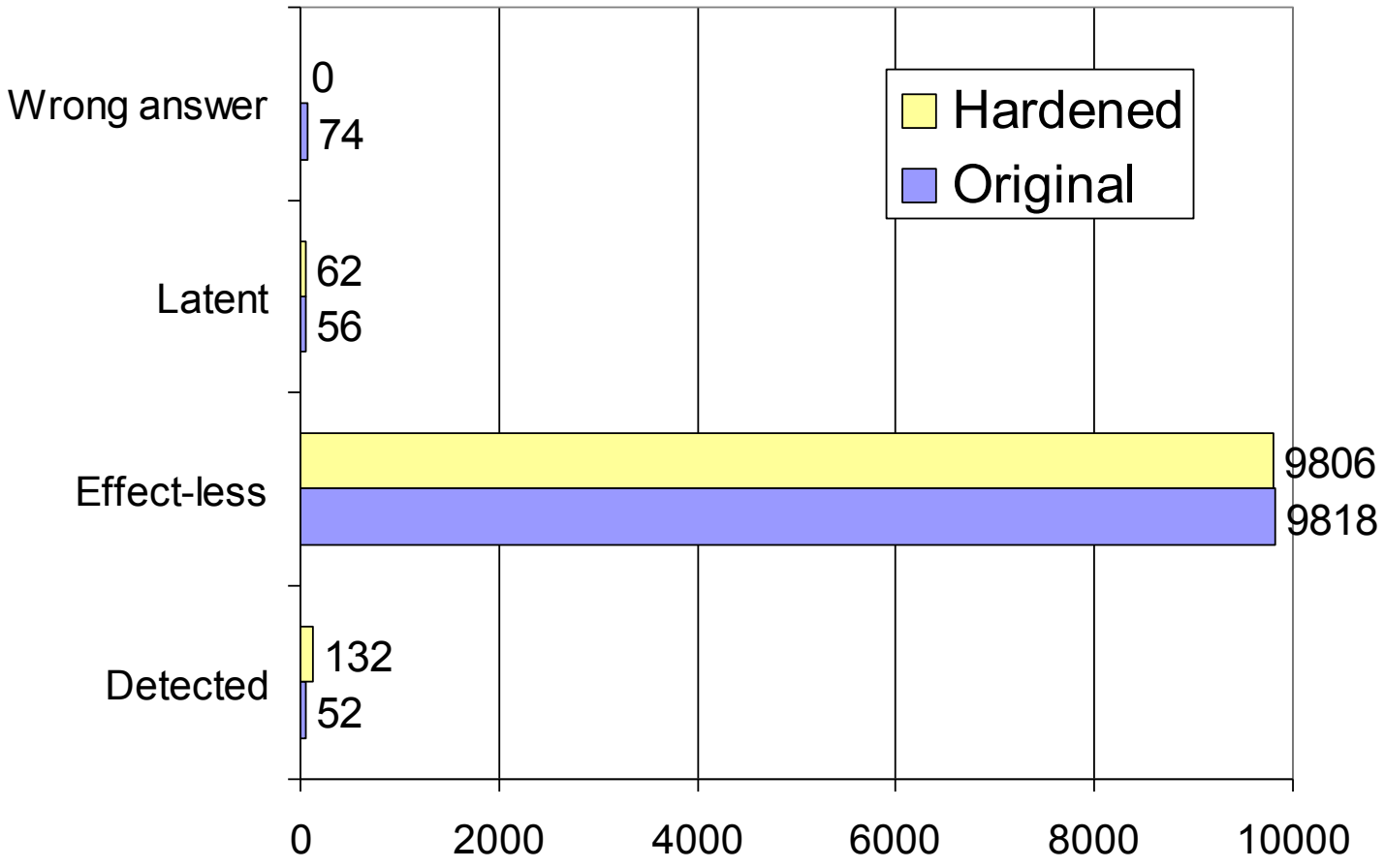
- Emulation-based fault injection has been exploited

SEUs in pipeline registers



10,000 SEUs injected

SEUs in register file



10,000 SEUs injected

Observations (I)

- The method detects faults in:
  - Cache
  - User registers
  - Hidden registers (e.g., in the control unit, or in the pipeline)
  - Combinational logic

Observations (II)

- The method is able to detect:
  - Any kind of fault creating a mismatch between the two replicas of a variable
  - Most of the faults affecting executed instructions

- The method is NOT able to detect:
  - Faults affecting the execution flow
  - Persistent faults

Observations (III)

- The method can be applied flexibly :
  - A subset of rules may be applied (e.g., only duplication rules)
  - A subset of variables may be hardened
  - A subset of the code may be hardened

- In this way, the most suitable trade-off between detection capabilities and overhead can be attained

Extension to fault tolerance

- The rules can be extended to obtain fault tolerance by means of forward error correction
- Only faults affecting data have been targeted

SIHFT techniques

- Instruction-level time redundancy
  - Data-oriented techniques
  - Control-oriented techniques
- Task-level time redundancy

Control-oriented techniques

- Aim at detecting faults changing the execution flow of a program

- All instruction-level techniques are based on:
  - Dividing the program code in basic blocks
  - Building the program graph
  - Checking at run-time the correctness of each transition performed during the program execution

Basic blocks (BBs)

- A basic block (also called *branch free interval*) is a maximal sequence set of consecutive program instructions that, in absence of faults, are always executed together from the first to the last one

- There is no branching instruction in a BB except possibly for the last one

- No instructions within the BB can be the destination of a branch, jump or call instruction, except for the first one, possibly

Program Graph (PG)

- It is a graph where:
  - Vertices are basic blocks
  - Edges are transitions in the execution flow, i.e., branches, call and return instructions, etc.

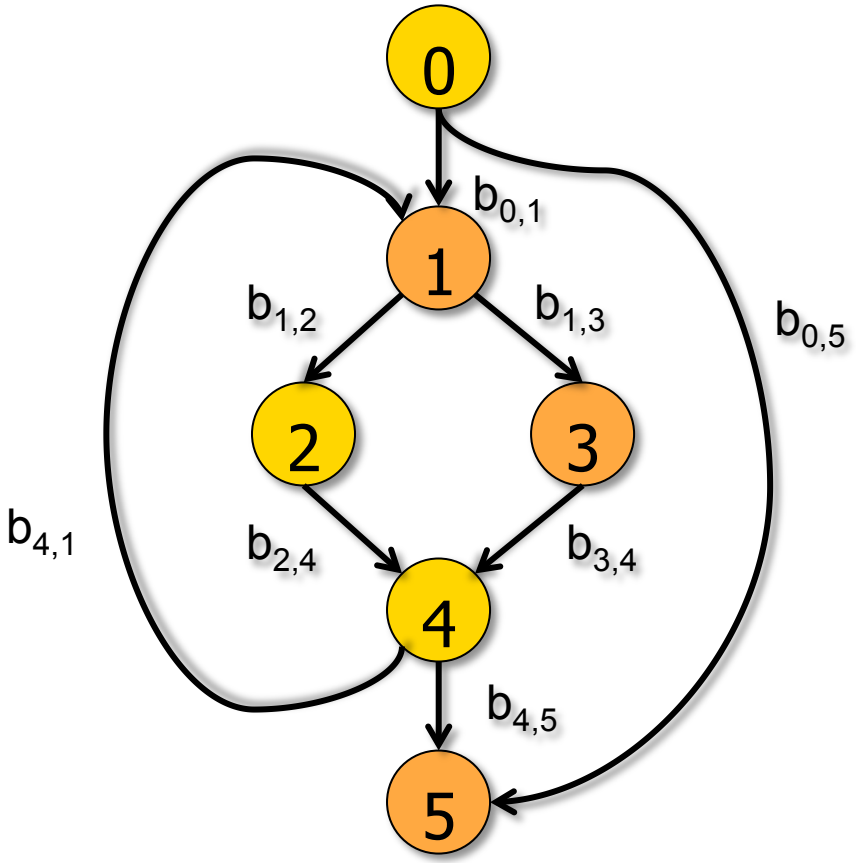
- Let's call V the set of nodes, and B the set of edges in PG

Example

```
0:  i = 0;
    while( i < n ) {
1:   if( a[i] < b[i] )
2:    x[i] = a[i];
3:   else
      x[i] = b[i];
4:   i++; }
5:
```


Example

```
0:  i = 0;
    while( i < n ) {
1:    if( a[i] < b[i] )
2:      x[i] = a[i];
3:    else
        x[i] = b[i];
4:    i++; }
5:
```



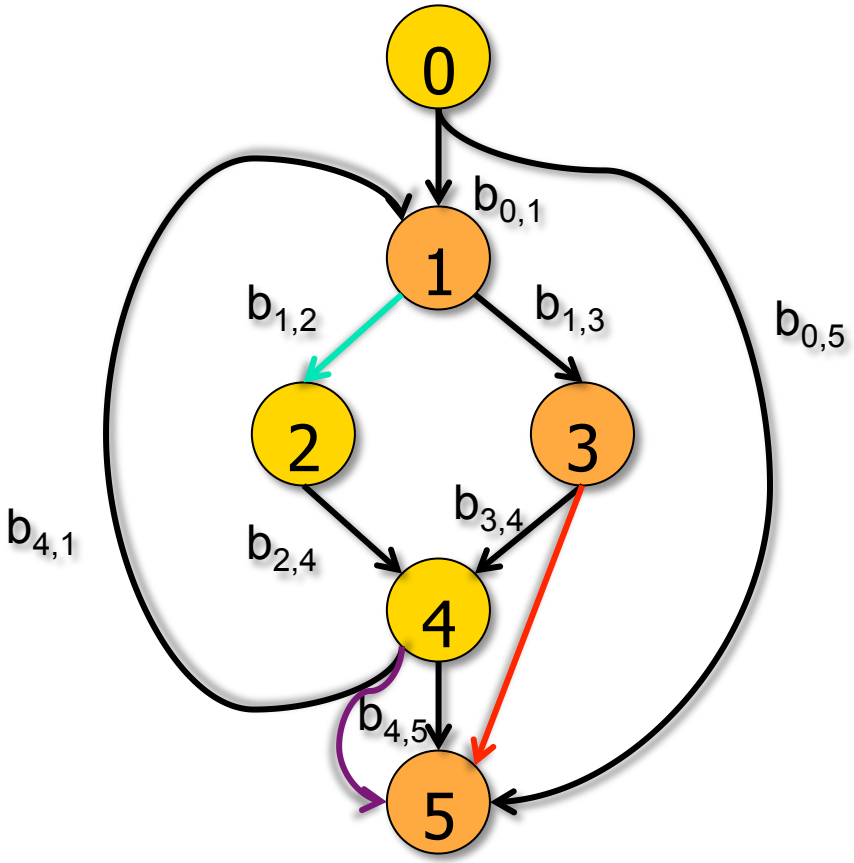
Edge classification

- An edge connecting two vertices in V can be:
  - Legal, if it belongs to B
  - Illegal, if it does not belong to B
  - Wrong, if it is legal, but not consisten with the input data

- Illegal and wrong edges correspond to Control Flow Errors (CFEs)

Examples

Legal
Illegal
Wrong



Control Flow Errors

- They include
  - Wrong and illegal edges in the Program Graph
  - Edges that can not be modeled in the Program Graph (e.g., CFEs causing a wrong branch within a BB)

Control Flow Error causes

- They may include
  - Faults in the Instruction Register
  - Faults in the Program Counter
  - Faults in the stack
  - Faults in the processor decode unit
  - …

Control Flow Checking

- Ideally, it can be implemented by:
  - Building the Program Graph
  - Monitoring the program execution
  - Checking whether the followed control flow is compatible with the Program Graph

Control Flow Checking

- Ideally, it can be implemented by:
  - Building the Program Graph
  - Monitoring the program execution using a block identifier
  - Checking whether the followed control flow is compatible with the Program Graph

When the SIHFT approach is followed, this is done by embedding in the original code some additional instruction

Selected approaches

- ECCA (Abraham et al.)
- CFCSS (McCluskey et al.)
- YACCA (Goloubeva et al.)

Selected approaches

- ECCA (Abraham et al.)
- CF McCluskey et al.)

- A *test assertion* is executed at the beginning of the block and checks if the previous basic block is permissible
- A *set assignment* is executed at the end of the block and updates a unique block identifier

Selected approaches

- ECCA (Abraham et al.)
- CFCSS (McCluskey et al.)
- YAC[illegible]va et al.)

At the beginning of the block:
- A set assignment updates a unique block identifier
- A test assertion checks the correctness of the identifier

Selected approaches

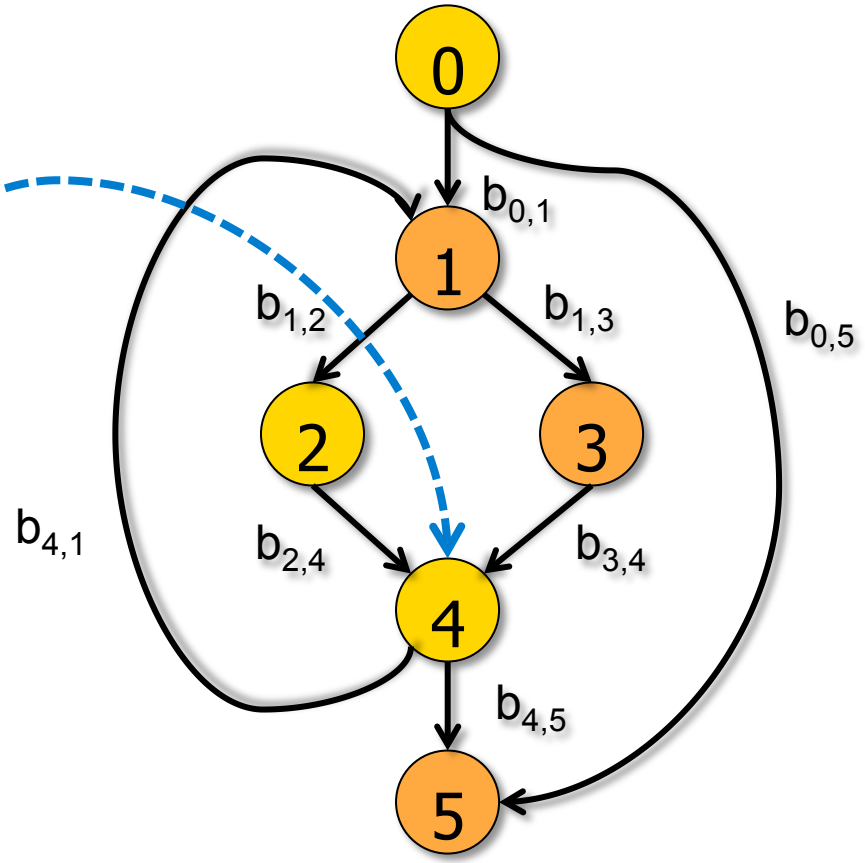
At the beginning AND at the end of each basic block additional instructions are introduced:

- A test assertion controls the current signature and checks if it is permissible
- A set assignment updates the signature

- YACCA (Goloubeva et al.)

Test Assertion

- When the program enters into a basic block, an assertion checks if the transition is legal




```
assert( (code==B2) || (code==B3) )
```

Set Assignment

- Ideally, it could be an assertion such as:

```
code = B_i
```

- Any fault resulting in a jump to this instruction would not be detected

Set Assignment

- Ideally, it could be an assertion such as:

```
code = B_i
```

- Any fault resulting in a jump to this instruction would not be detected

- It is preferrable use a function of the current signature

```
code = (code & M1) ⊕ M2
```


Set Assignment

- Ideally, it could be an assertion such as:

```
code = (code & M1) ⊕ M2
```

- An [illegible] a ju[illegible]on wo[illegible]
- It is preferrable [illegible] a function [illegible]f the current signature

It represents a *constant mask* depending on the signatures of the *preceding* blocks

It represents a *constant mask* depending on the signatures of the *current* and the *preceding* blocks

Set Assignment

- Ideally, it could be an assignment such as:

Both M1 and M2 can be computed for each assertion at compile time

- Any fault in this instruction would not be detected

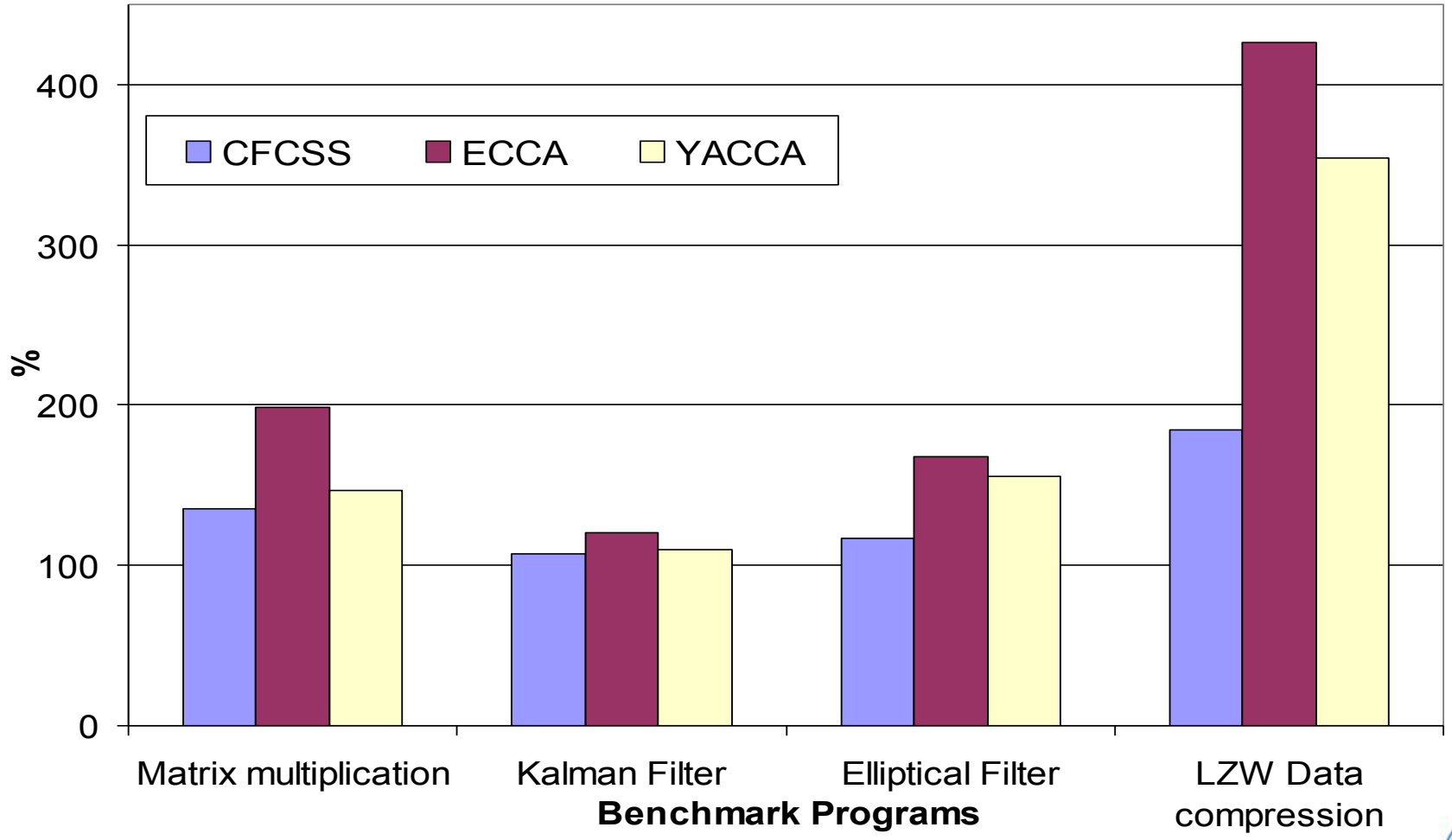
- It is preferrable use a function of the current signature

```
code = (code & M1) ⊕ M2
```

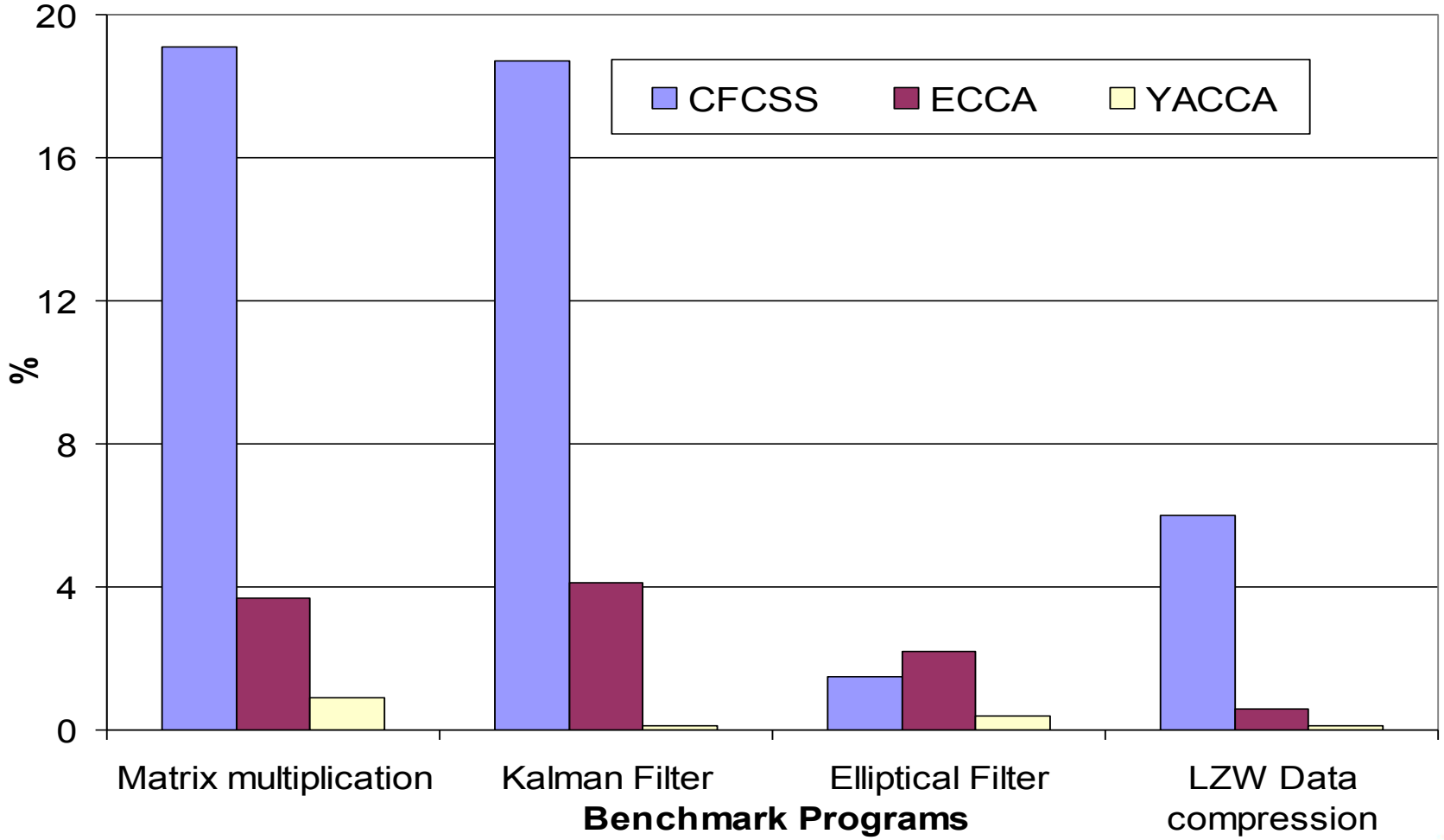
Experimental environment

- Simple benchmark programs (in C language)
- Target system based on a SPARC microprocessor
- Code transformations implemented by an automatic tool
- Simulation-based Fault Injection environment

Performance slow-down



Wrong answers

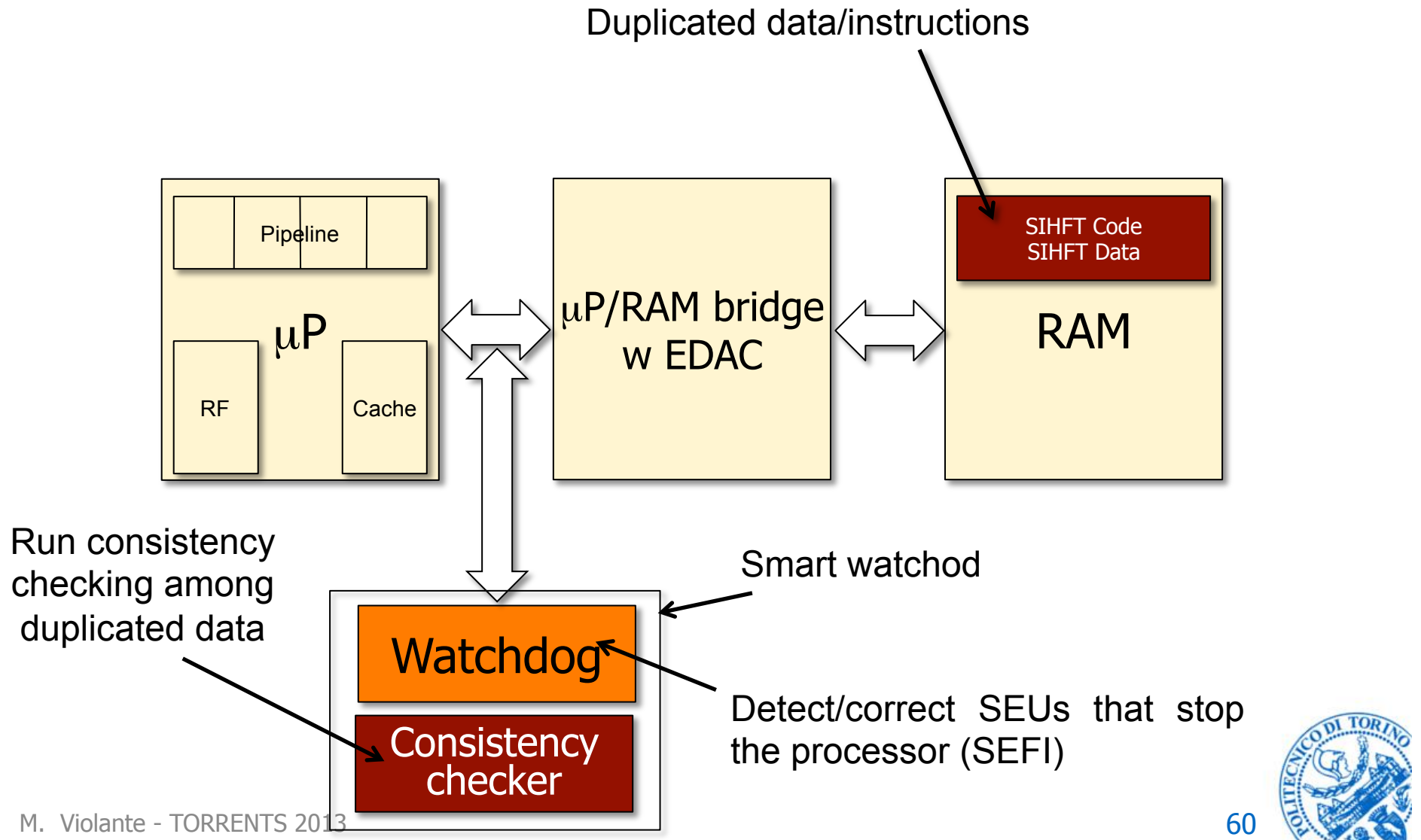


Coping with performance issues

- Performance overhead due to three factors:
  - Need for disabling compiler optimization → hard to avoid
    - SIHFT methods assumes a certain ordering of instructions
  - Duplication of operations → impossible to avoid
    - SIHFT exploits time redundancy
  - Execution of consistency checks → avoidable?

```
int a1, b1, a2, b2;
…
b1 = a1+5;      b2 = a2+5;
assert( a1==a2 );
```


Improved System architecture



Hardware consistency checker

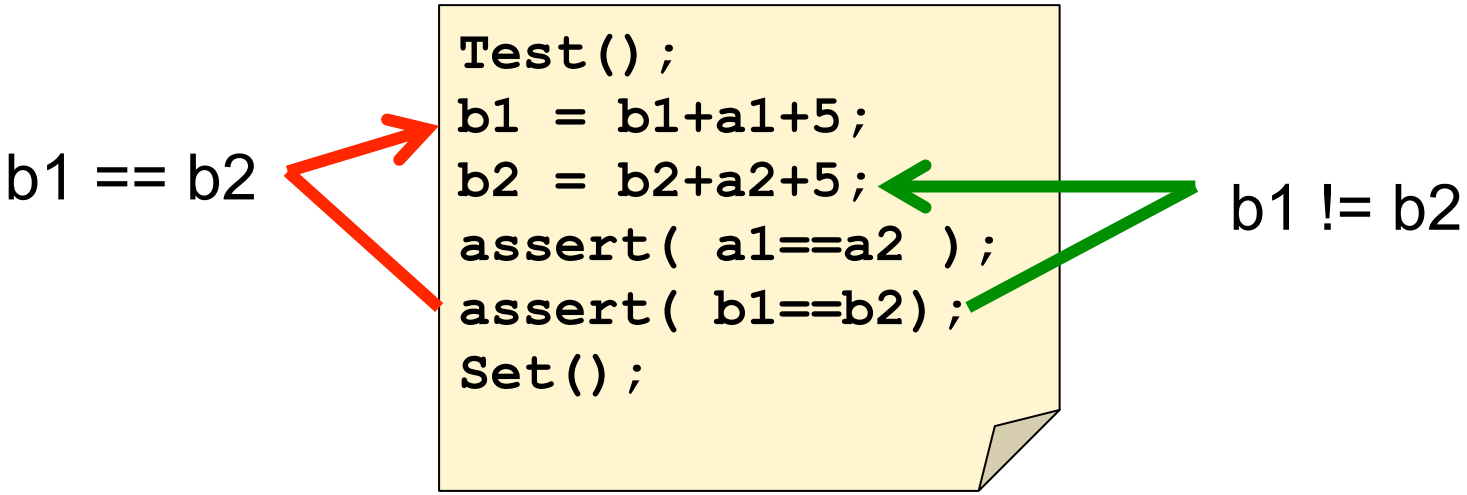
- Exploits specific ordering of duplicated data obtained when compiling the SIHFT software
  - ADX(a1) = ADX(a0)+OFFSET
- "Snoops" read/write cycles and stores (ADX, value) pairs in a context addressable memory (CAM)
  - ADX is used as key for the CAM
- Based on ADX value, it identifies replicas of the same data and runs consistency checks

Hardware consistency checker

- Same fault coverage
- Performance overhead ~2x (compared to 3x-4x)
- Area overhead <10%

Wrap-up

- Instruction-level SIHFT detects:
  - 100% SEE affecting data (no matter where it is stored)
  - >98% SEE affecting control flow
    - SEE provoking jumps within the basic block can escape



- Costs: >2x performance, < 10% area

SIHFT techniques

- Instruction-level time redundancy
  - Data-oriented techniques
  - Control-oriented techniques

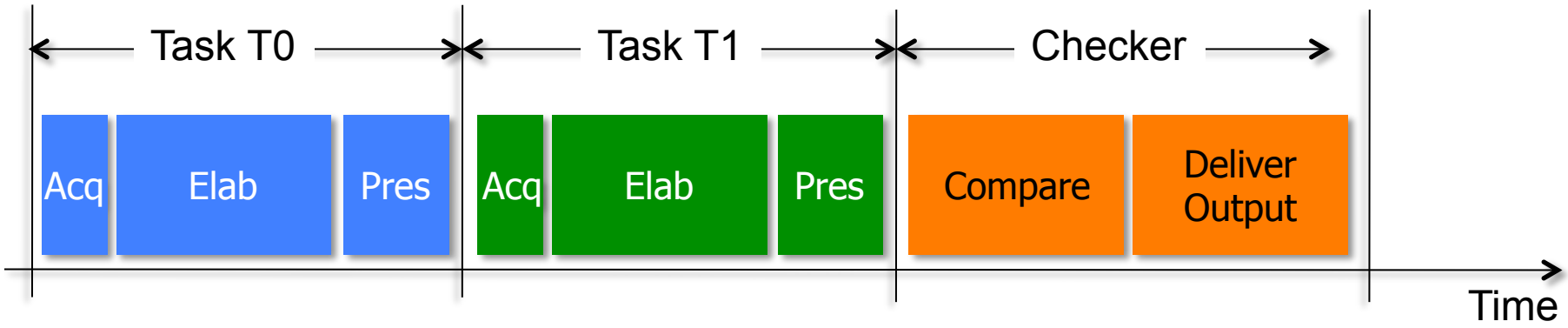
- Task-level time redundancy

Task-level redundancy

- Idea:
  - Time redundancy is applied at task level
  - Consistency checks done over task outputs
- Lower performance overhead:
  - Fewer consistency checks
  - No need for disabling compiler optimizations
  - Applicable also when source code not available (e.g., libraries)
- Require substantial "manual" design
  - No automation

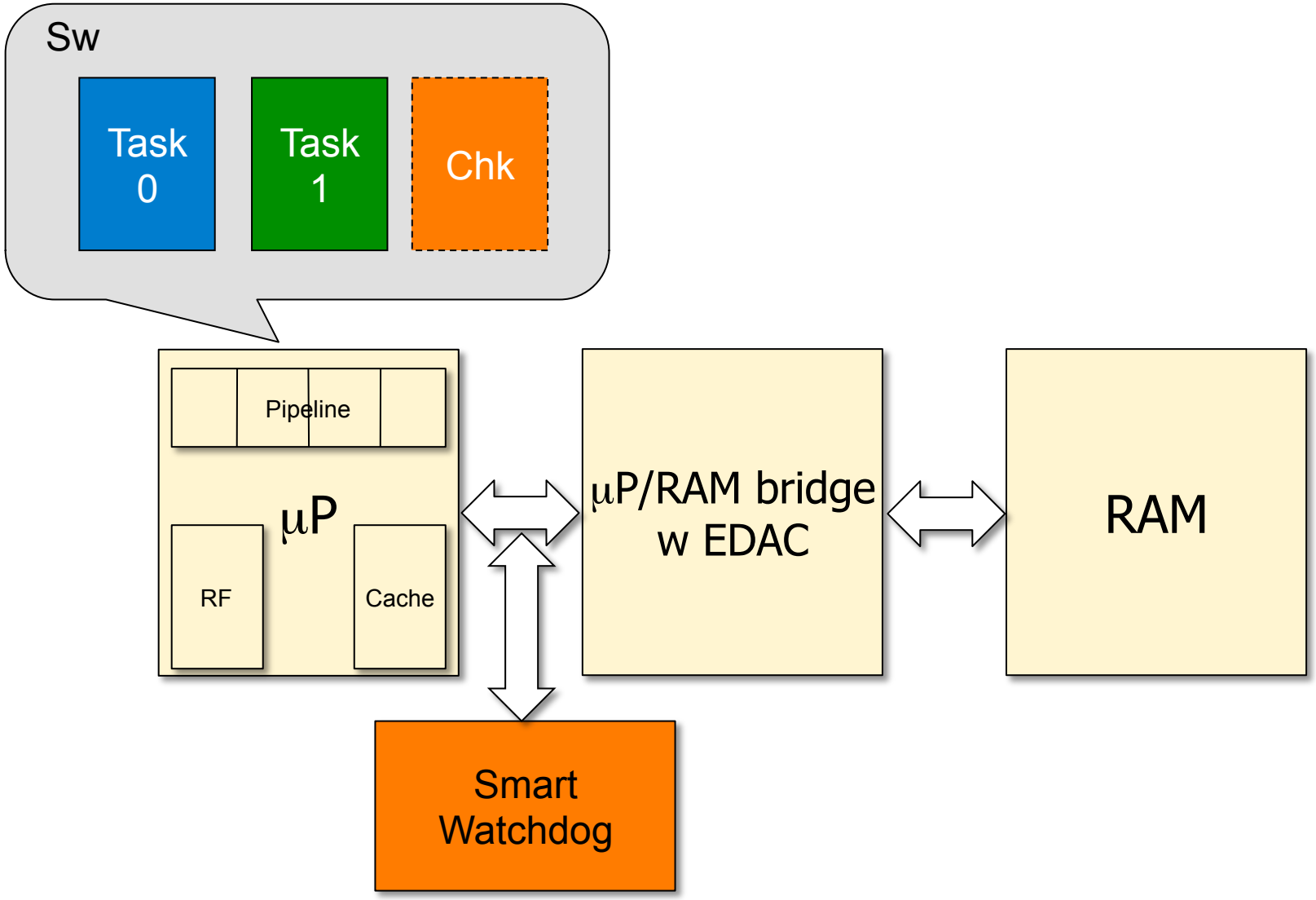
Task-level redundancy in a nutshell

- Run two instances (T0, T1) of the same task

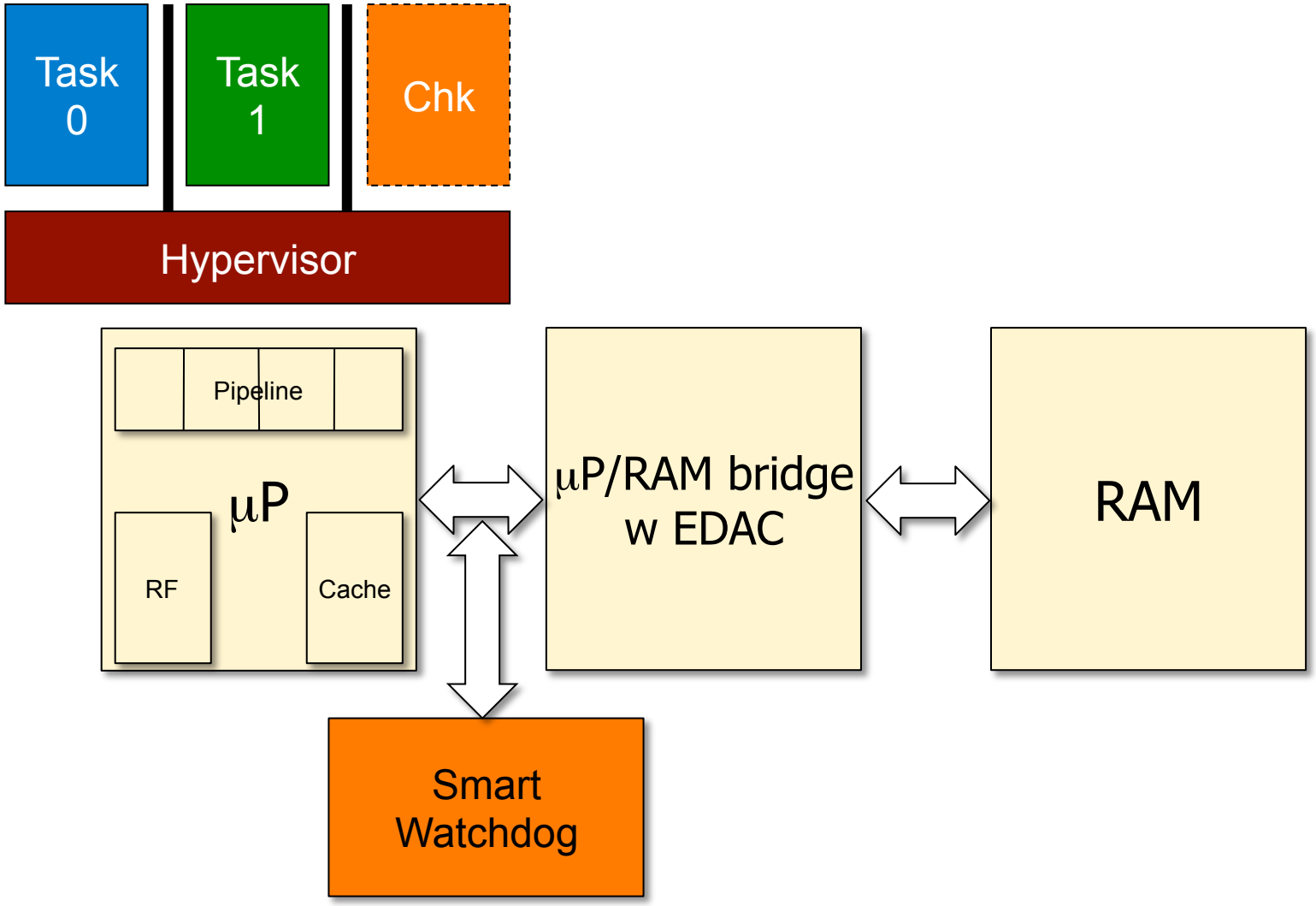


- The executions are segregated in their own address space
  - Task T0 should never corrupt Task T1
- "Independent" consistency check detects errors
  - Run by custom sw or hw

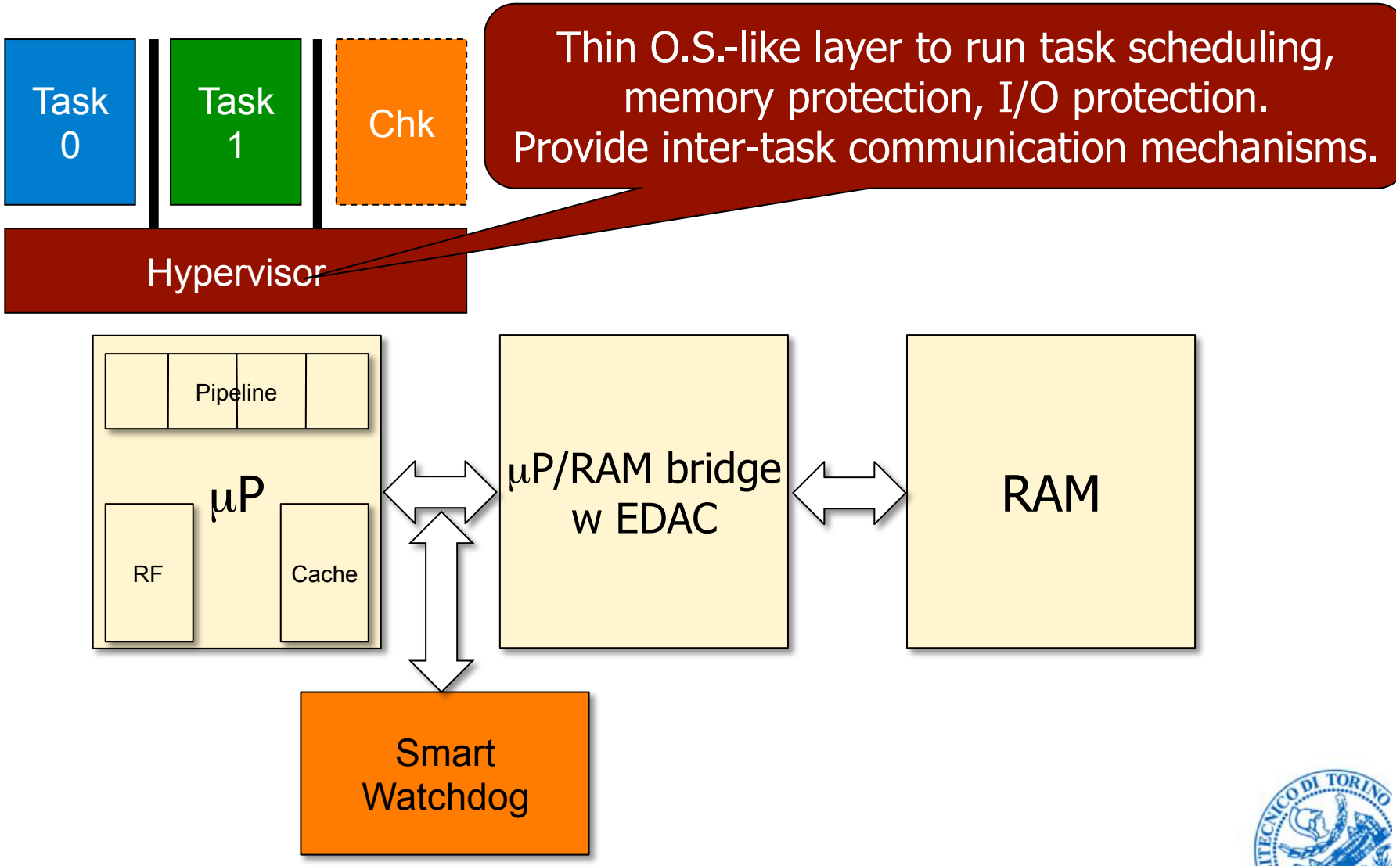
General task-level redundancy



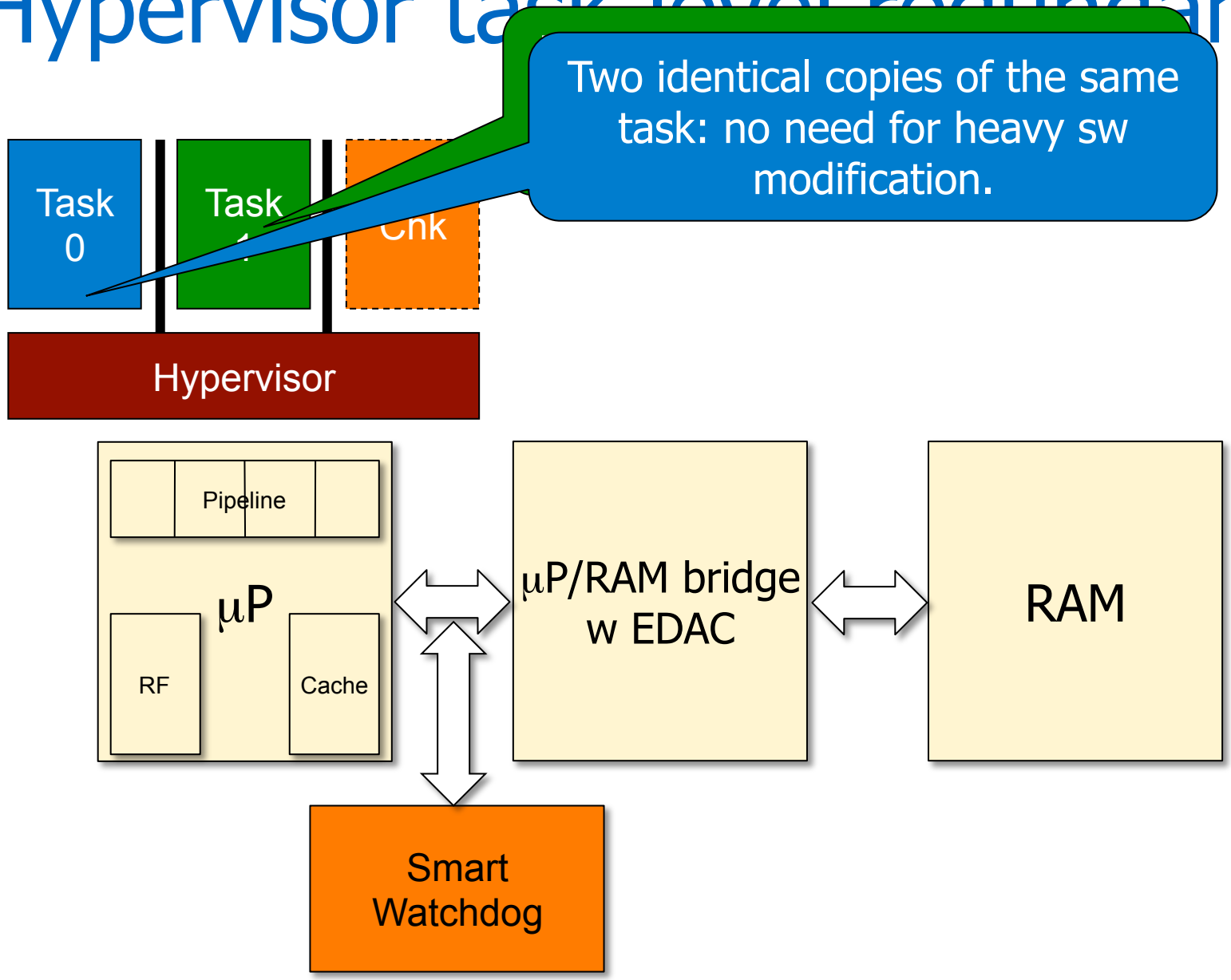
Hypervisor task-level redundancy



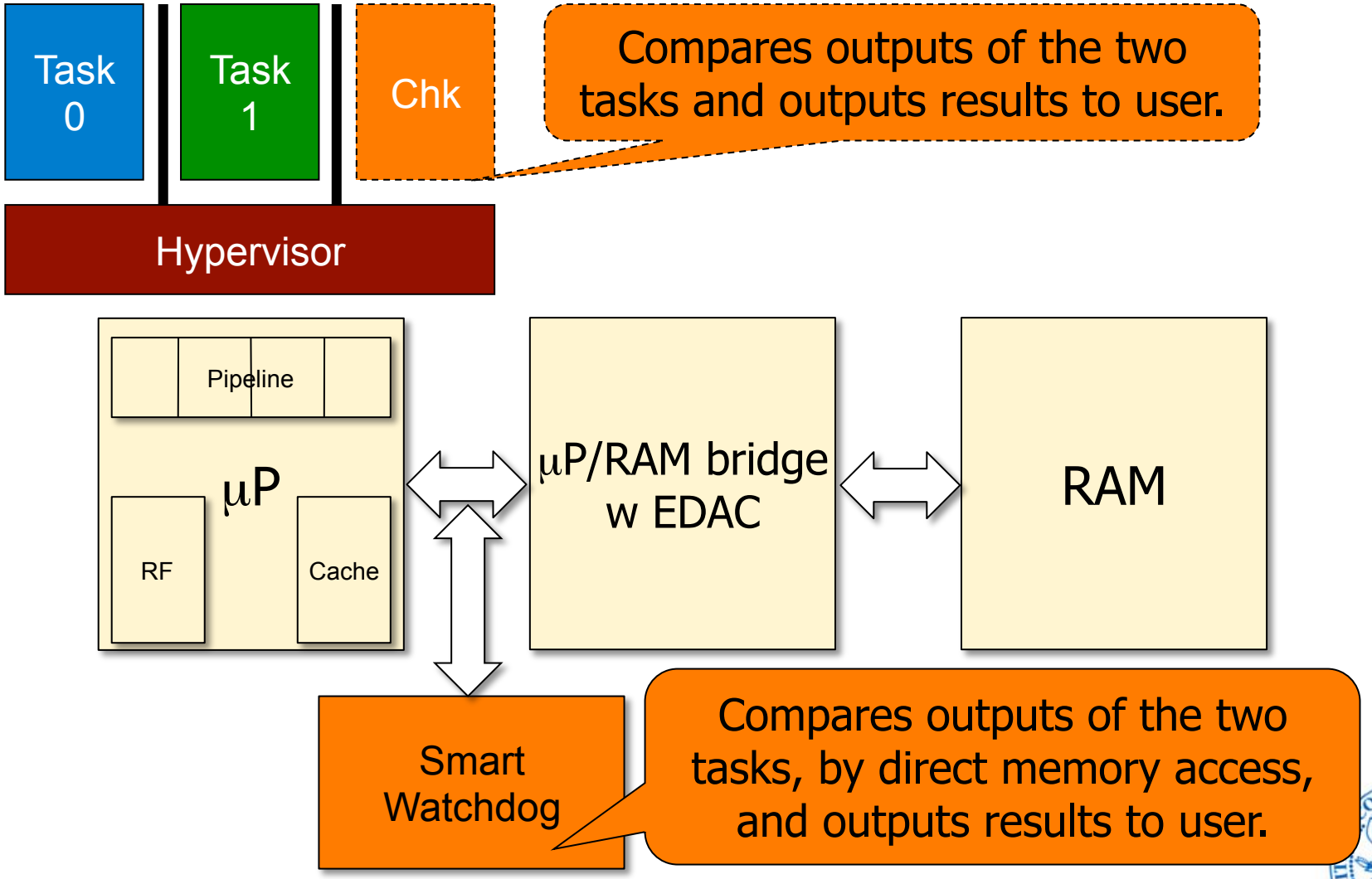
Hypervisor task-level redundancy



Hypervisor task level redundancy

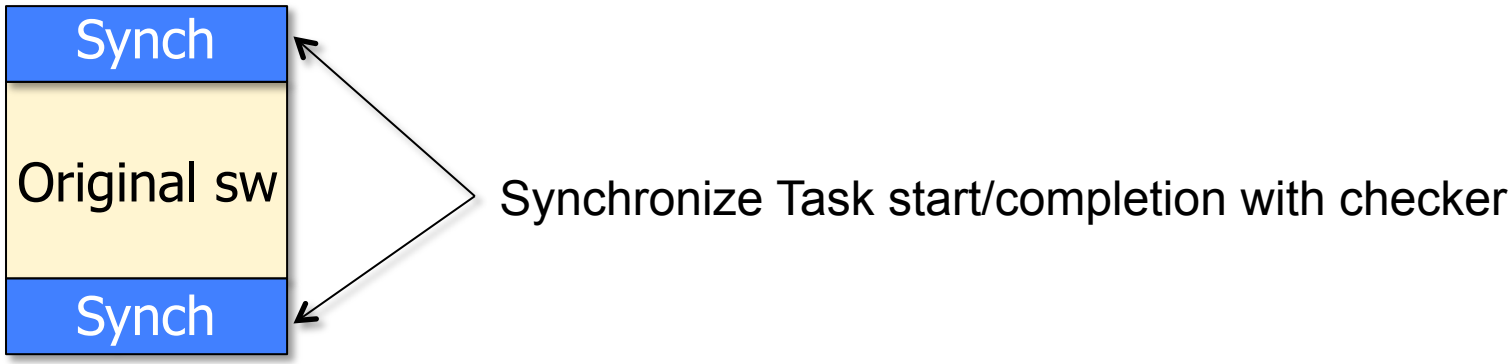


Hypervisor task-level redundancy



Advantages/Limitations

- Advantages:
  - Limited development effort (most of the complexity dealt by hypervisor, automated)

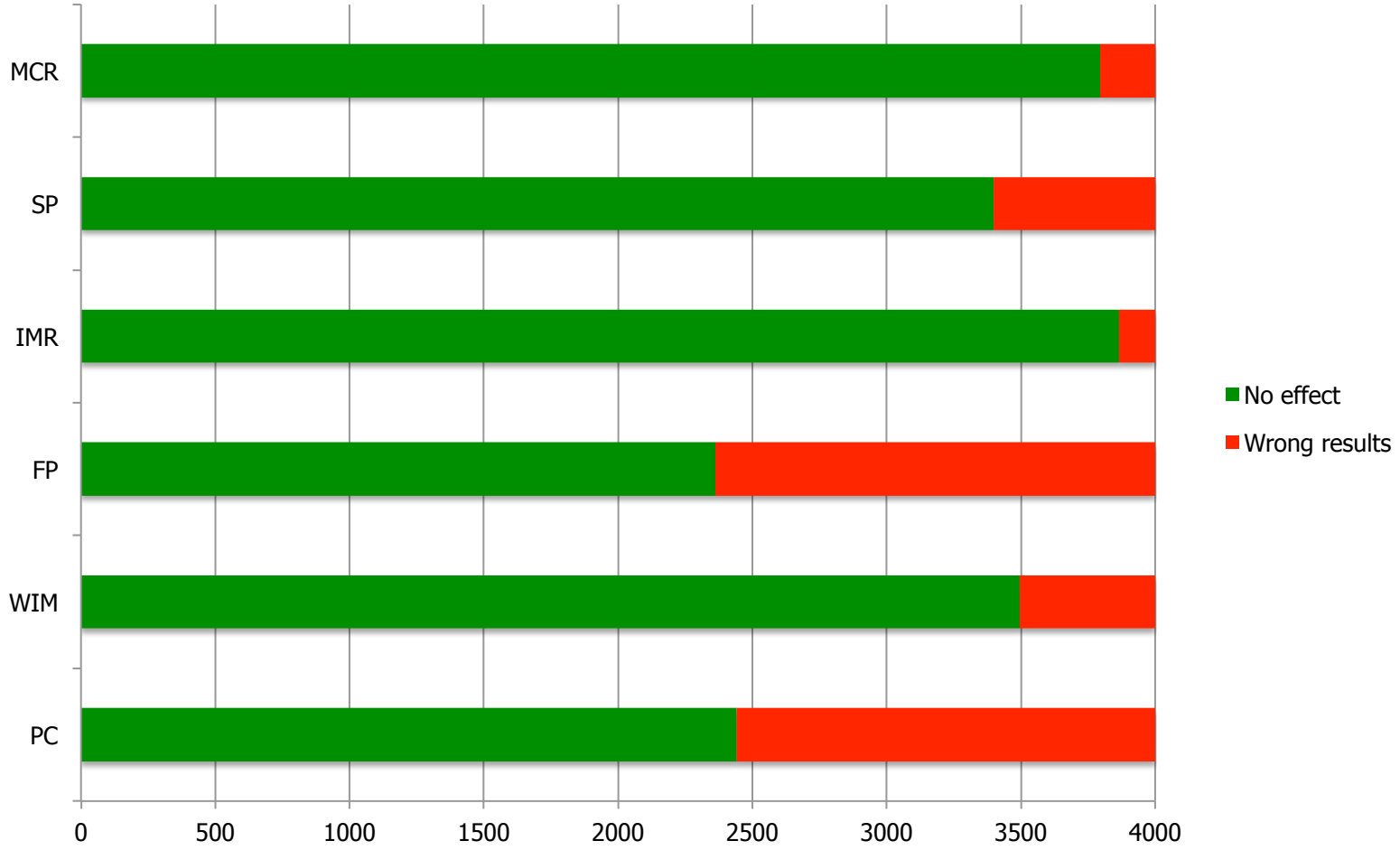


  - Easily portable/scalable (single- vs multi-core)
- Limitations:
  - Hypervisor is not replicated → single point of failure
  - Need for hypervisor for the adopted processor

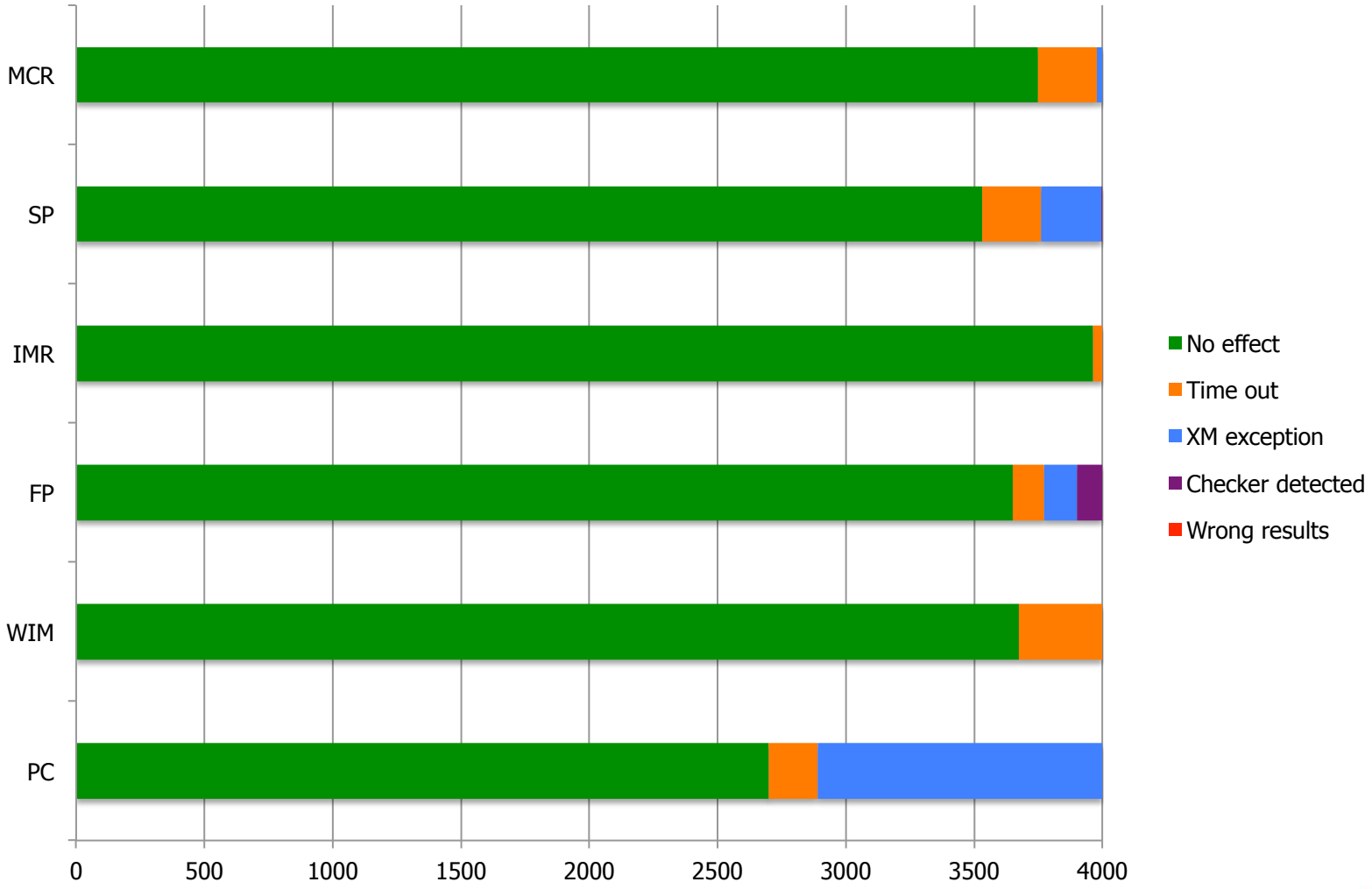
An example

- LEON3 processor from Aeroflex Gaisler
- GR-XC3S-1500 board from Pender
- Xtratum from Universidad Politécnica de Valencia (Spain)
- Smart watchdog coded in VHDL
  - DMA controller (AMBA bus master)
  - Watchdog timer for SEFI detection
  - 3% are overhead w.r.t. the LEON3 core
- Hypervisor vulnerability < 3% execution time

Original system



Hardened system



Conclusions

- SIHFT is viable solution for SEE mitigation in COTS processor (especially for payload processing)
- Instruction-level techniques seems best fit for DSP-based applications where ad-hoc compiler can be used
  - Plenty of execution slots can run duplicated instruction in parallel
- Task-level techniques seems best fit for general-purpose processors
  - Hypervisor can greatly help