IEEE/IFIP DSN 2020 Conference – June 29, 2020

Tutorial #1

Cross-Layer Soft-Error Resilience Analysis of Computing Systems

Alberto Bosio
École Centrale de Lyon, France

Stefano Di Carlo and **Alessandro Savino**
Politecnico di Torino, Italy

Dimitris Gizopoulos
University of Athens, Greece

Ramon Canal
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center, Spain

IEEE/IFIP DSN 2020 Conference – June 29, 2020

Part #4

Software Level Resilience Assessment

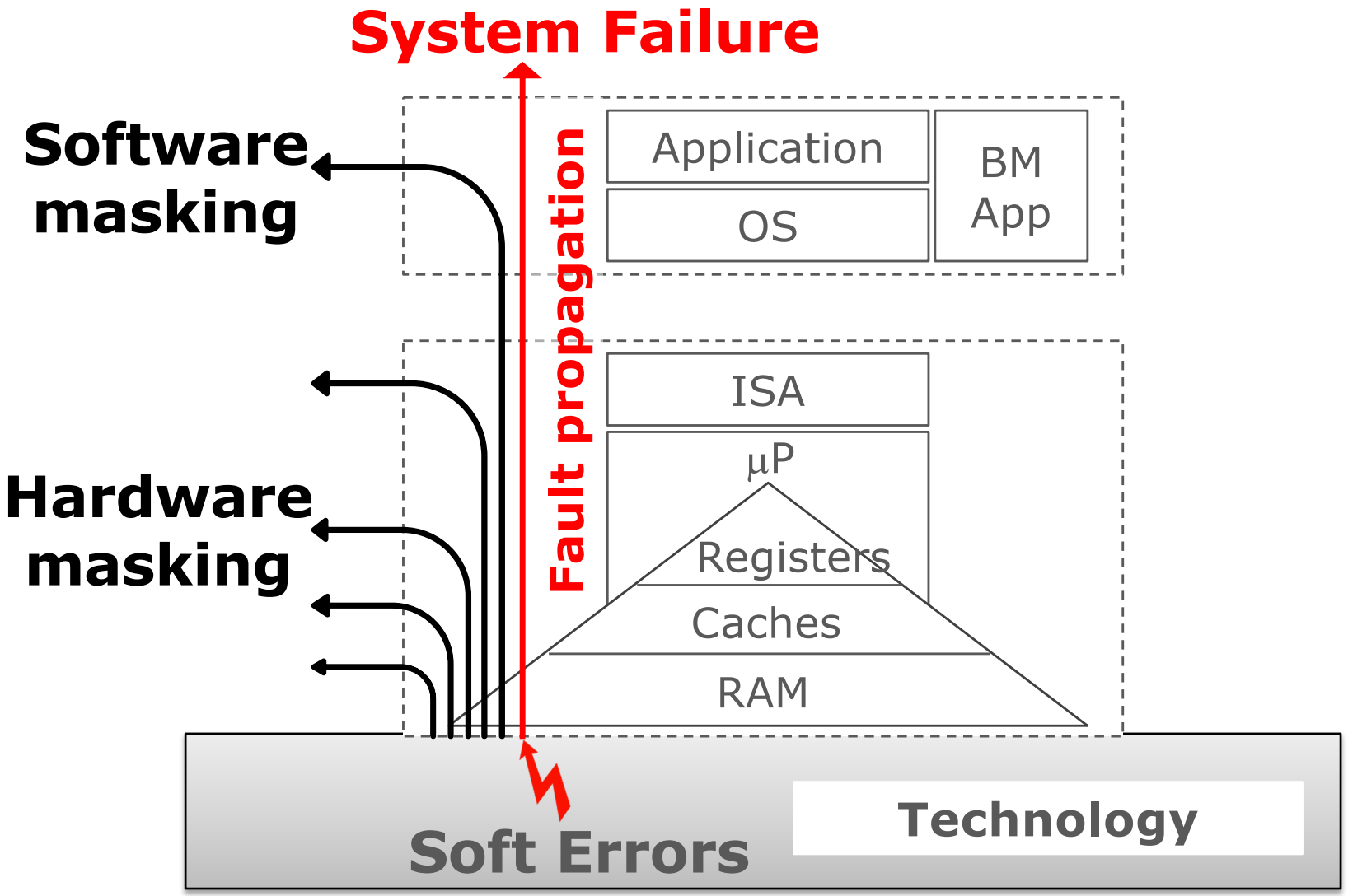
Alberto Bosio
École Centrale de Lyon, France

alberto.Bosio@ec-lyon.fr

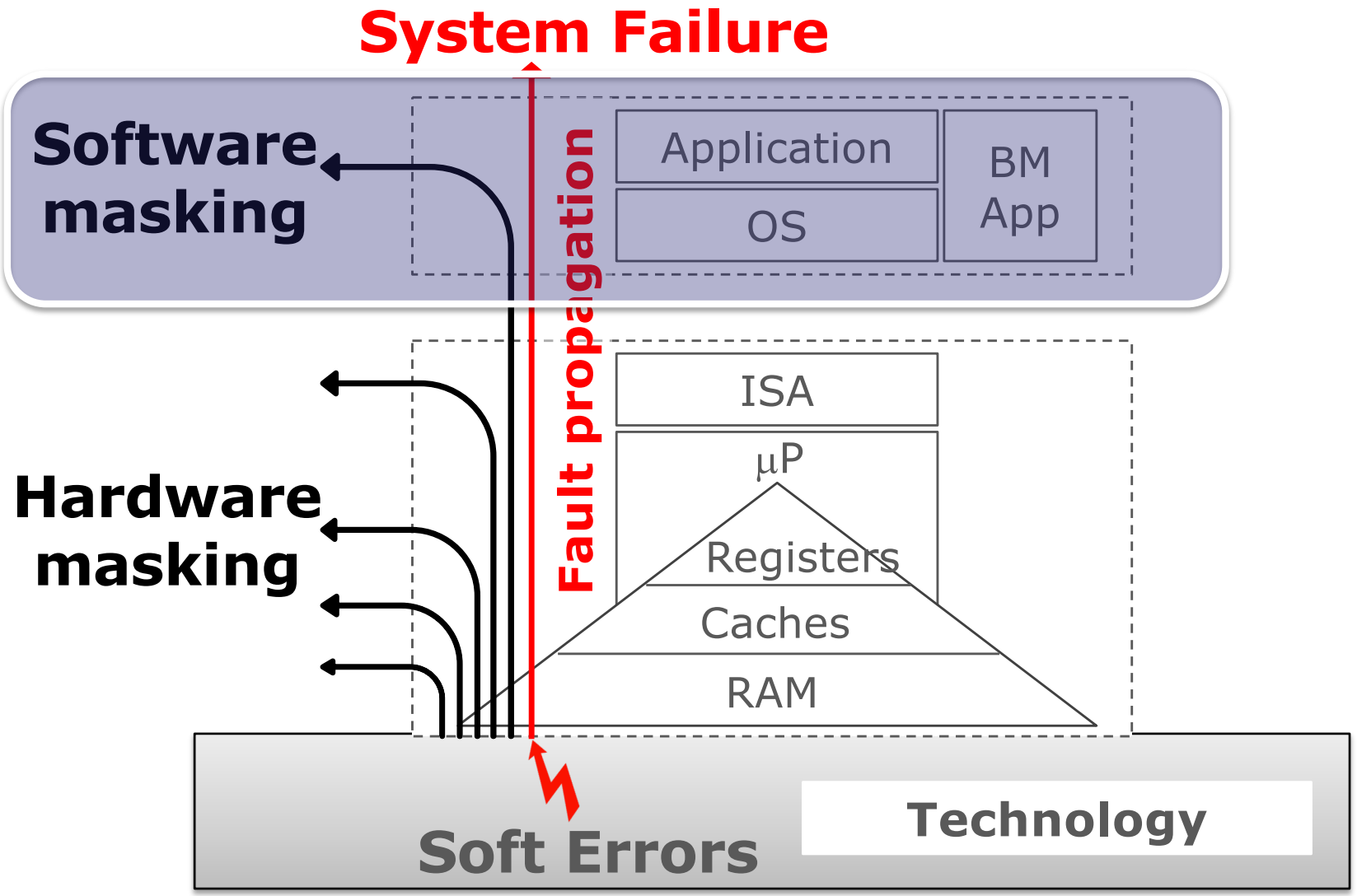
Acknowledgments

- Maha Kooli, Giorgio Di Natale

Context

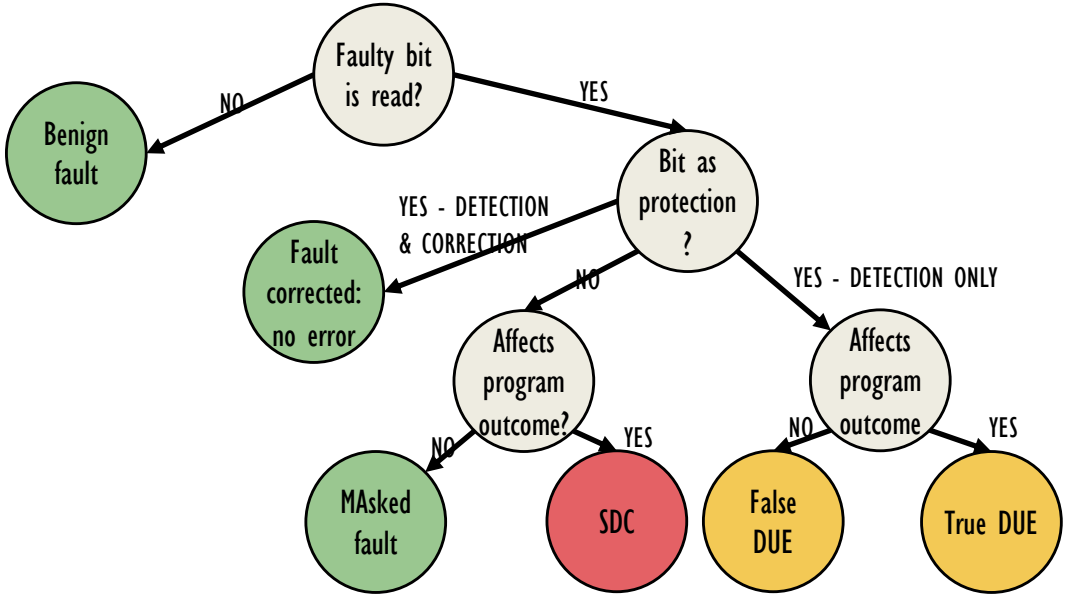

System Failure
Software masking
Hardware masking
Fault propagation
Application
OS
BM App
ISA
µP
Registers
Caches
RAM
Soft Errors
Technology

Context


System Failure
Software masking
Application
OS
BM App
Fault propagation
Hardware masking
ISA
µP
Registers
Caches
RAM
Soft Errors
Technology

SW level Resilience Assessment

- Inject a set of fault models representing the effect of soft errors in the software stack
  - Target faults in data and instructions
- Observe the impact on the software layer

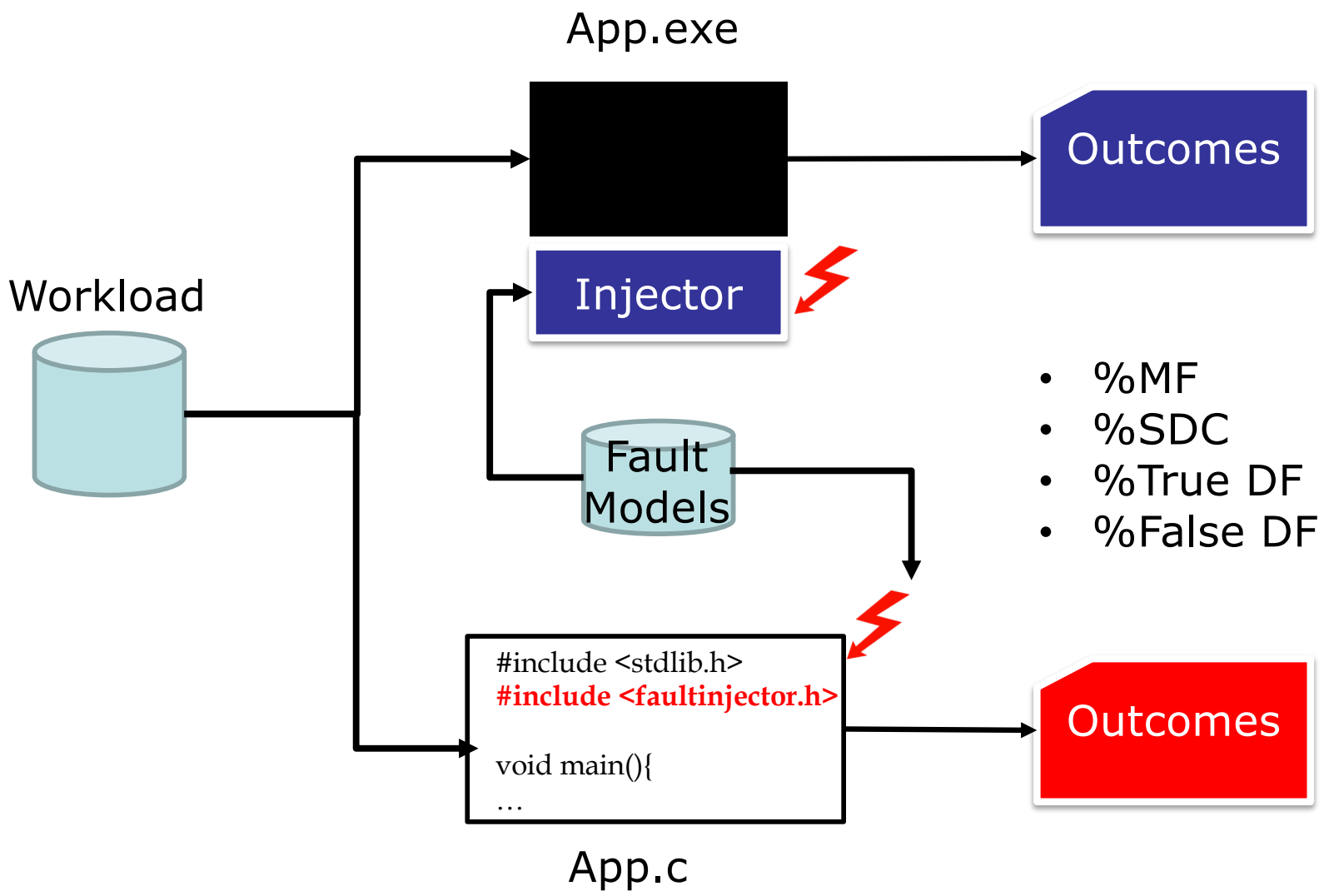


- **Masked Fault**
- **Silence Data Corruption**
- Detected Uncorrectable Error
  - False
  - True

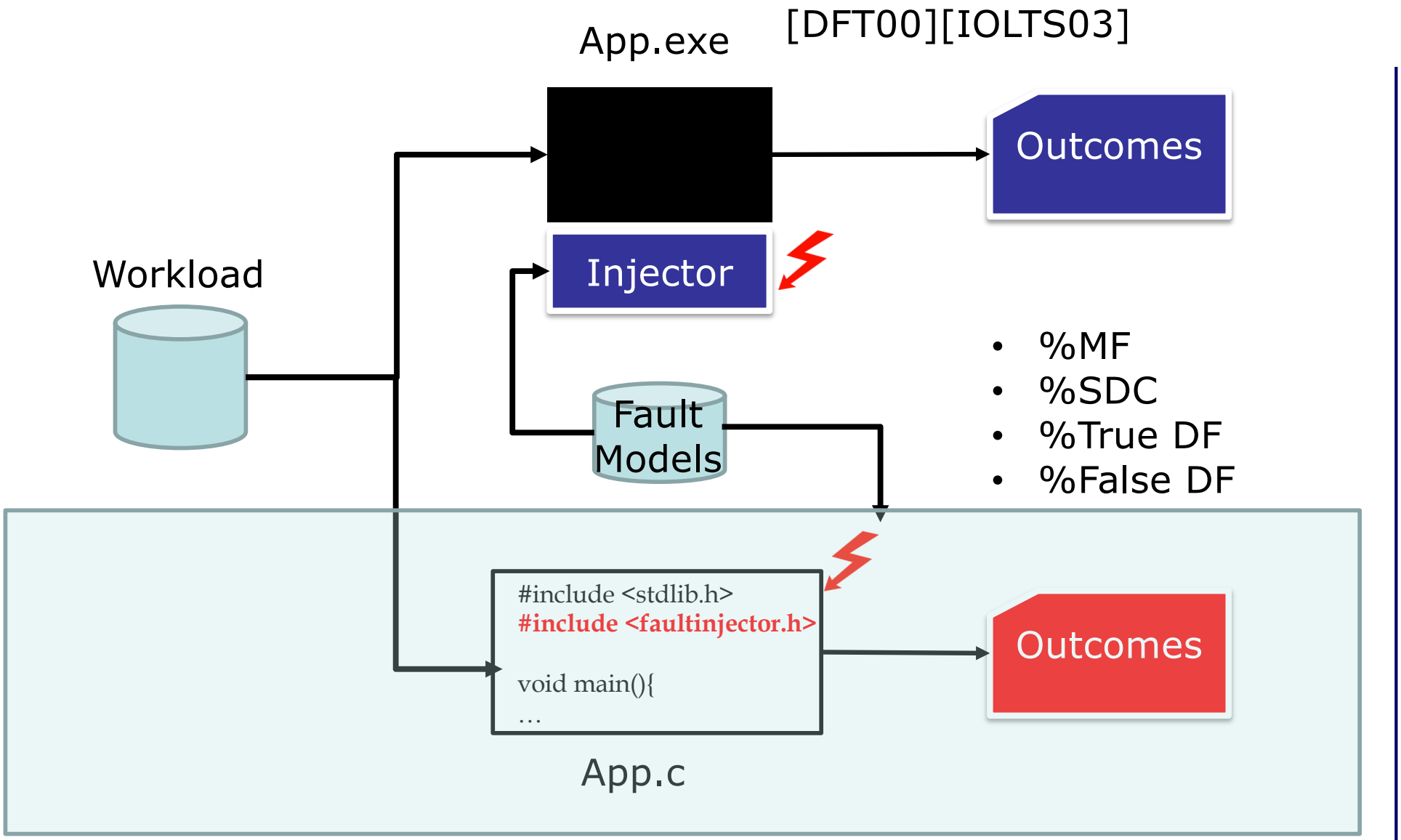
Outline

- SW Fault Injection Taxonomy
- Software Fault Models
- White Box approach
  - Validation
- Memory Hierarchy
  - Validation
- Conclusions

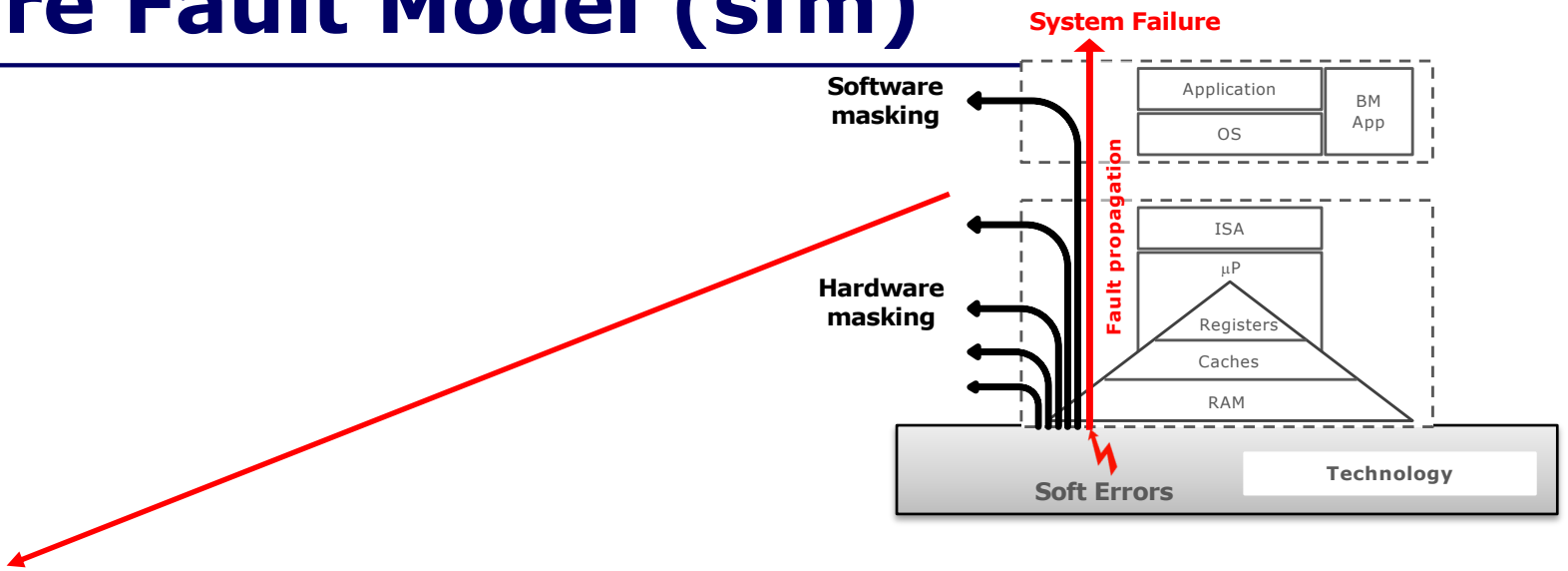
Taxonomy: Black & White



Taxonomy: Black & White

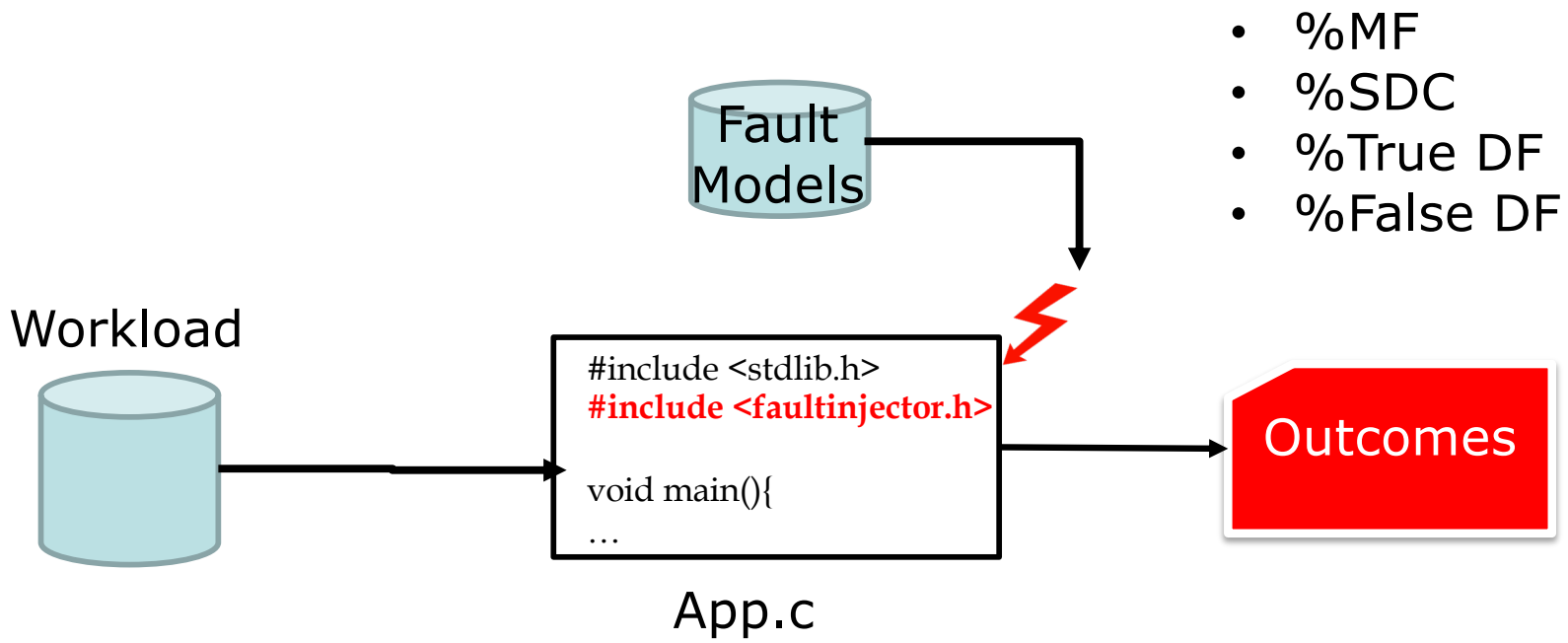


Software Fault Model (sfm)



Fault Model	Description	Example
Wrong Data in an Operand	- Effect of SEU occurring in memory segment storing the data - An operand of the virtual ISA changes its value	A = B A = C
Instruction Replacement	- Effect of SEU occurring in memory segment storing the code - An opcode in the virtual ISA is switched to a valid or invalid opcode	%A = add %B, %C %A = sub %B, %C

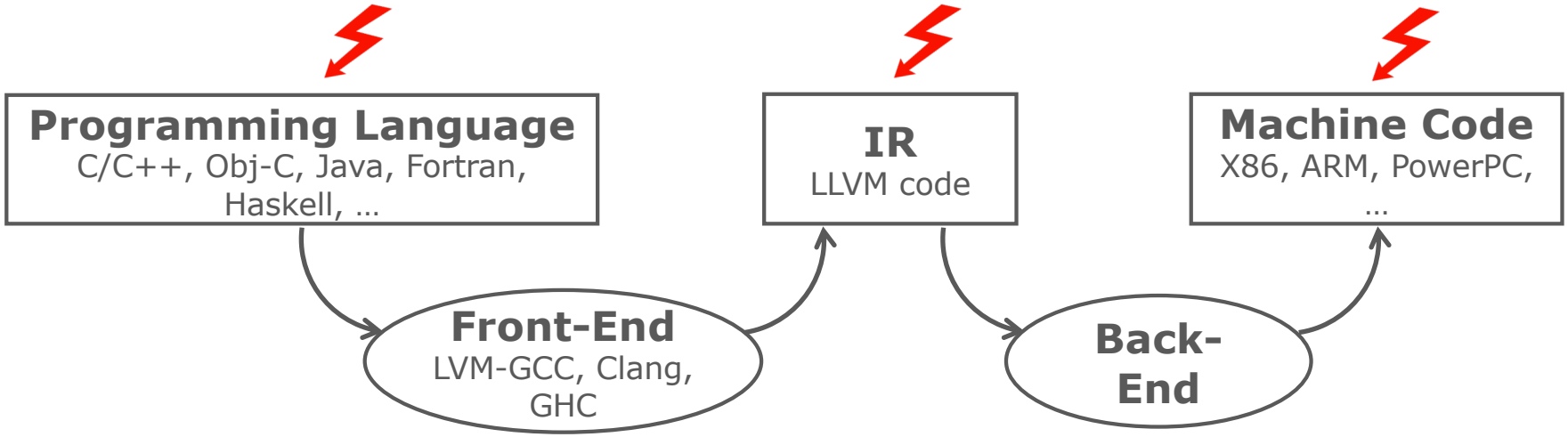
White Box: Injection Time



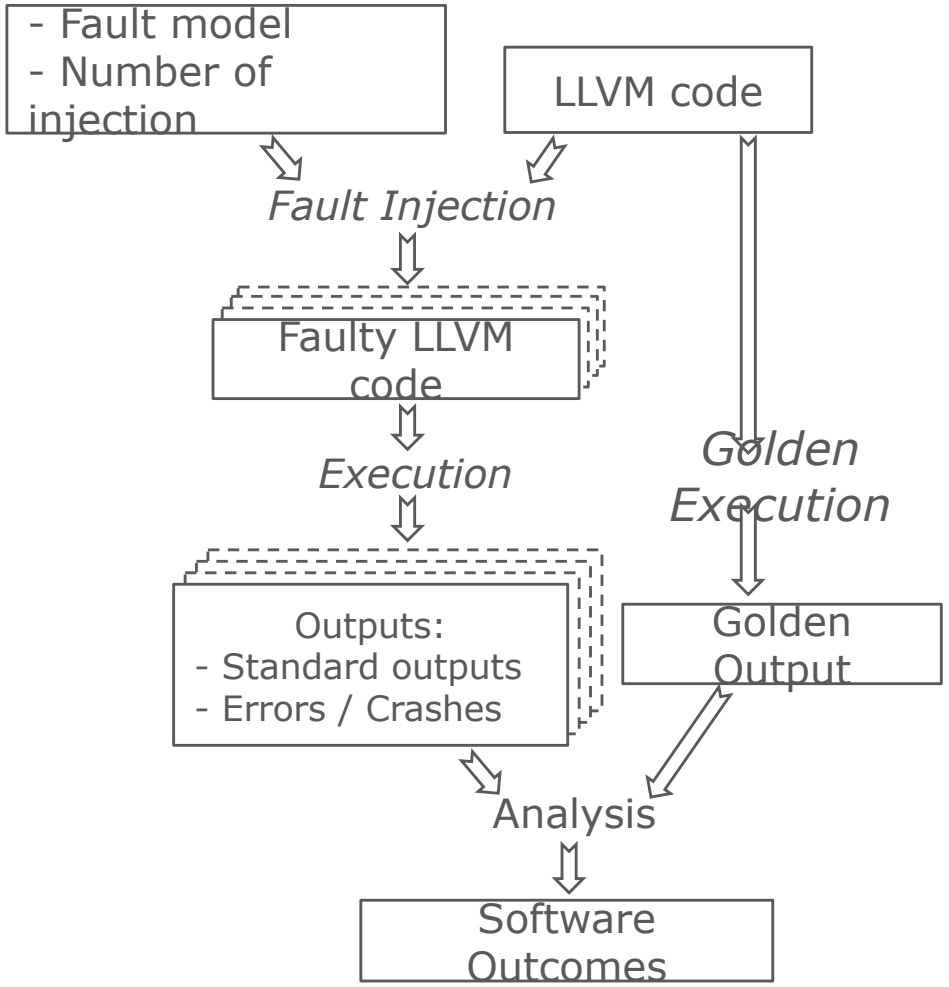
- **Static**: at compile time
- **Dynamic**: at run time

White Box: Which source code?

- High-Level: C, C++
- Assembly:
  - Virtual assembly: LLVM (Low Level Virtual Machine)
  - Real assembly: ARM, x86, …



LLVM-based Fault Injection [IOLTS16][ITC16]



Transient Fault Injection

```
define void @main(){
  ...
; <label>:1
  call @inject(i32* %a, i32 25)
  %3 = icmp ne i32 %2, 100
  br i1 %3, label %1, label %4
  ...
}

@clk = internal global i32 0
define void @inject(i32* %a, i32 %clkFI){
  %1 = icmp eq i32 @clk, %clkFI
  br i1 %1, label %2, label %3

  ; <label>:2
  %a1 = load i32* %a
  %aFI = xor i32 %a1, 64
  store i32 %aFI, i32* %a
  br label %3

  ; <label>:3
  %4 = add i32 @clk,1
  store i32 %4, i32* @clk
}
```

Loop

Transient injection in iteration 25

Variable BitFlip

C-based Fault Injection [DDECS16]

Program Source Code

```
#include <stdlib.h>

void main(){
…
}
```

Instrumentation

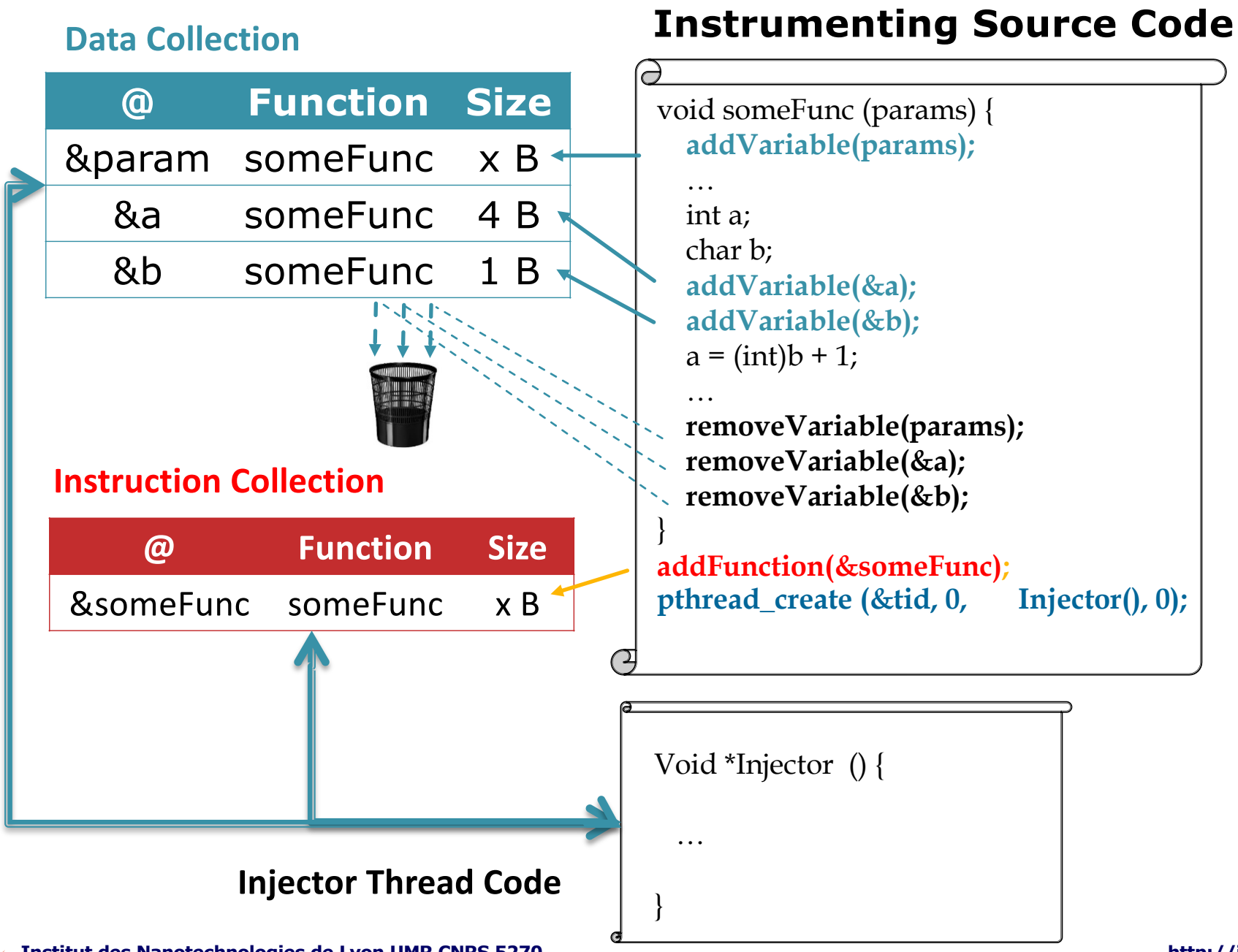
```
#include <stdlib.h>
#include <faultinjector.h>


void main(){
…
}
```

N executions

Fault Classification

N Outputs

Code Instrumentation



Injector

Fault Injection in Data

Original Source Code

```
void someFunc (params) {
    int a;
}
```

Modified Source Code

```
void someFunc (params) {
    int a;
    addVariable(&a);
}
```

```
*(address+ byte) ^= 1 << b i t ;
```

Fault Injection in Instructions

Original Source Code

```
void someFunc (params) {
    int a;
}
```

Modified Source Code

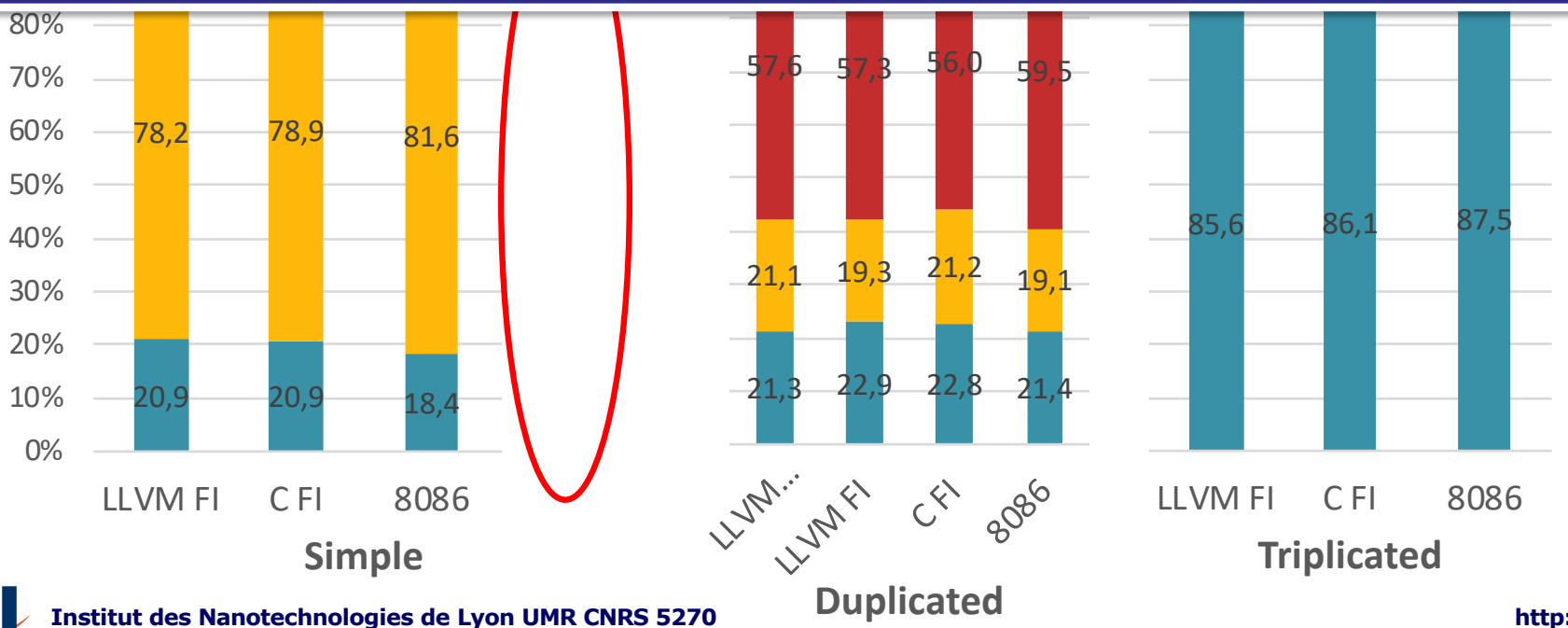
```
void someFunc (params) {
    int a;
}
addFunction(&someFunc);
```

```
*(address+ byte) ^= 1 << b i t ;
```

Validation

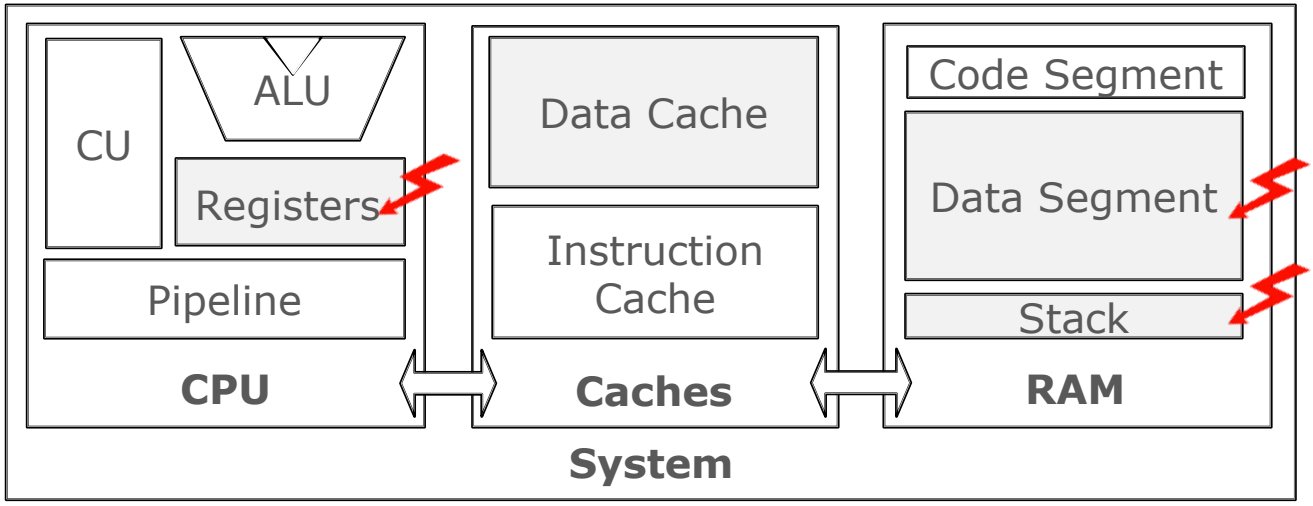
- **Benchmarks:** matrix multiplication
- **Comparison:**
  1. Simulation-based fault injection on Intel x86 processor
  2. FPGA-based fault injection on LEON3 processor

- Accurate reliability evaluation
  - Simple microprocessors without cache
  - Microprocessors with one or multiple cache levels

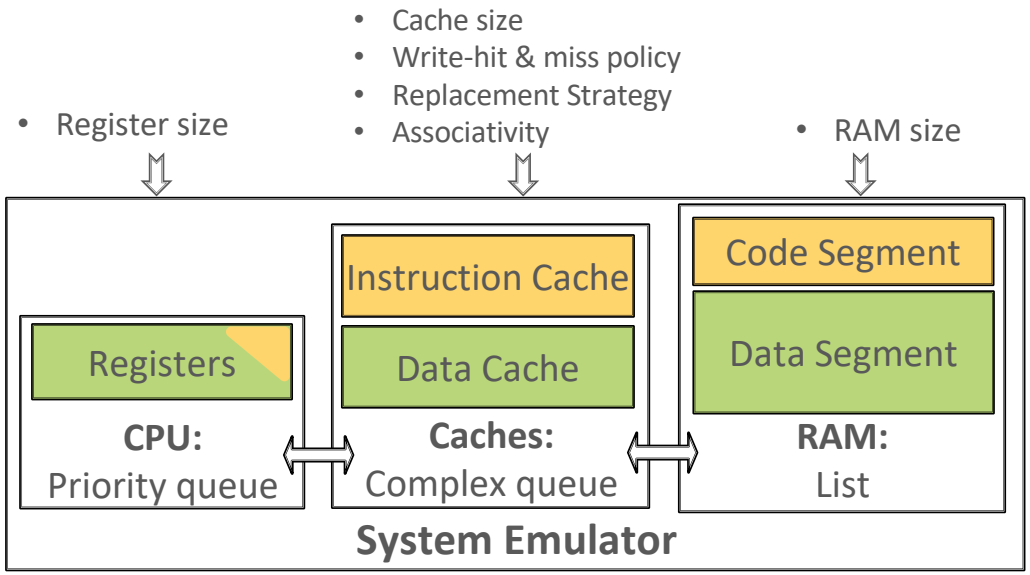


Granularity of the fault injection

- The same variable/instruction resides in different memory locations
  - At the software level, it is the same variable/instruction
  - At the hardware level only one copy is active and impacts the program execution

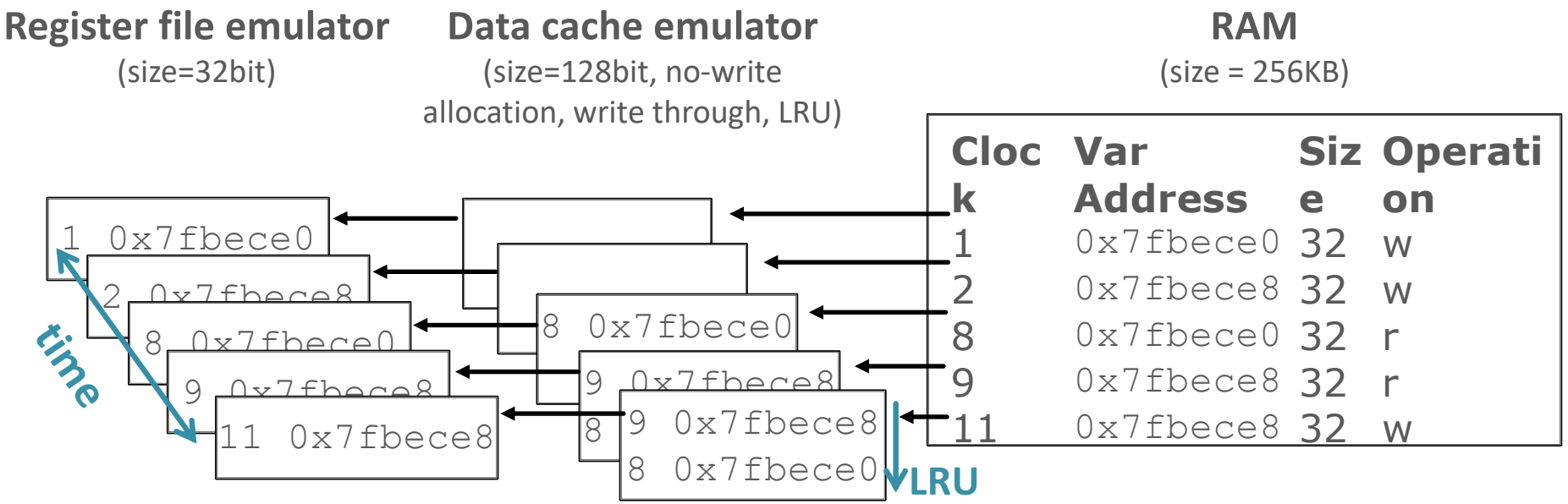


Memory Subsystem Emulator [VTS16]



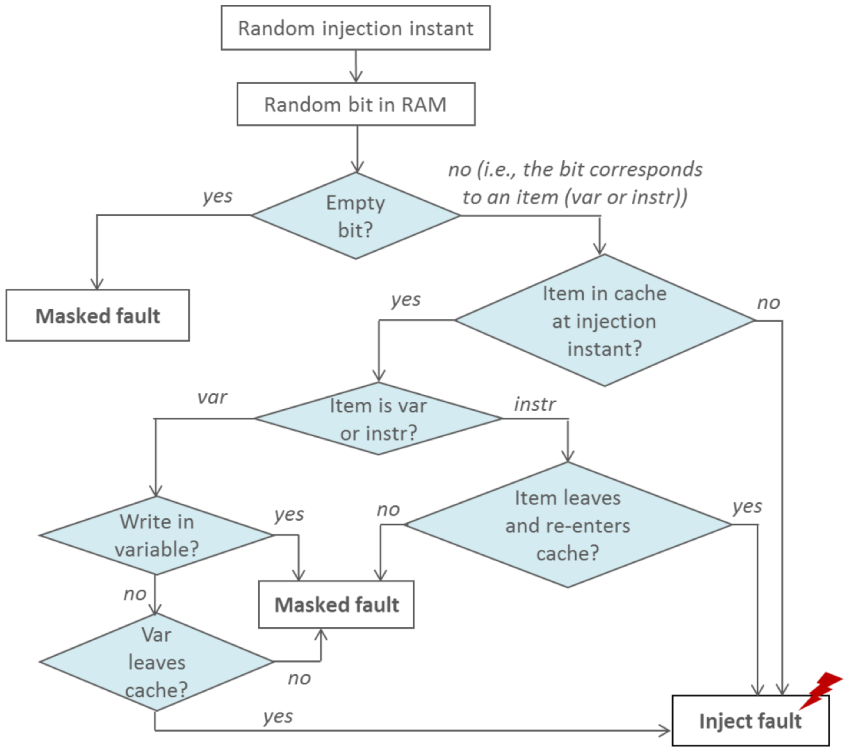
- Advantages:
  - Based on the virtual LLVM ISA: independence from microprocessor ISA (w.r.t. Gem5 and SimpleScalar)
  - Supports different hardware configurations

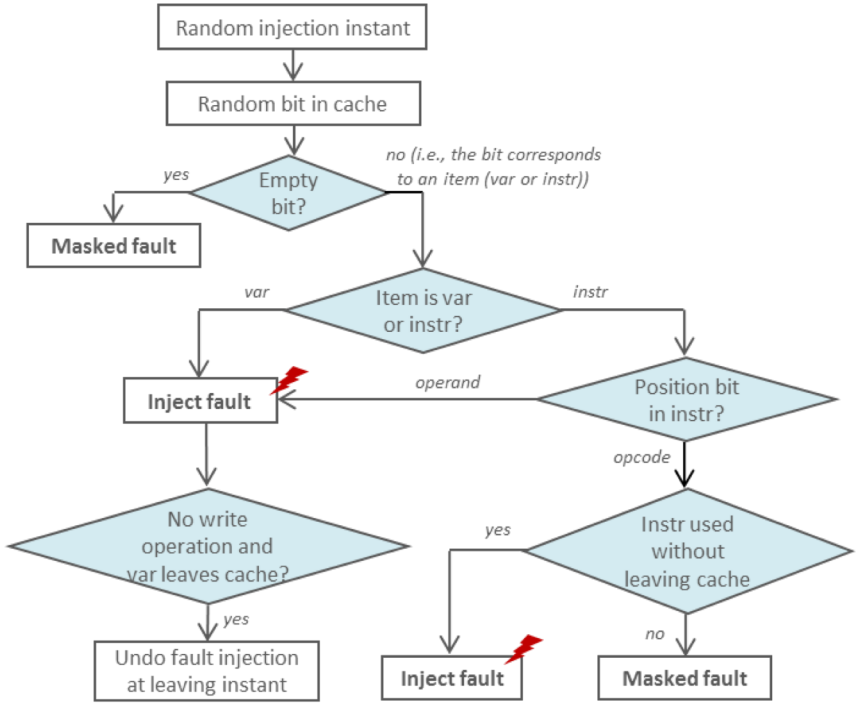
Memory Subsystem Emulator



Memory Subsystem Emulator

- The activation of the cache has big influence on the fault simulation
- Adapt the proposed fault injections to modern processors
- Integrate the memory subsystem emulator

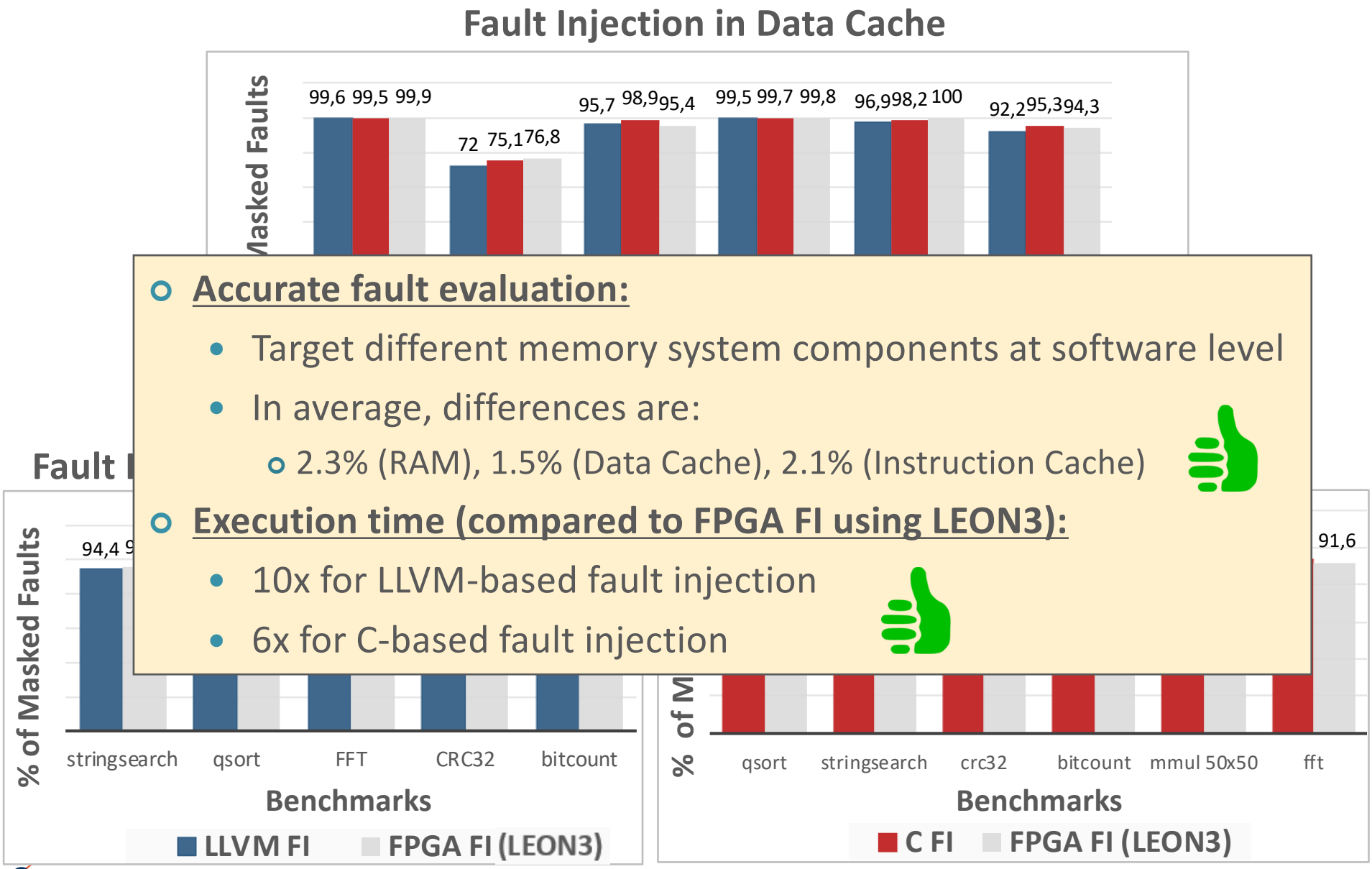




Validation

- **Benchmarks:**
  - Matrix multiplication 50x50
  - *MiBench suite*
- **Comparison:**
  - *FPGA-based fault injection (*SCFIT*) using the LEON3 processor*
- **Injections:**
  - 10K fault injections per program
- **Configurations:**
  - *RAM:* 256 KB size
  - *Cache:* 4 KB size, write-through, no-write allocate, last recently used and fully associative
  - *Register-File:* 512 B size

Validation



- **Accurate fault evaluation:**
  - Target different memory system components at software level
  - In average, differences are:
    - 2.3% (RAM), 1.5% (Data Cache), 2.1% (Instruction Cache)
- **Execution time (compared to FPGA FI using LEON3):**
  - 10x for LLVM-based fault injection
  - 6x for C-based fault injection

Comparison

Existing Methodologies			Hardware Location		Accuracy	Time	Hardware Cost	Early Design
			μP Core	Rest				
Hardware-based Fault Injection	Physical FI	**Radiation**: FIST [Karlsson94]	○		+	+	−	−
		Laser: [Velazco92] [Fouillat04]						
		Pin: RIFLE [Madeira94] MESSALINE [Arlat90]						
	FPGA FI	[AntoniLeveugle03] SCFIT [Ebrahimi14]	+		+	○	○	−
	Simulation FI	**VHDL:** VERIFY [Sieh97]	+		+	−	+	○
		Verilog: LIFTING [Bosio08]						
		Gem5: [Gizopoulos15]	○		+	○	+	○
Software-based Fault Injection		**OS-level:** FERRARI[Kanawati95] XCEPTION[Carreira98]	−		−	+	+	○
		Virtual-level: LLVM-based FI	−	+	+	+	+	○
		Code-level: C-based FI						

Conclusions

- New approaches to evaluate the reliability of computing systems running software
- Subsystem emulator based on LLVM framework allowing to emulate, at software level, memory components (RAM, cache, register file)
- Compared to hardware-based fault injection
  - Accurate (enough)
  - Low execution time
  - Not required hardware definition while considering hardware components