Self-Test Mechanisms for Automotive Multi-Processor System-on-Chips

Andrea Floridia

Supervisor: Ernesto Sanchez

23rd September 2021 – Ph.D. Final Discussion

Outline

- Problem Statement
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - Deterministic cache-based execution of Software Test Libraries
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - JTAG-based fault emulation platform

Outline

- **Problem Statement**
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - Deterministic cache-based execution of Software Test Libraries
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - JTAG-based fault emulation platform

Problem Statement – Automotive MPSoCs

- Automotive Electronics Control Units (ECUs) are based on multiple processor cores (MPSoCs):
  - **Homogeneous**: processor cores of the same type;
  - **Heterogeneous**: processor cores differ;

- Different in-field test solutions required to comply ISO26262 requirements:
  - Hardware-based (Logic BIST, LBIST);
  - Software Test Libraries (STLs) – for the most critical component, the processor.

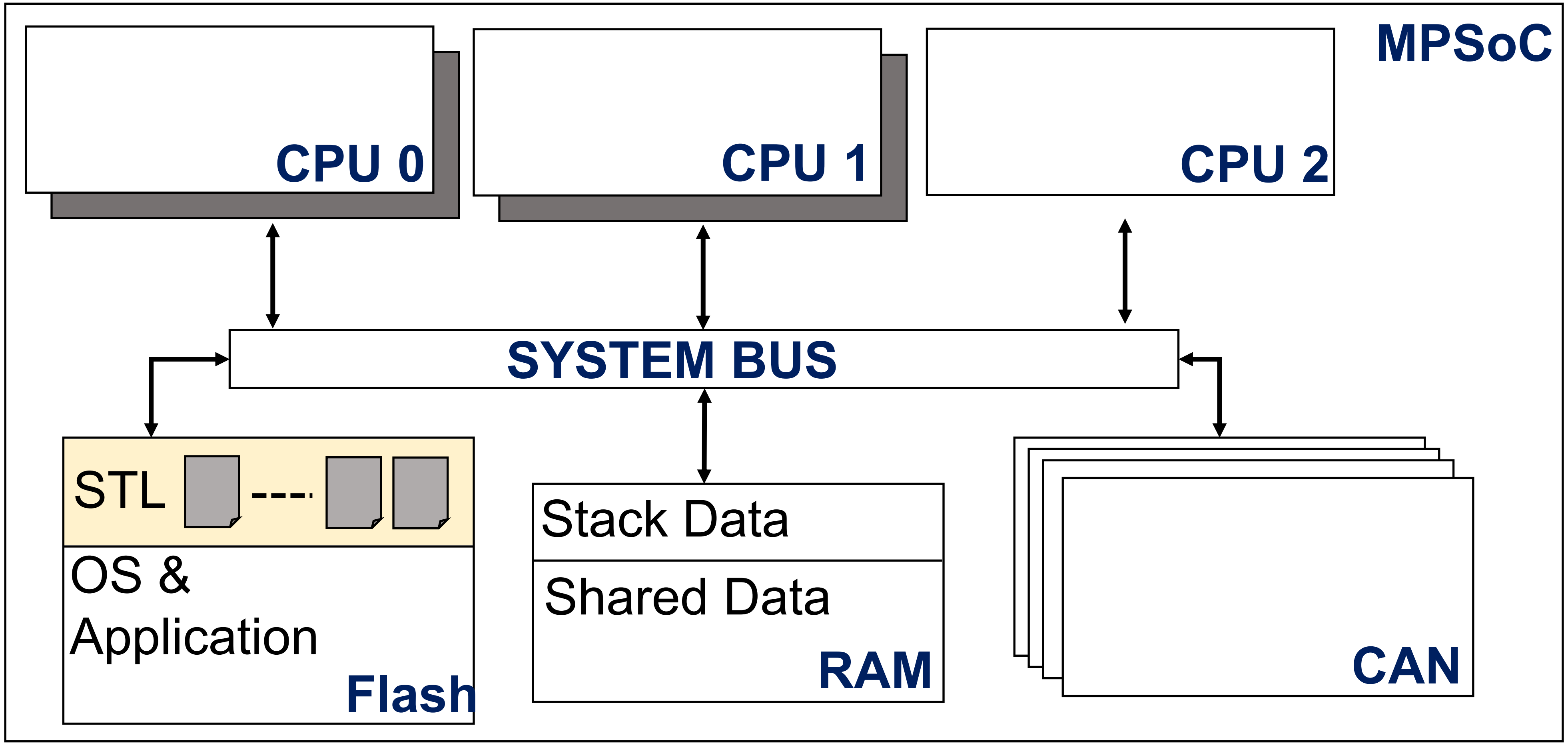
Problem Statement – BIST-based mechanisms

- In-field test mechanisms major hurdle: **test application time**;
- With BIST-based approaches, to reach the same coverage figures, pattern count increases;
- Recent researches focused **on BIST-based methods**:
  - Power during shift;
  - Optimal insertion of test point for improving controllability and observability;

Problem Statement – Software-based mechanisms

- STL: self-test procedures targeting faults within the CPU;
- Test procedures categories:
  - Run-time test procedures – low invasiveness;
  - **Boot-time test procedures** – high invasiveness (e.g., **system RAM**);
- Consolidated strategies for single-core devices;
- For MPSoCs: **exclusively** end-of-manufacturing testing.

Problem Statement – STL scenario

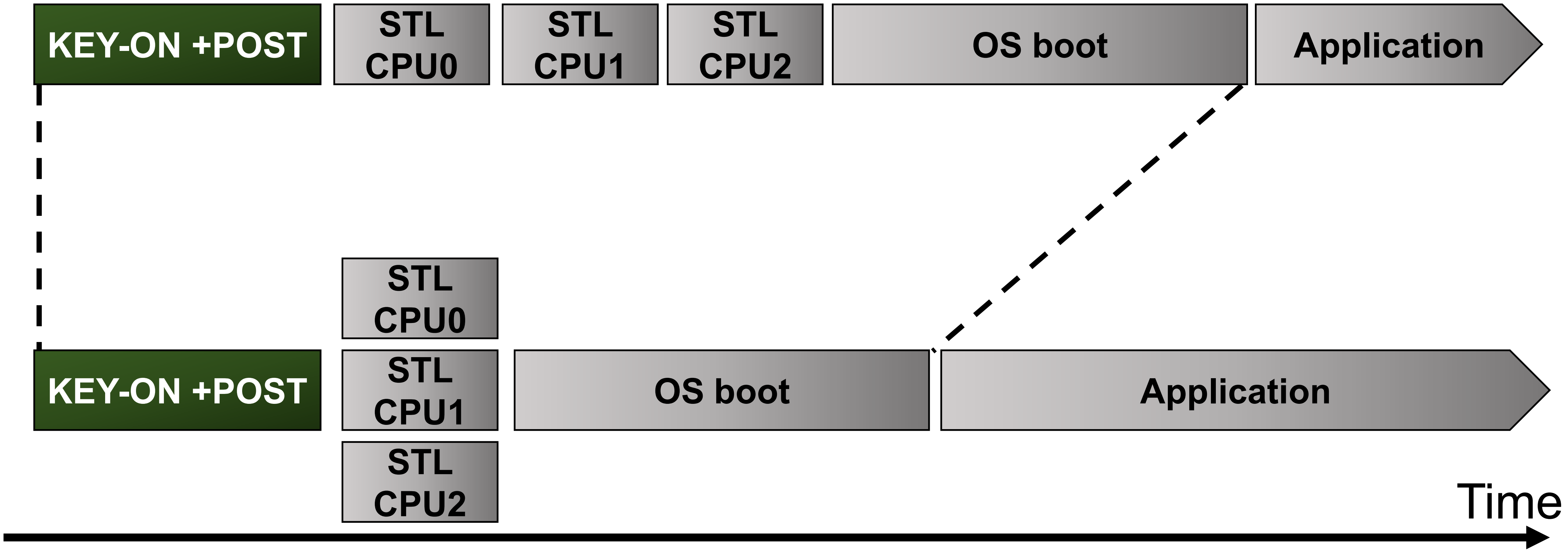


Outline

- Problem Statement
- On-line self-test mechanisms
  - **Software Scheduler for Software Test Libraries**
  - Deterministic cache-based execution of Software Test Libraries
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - JTAG-based fault emulation platform

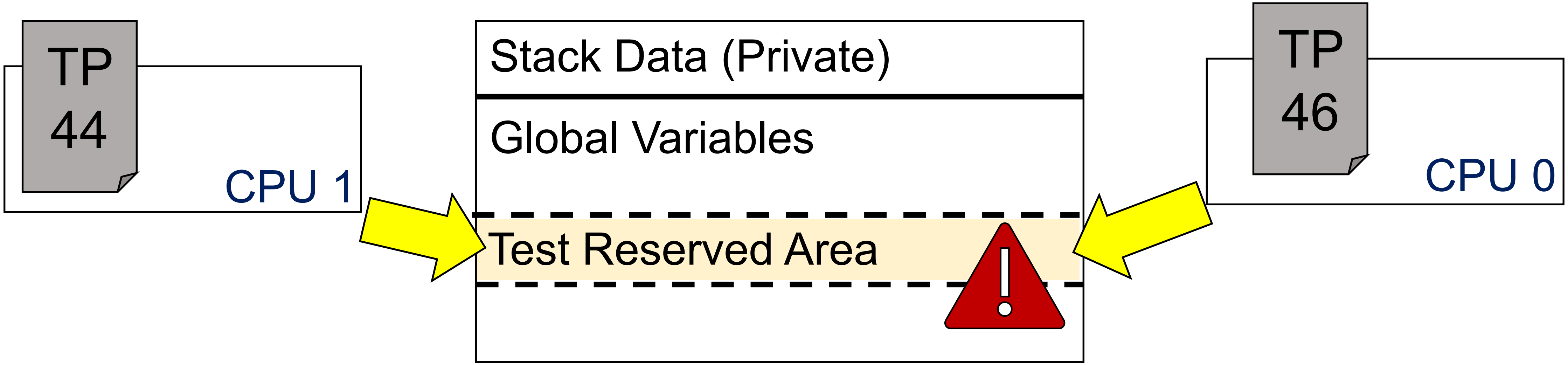
Software Scheduler for STLs – Challenges

- Parallel test to increase system availability:



Software Scheduler for STLs – Challenges

- Parallel test to increase system availability:
  - Run-time tests – executed without problems;
- Boot-time tests create parallelization difficulties due to shared resources (e.g., the shared portion of system RAM):

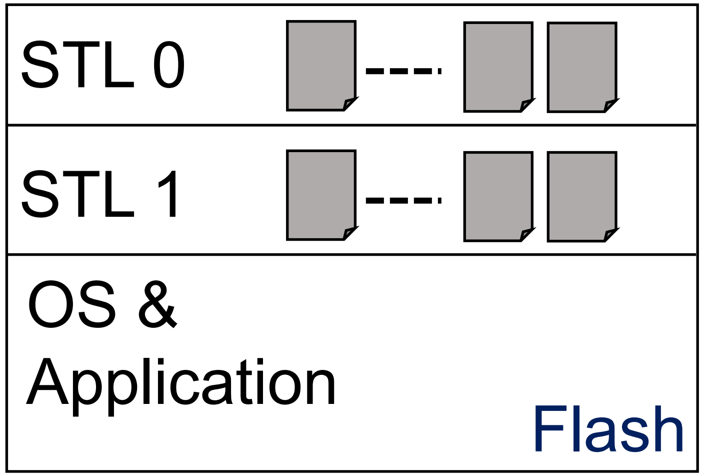


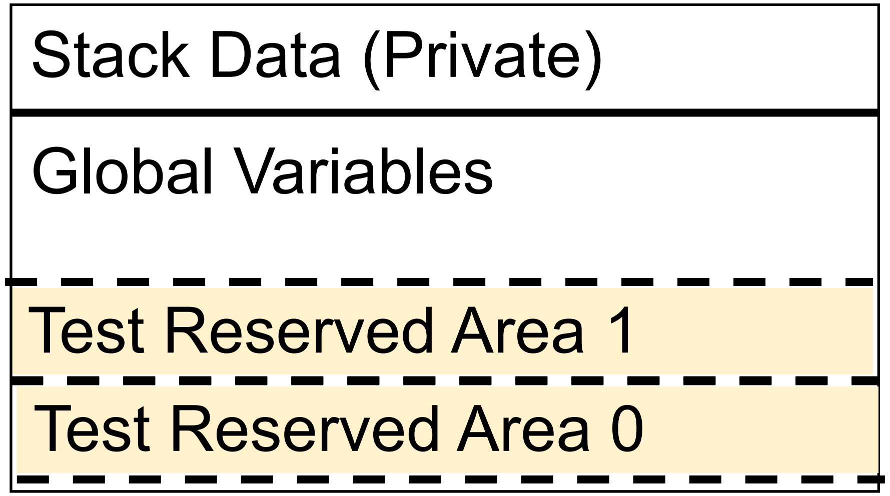
Software Scheduler for STLs – Challenges

- Parallel test to increase system availability:
  - Run-time tests – executed without problems;
- Boot-time tests create parallelization difficulties due to shared resources (e.g., the shared portion of system RAM):
  - Multiple “Test Reserved Area” **not feasible in real applications**;
  - Additionally, replication sometimes not physically possible;

Software Scheduler for STLs – Main features

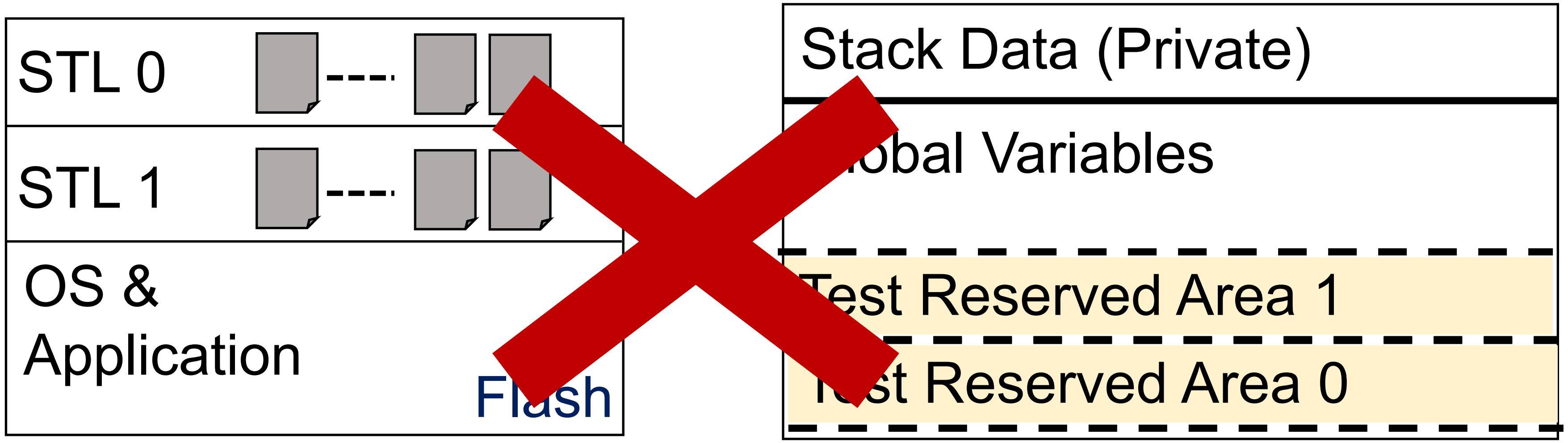
- Main characteristics of a multi-core STL scheduler:
  1. Does **not alter** STL fault coverage;
  2. Minimize system resources usage:





Software Scheduler for STLs – Main features

- Main characteristics of a multi-core STL scheduler:
  1. Does **not alter** STL fault coverage;
  2. Minimize system resources usage:



Software Scheduler for STLs – Main features

- Main characteristics of a multi-core STL scheduler:
  1. Does **not alter** STL fault coverage;
  2. Minimize system resources usage:
     - **Unique copy** of the STL in code memory, and;
     - **No replication of shared resources** (e.g., unique portion of system RAM available for testing purposes);
  3. Does not rely on OS support.

Software Scheduler for STLs – Observations

- Few test programs cannot be executed in parallel (~12%) due to shared resources;
- Other test programs access the system bus for fetching data from code memory;

- Multi-core system as distributed system ➔ **decentralized scheduler (DS):**
  - Set of local schedulers interacting each other.

Decentralized Scheduler for STLs

- Local schedulers interactions through mutex:
  - shared resource is **busy/free**;
- **Each scheduler** has 3 data structures:

1. **TestTable**: ordered list of test programs composing the STL;
2. **PendingList**: tracks the test programs **to be executed**;
3. **ShareResource**: list(s) of test programs that **cannot** be executed in parallel due to shared resources.

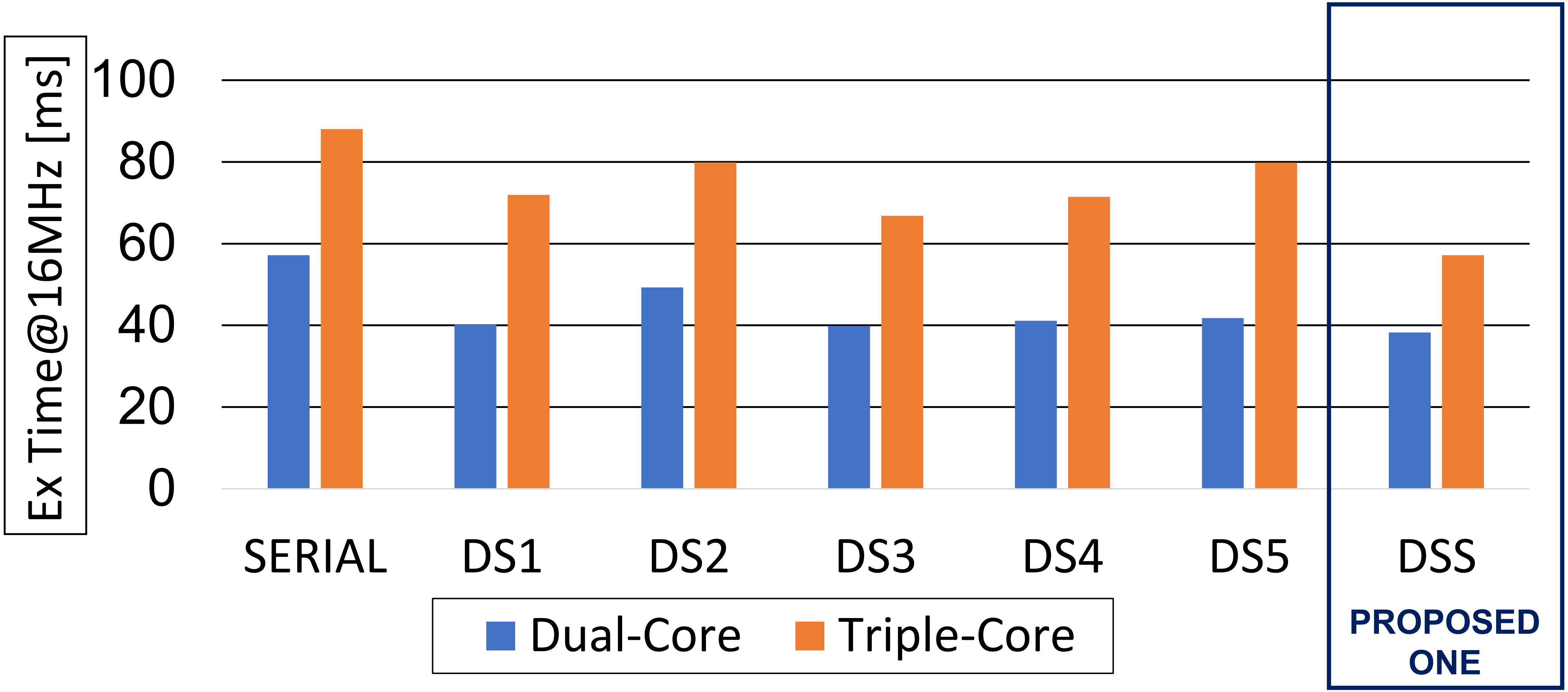
Decentralized Selfish Scheduler

- Heuristics: programs within ShareResource executed monolithically – without freeing the shared resource;
- The resource is released **at the end** of ShareResource only (**selfish**);
- If a test program requiring the shared resource cannot be executed (resource busy) is skipped, and another test program is executed ➔ Reduced number of conflicts for accessing shared resources;
- STL fault coverage unaltered: **non-preemptive scheduler**;

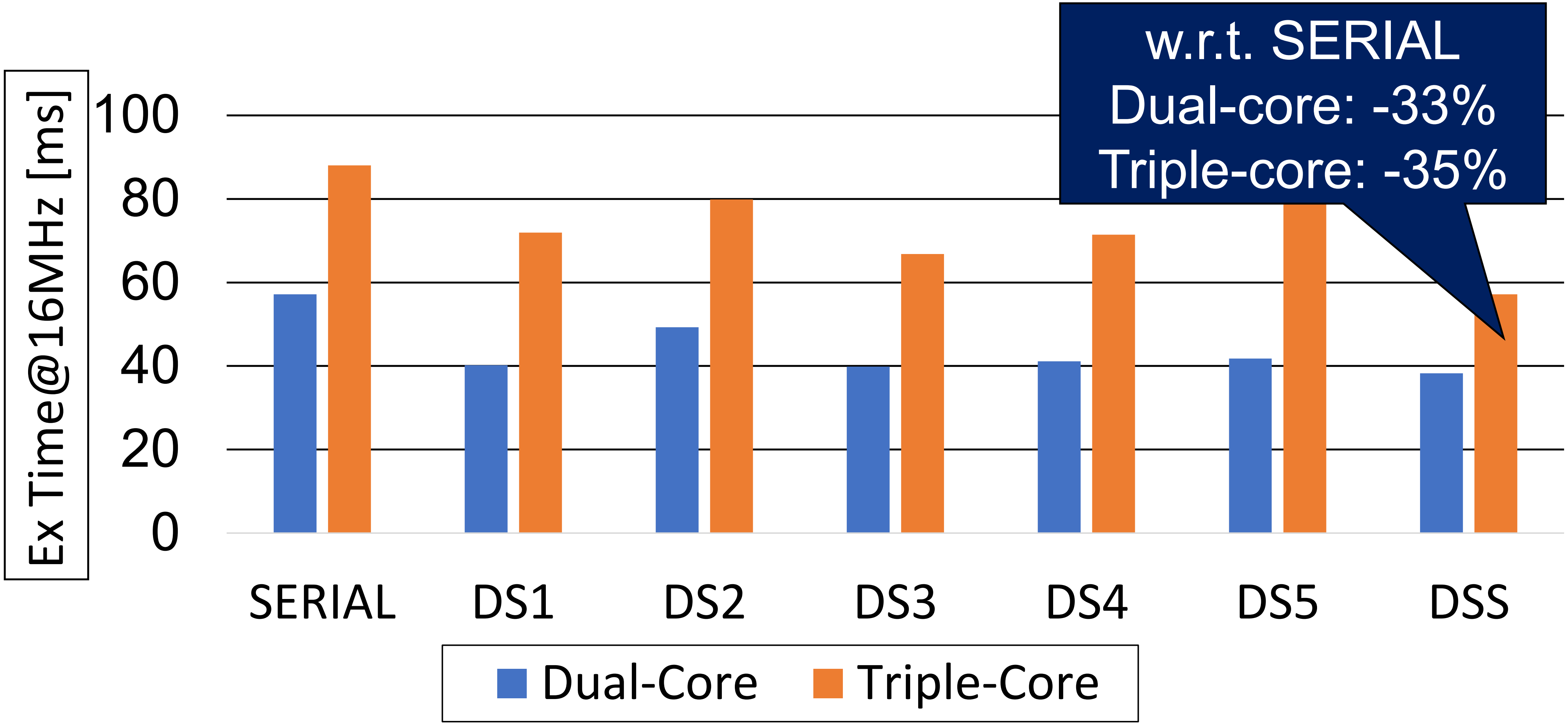
Experimental Results – Decentralized Scheduler

- Experiments carried out on industrial heterogenous/homogeneous MPSoCs;
- Different Decentralized Schedulers (**DS1-5**) compared against the proposed one (**DSS**);
- DSS cumulative memory overhead: less than 100KB.

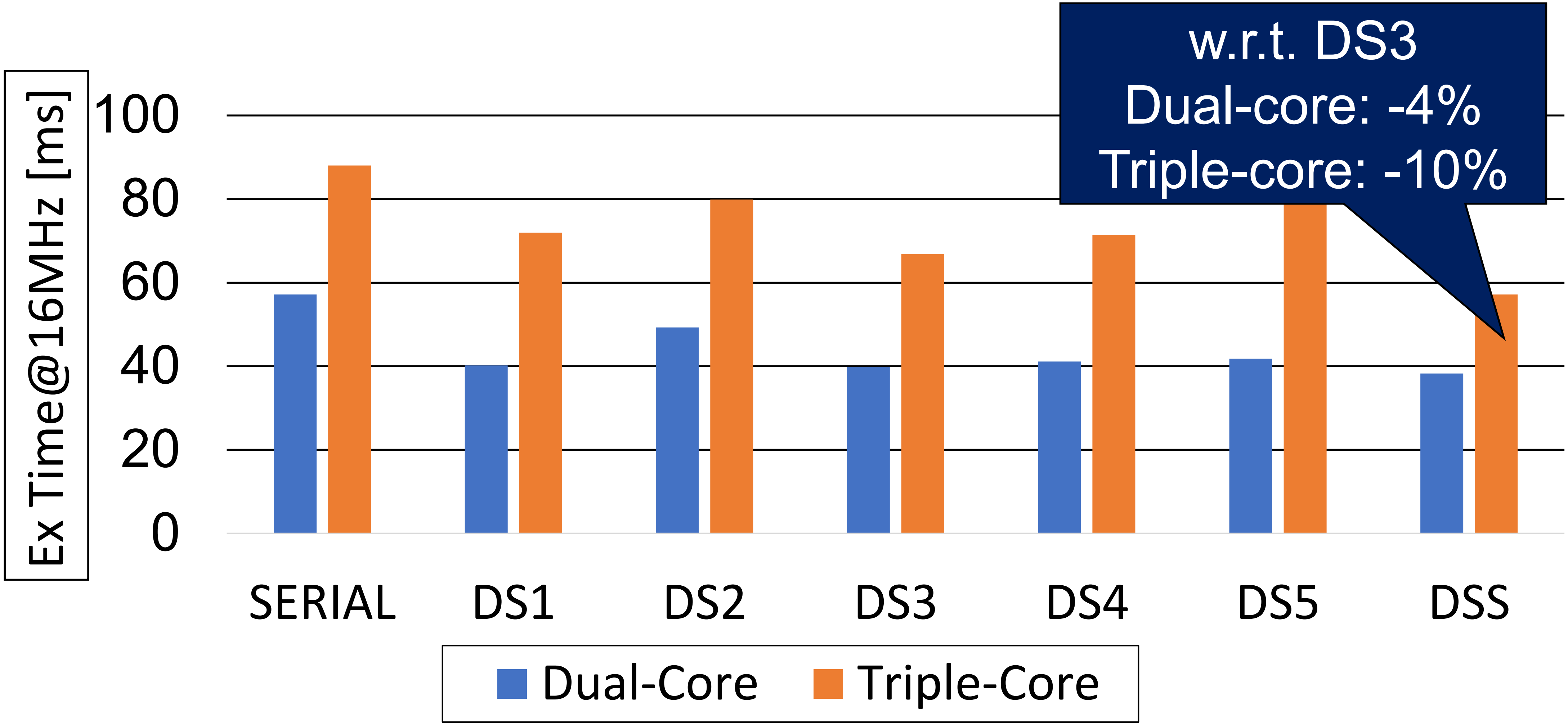
Experimental Results – Homogeneous, single-resource



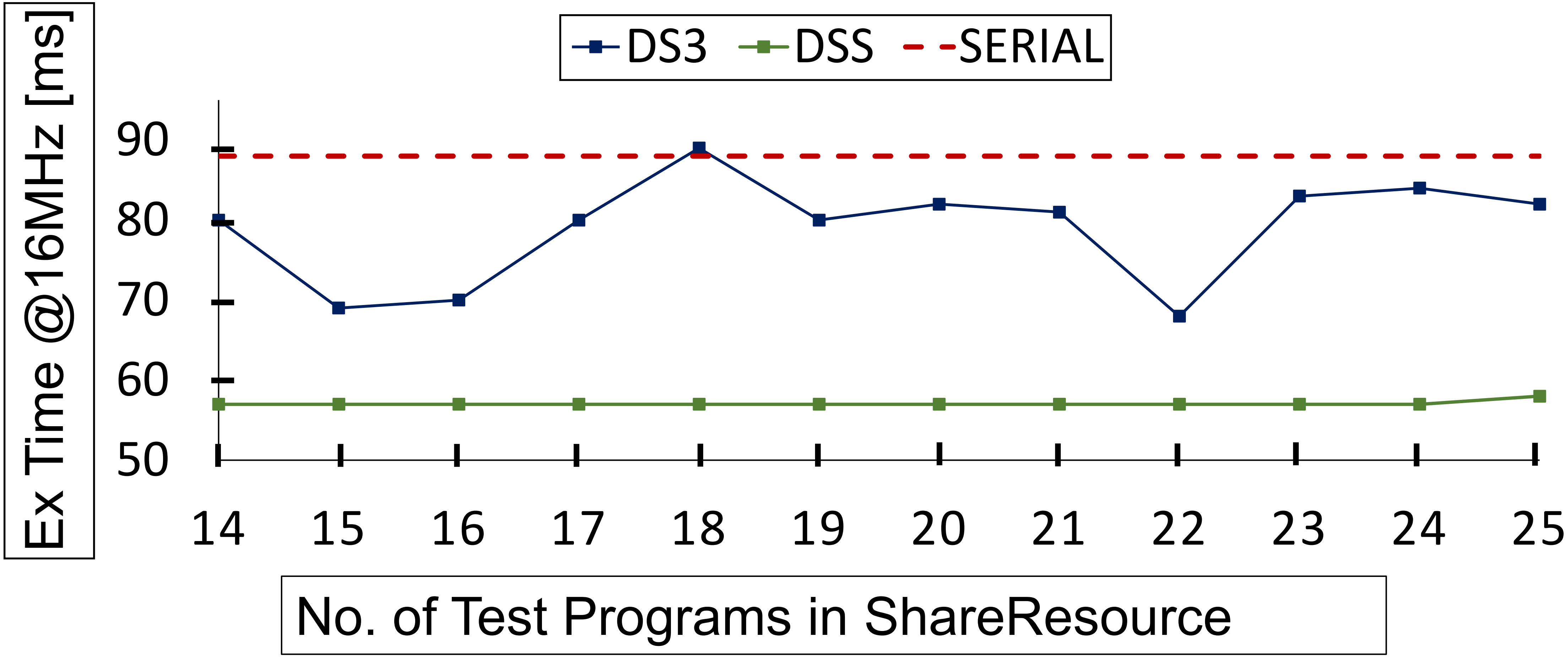
Experimental Results – Homogeneous, single-resource



Experimental Results – Homogeneous, single-resource



Experimental Results – DS3 vs DSS in Triple-Core



Software Scheduler for STLs – Conclusions

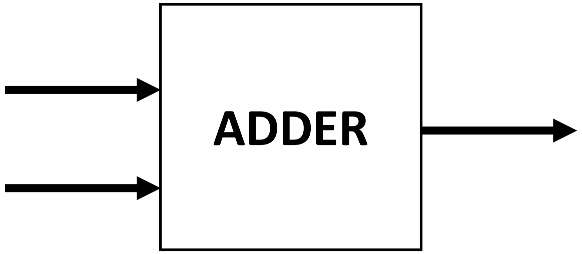
- Decentralized Selfish Scheduler for multi-core STL:
  - Reduced Test Application time;
  - Minimum Resource usage: identical processor cores exploit same scheduler image (1 scheduler per STL to be executed);
  - Unaltered STL fault coverage;
- Such scheduler supports:
  - Heterogeneous/Homogeneous MPSoCs;
  - Multiple shared resources.

Outline

- Problem Statement
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - **Deterministic cache-based execution of Software Test Libraries**
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - JTAG-based fault emulation platform

STLs – Additional details

- Detection mechanism: **test signature**;
- Boot-time – some require a proper sequence of instructions **without any interruption**;

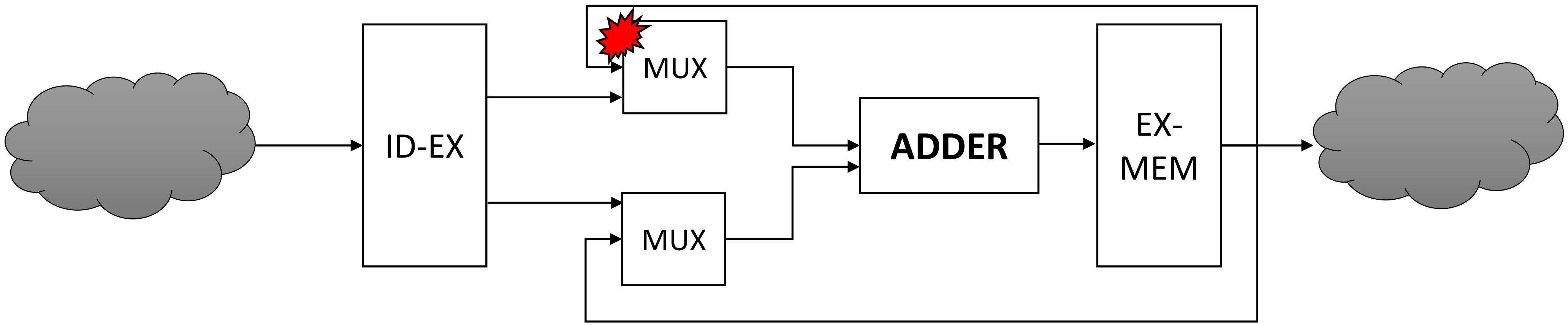



```
; R4 Signature reg
…
LOAD R5, PATTERNS(R1)
LOAD R6, PATTERNS+4(R1)
ADD R7, R5, R6
ACCUMULATE(R4, R7)
…
CHECK(R4, EXPECTED_SIGNATURE)
```

Problem Formulation – Effects & Consequences

- Higher system bus contention ➔ Embedded Software suffers of limited determinism;
- **Effects** on the self-test procedures:
  - **Higher number of pipeline stalls** ➔ the exact stream of instructions entering the pipeline cannot be determined in advance anymore;
- **Consequences** on boot-time procedures:
  - Uncertain Fault Coverage;
  - Unstable Signature.

Uncertain fault coverage – Forwarding mechanism

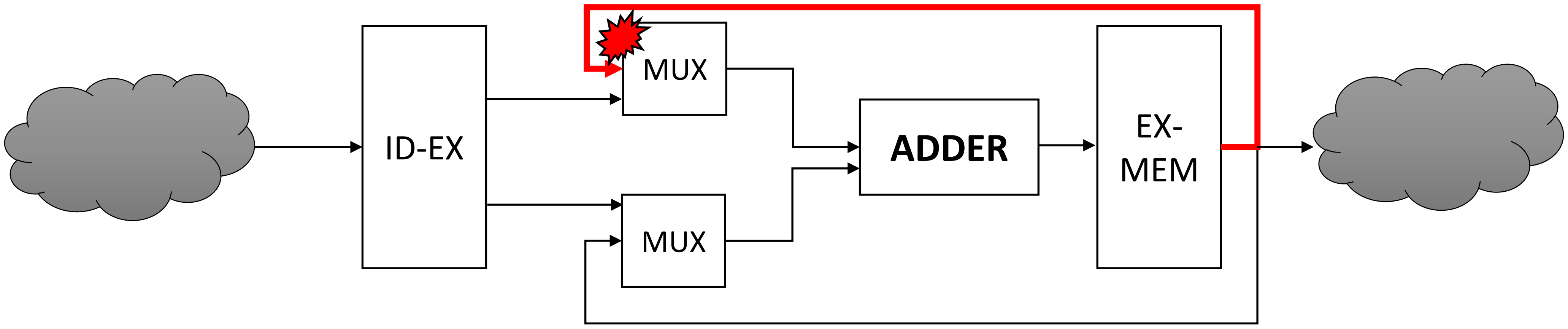



```
…
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6
ADD R7..	IF	ID	EX	MEM	WB	
ADD R10..		IF	ID	EX	MEM	WB

Single-Core

Uncertain fault coverage – Forwarding example

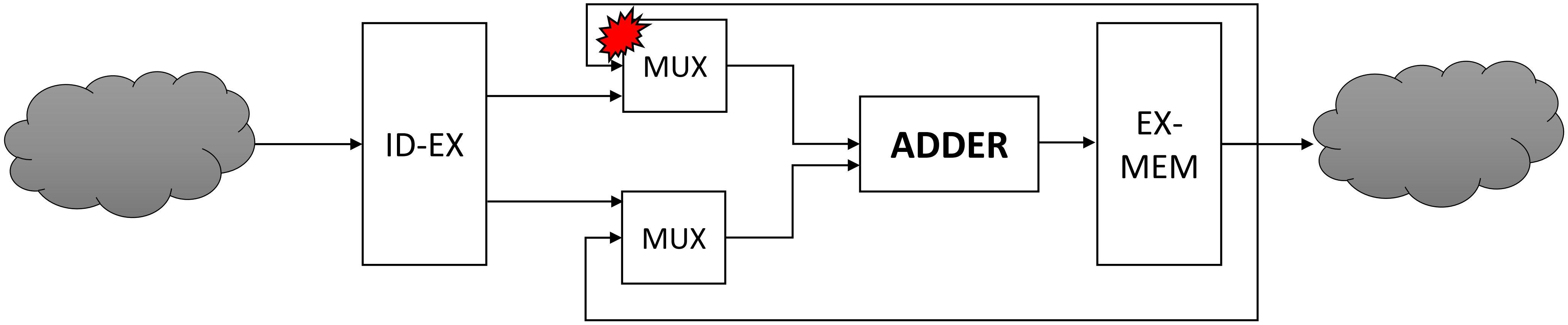



```
...
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6
ADD R7..	IF	ID	EX	MEM	WB	
ADD R10..		IF	ID	EX	MEM	WB

Single-Core

Uncertain fault coverage – Forwarding mechanism

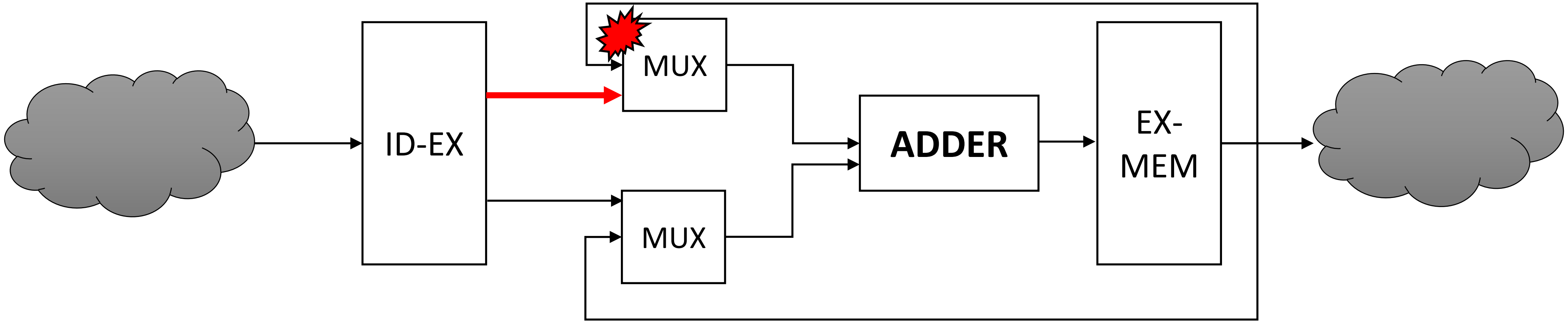



```
…
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6	7	8	9
ADD R7..	IF	ID	EX	MEM	WB				
ADD R10..		STALLS			IF	ID	EX	MEM	WB

Multi-Core

Uncertain fault coverage – Forwarding mechanism



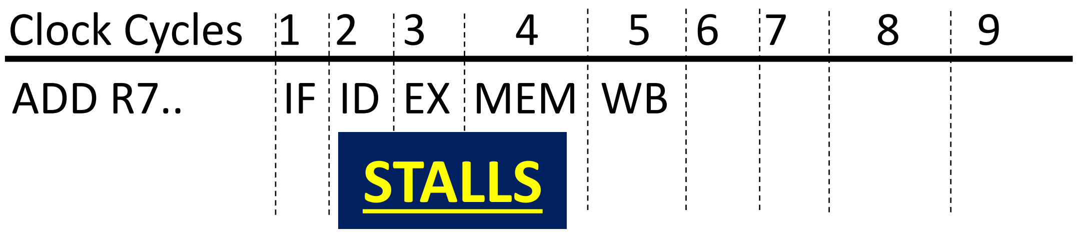

```
...
ADD R7, R8, R9
ADD R10, R7, R8
CHECK_SIGNATURE
```

Clock Cycles	1	2	3	4	5	6	7	8	9
ADD R7..	IF	ID	EX	MEM	WB				
ADD R10..		STALLS			IF	ID	EX	MEM	WB

Multi-Core

Unstable signature – Performance Counters

```
MOV #0, PCs
…
ADD R7, R8, R9
ADD R10, R7, R8
…
MOV PCs, R4
CHECK_SIGNATURE
```



PC_1: PIPELINE STALLS | **N + 3**

Multi-Core

Problem Formulation – Summary

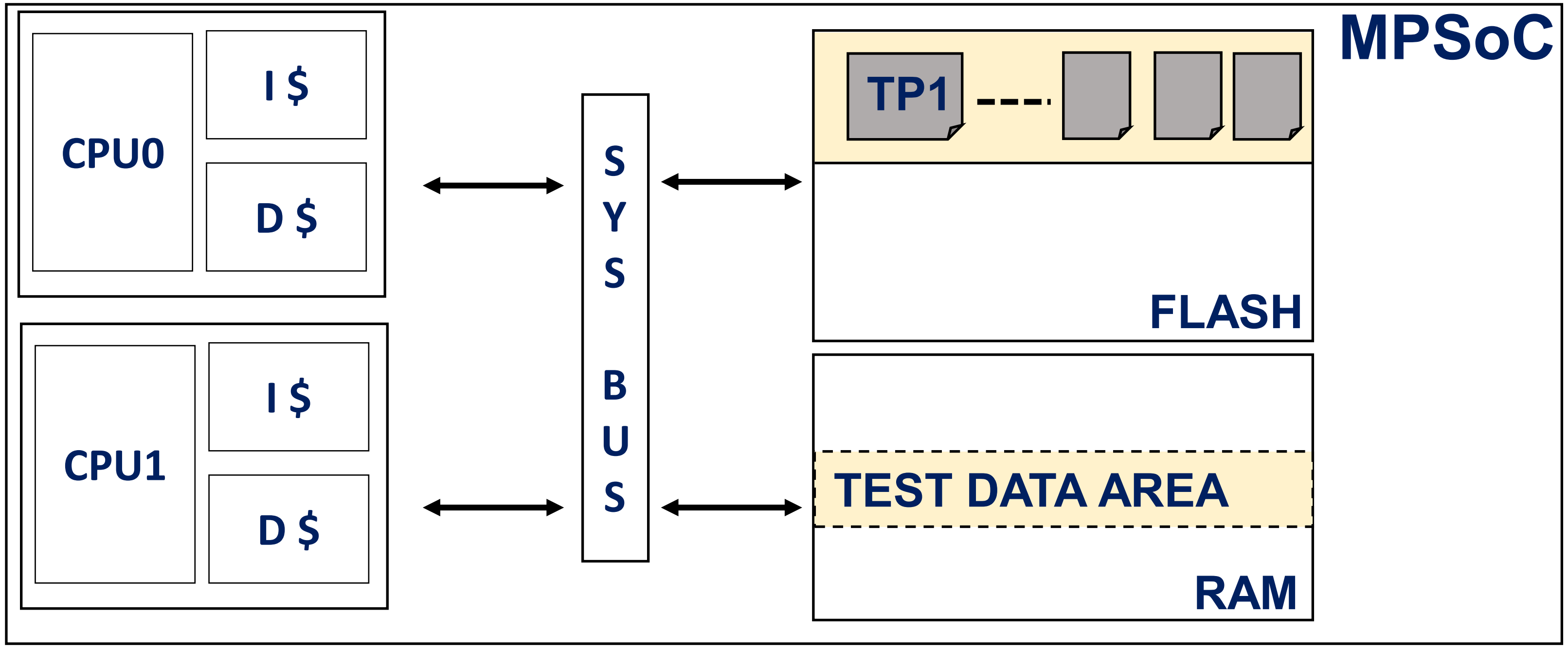
- **Uncertain Fault Coverage**: it varies depending on the whole SoC activity – processor features (fault locations) not correctly excited;

- **Unstable Signature:** mismatch is due to the occurrence of a fault or an altered instructions stream?

Proposed method – Cache memories

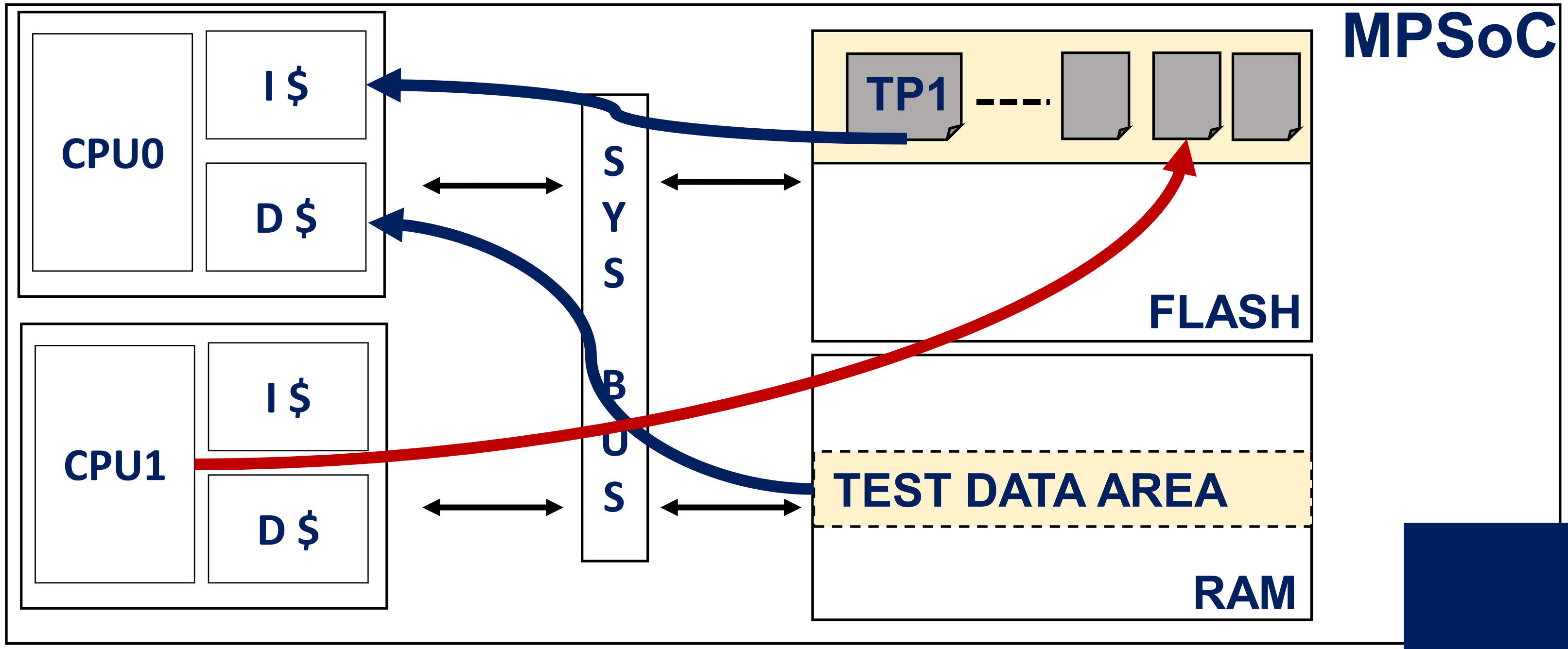
- Exploit cache memories to avoid these issues;
- **Isolate** the self-test procedure execution from the system activities;
- **Apply minimal modifications** to self-test procedures to better exploit locality principles – **deterministic usage of caches**;

Proposed method – Details



Proposed method – Details

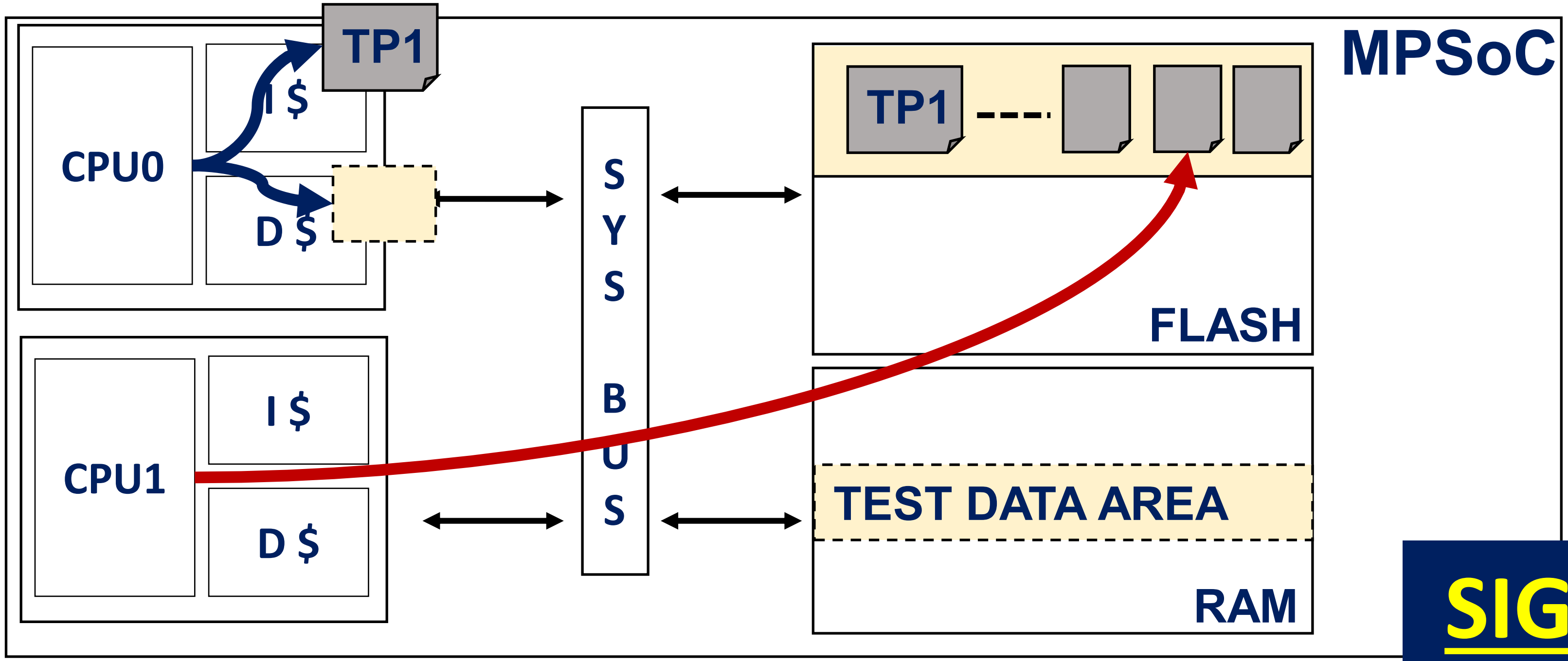
Loading Loop



NO SIGNATURE CHECK!

Proposed method – Details

Execution Loop



SIGNATURE CHECK!

Experimental Results – Uncertain Fault Coverage

- Forwarding mechanism of a heterogeneous Triple-core MPSoC

CORE	# of Faults	FC[%] No Caches	FC [%] With Caches
A	53,298	64.14 – 75.19	79.61
B	57,506	63.61 – 79.59	82.08
C	113,212	56.24 – 66.48	68.79

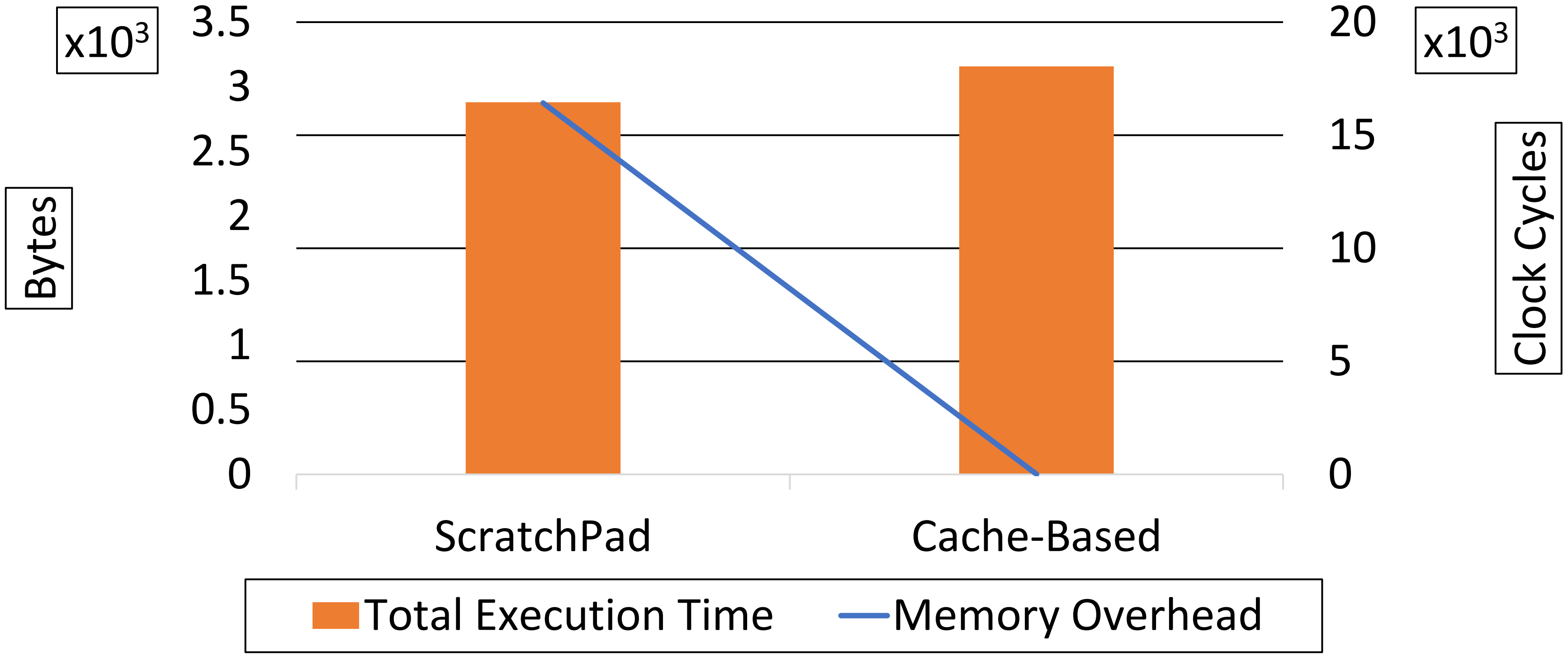
Experimental Results – Uncertain Fault Coverage

- Forwarding mechanism of a heterogeneous Triple-core MPSoC

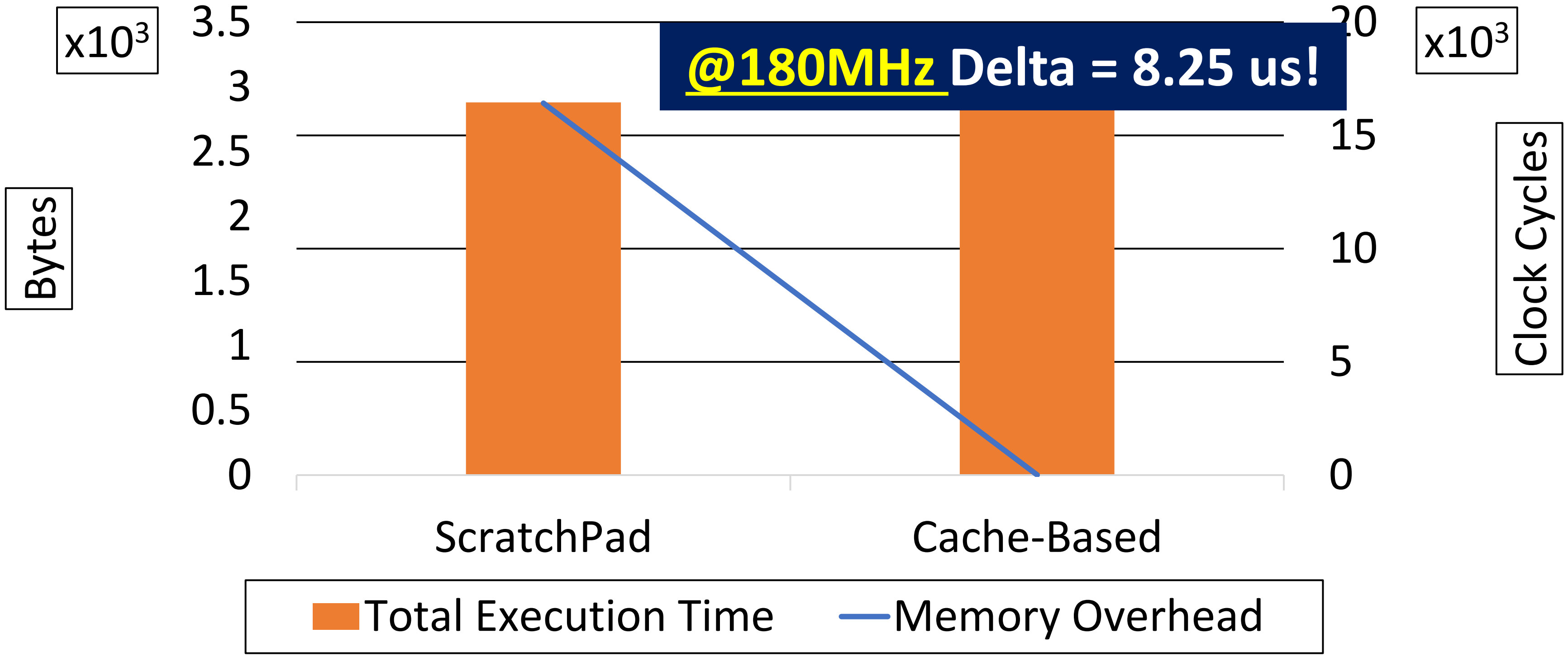
CORE	# of Faults	FC[%] No Caches	FC [%] With Caches
A	53,298	64.14 – 75.19	79.61
B	57,506	63.61 – 79.59	82.08
C	113,212	56.24 – 66.48	68.79

Max Difference Observed: 16%

Comparison with ScratchPad memories



Comparison with ScratchPad memories



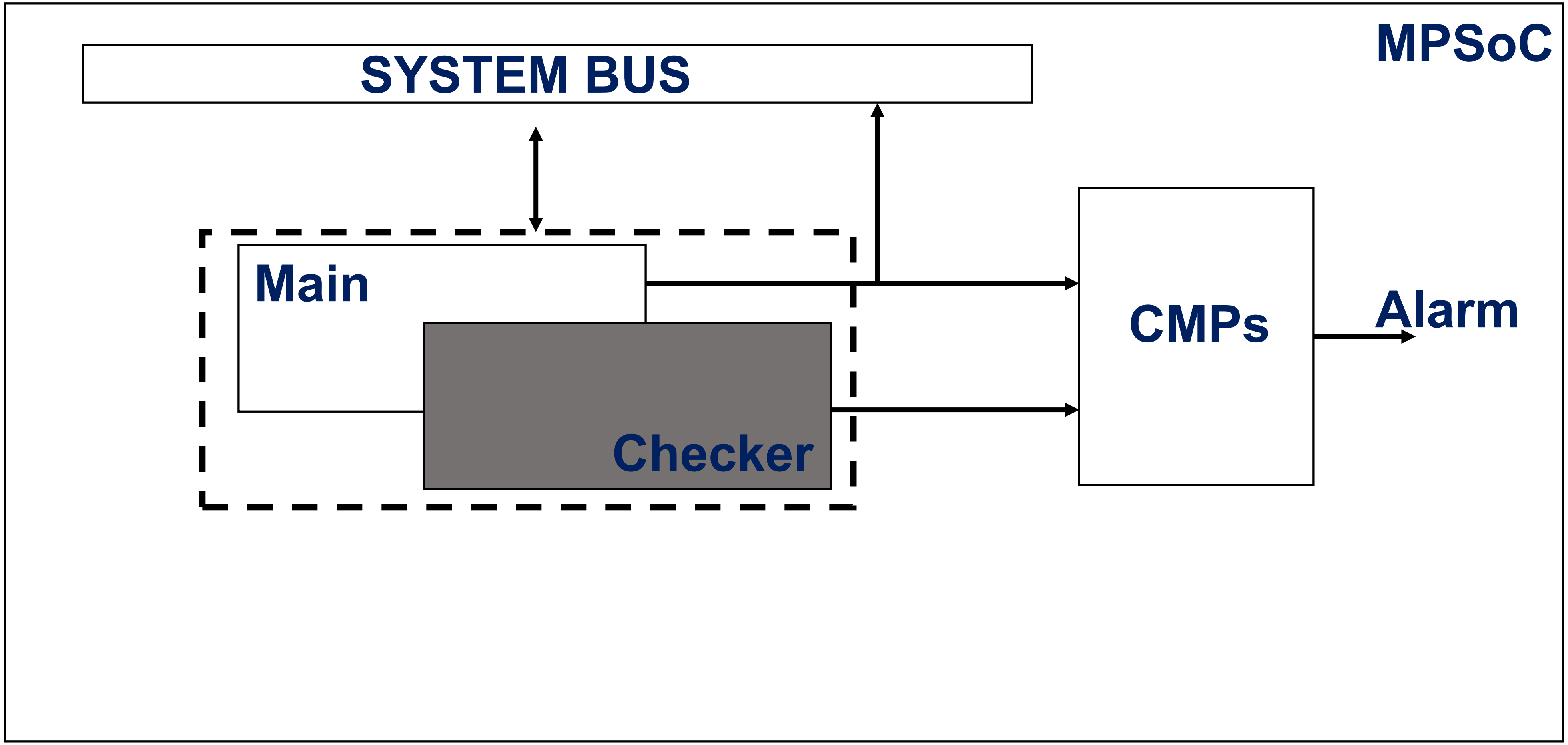
Cache-based execution – Conclusions

- Advantages:
  - **Reusability** of already existing programs (debugged and validated);
  - **Negligible memory penalty**;
  - **No modification of the existing hardware**;
- Drawback:
  - Increased test duration w.r.t ScratchPad memories;
- Future directions: **delay faults**.

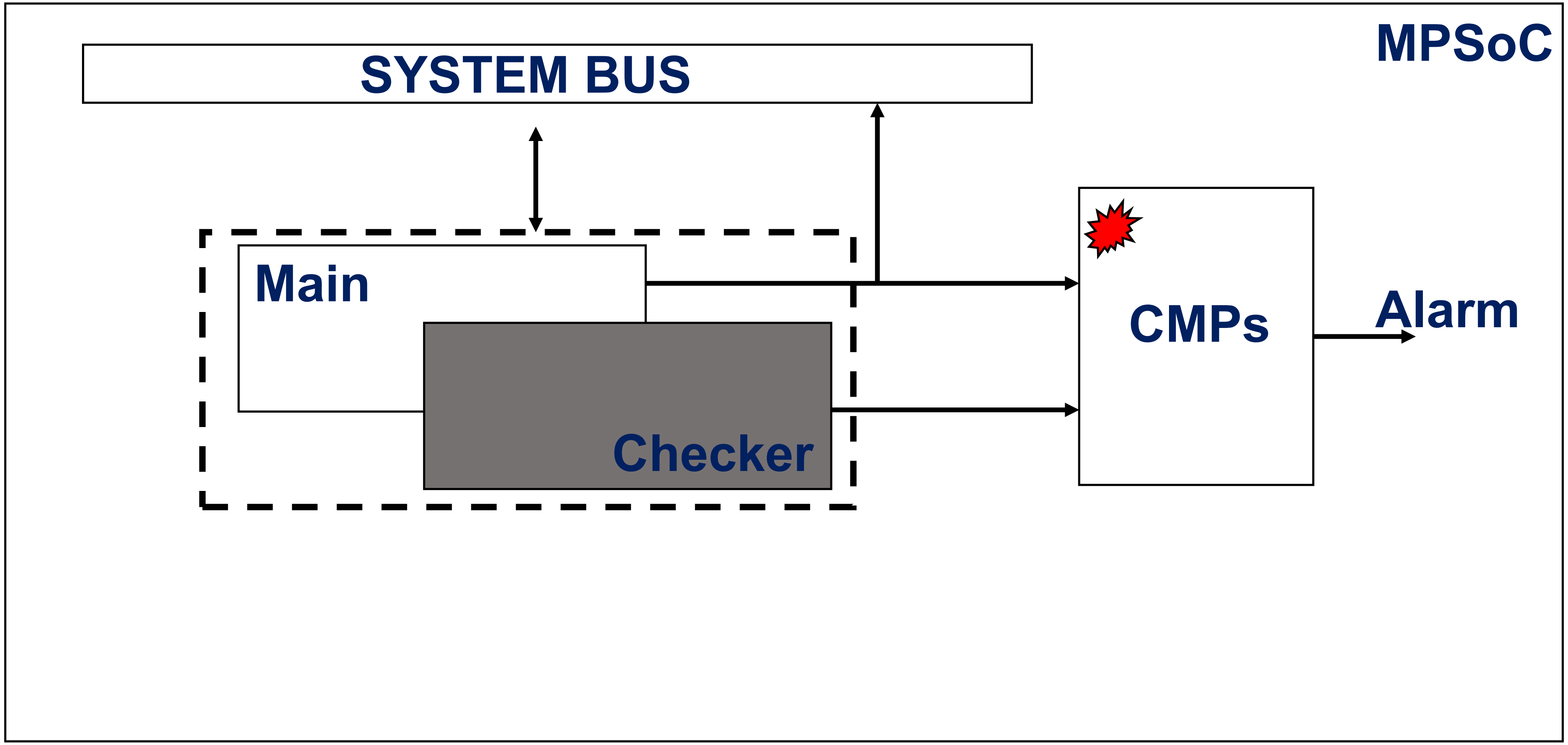
Outline

- Problem Statement
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - Deterministic cache-based execution of Software Test Libraries
  - **Hybrid self-test mechanisms for Lockstep CPUs**
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - JTAG-based fault emulation platform

Dual-Core Lockstep (DCLS) system



DCLS system – Point of Failure



DCLS system comparators in-field test

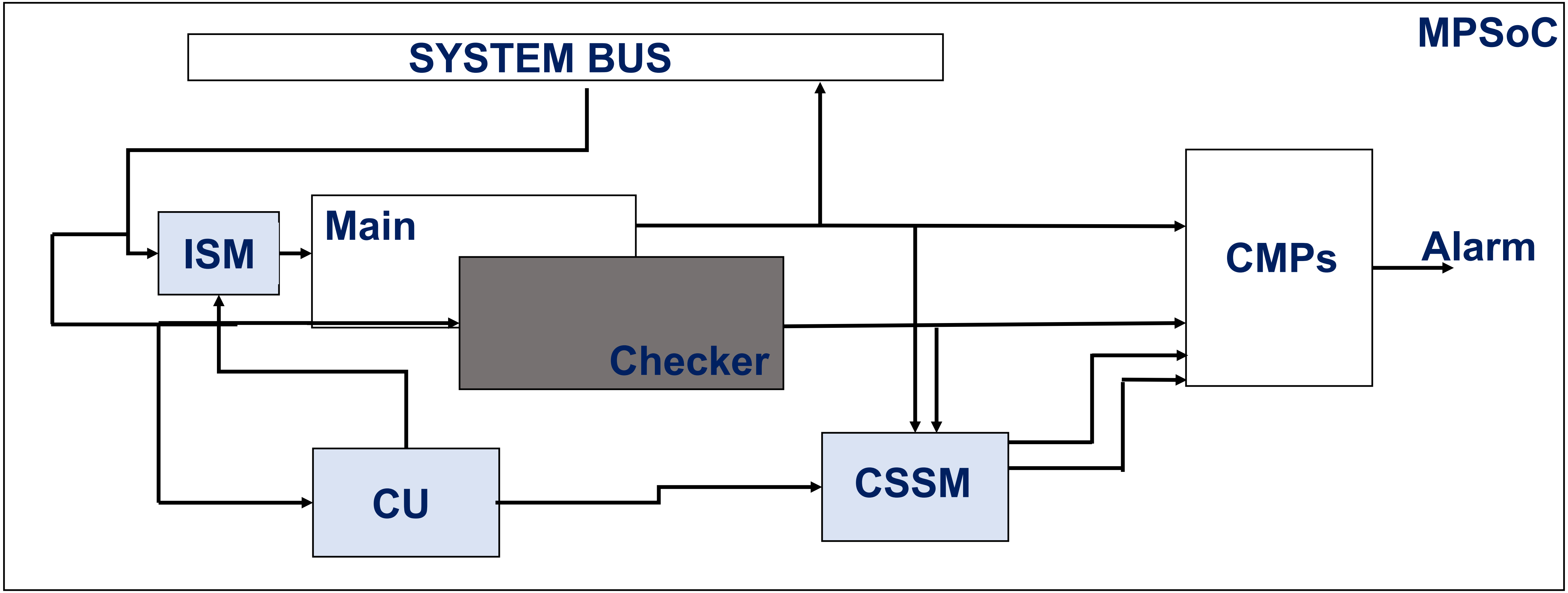
- Permanent faults in comparators might lead **to failures being masked during run-time**;
- Hardware solutions:
  - Time effective;
  - Area overhead;
  - Complete stimuli;
- Software solutions (STL):
  - No area overhead;
  - Limited coverage on comparators.

# pattern	Input A	Input B
1	0111	1111
2	1011	1111
3	1101	1111
4	1110	1111
5	1111	0111
6	1111	1011
7	1111	1101
8	1111	1110
9	1111	1111
10	0000	0000

Proposed approach – Hybrid Self-test

- Software used **for generating test patterns**;
- Hardware (Lockstep Self-test Management Unit, LSMU) oversees:
  - **Altering Main core** instruction stream (Instruction Substitution Module, **ISM**)
  - Direct stimuli application **to control signals comparators** (Control Signal Substitution Module, **CSSM**)
  - Hardware trigged **when specific instruction** is entering the processor (Control Unit, **CU**).

LSMU Architecture



Hybrid solution – Data bus self-test

```
; Program ISM to replace sw 0(r3), r6
; with sw 0(r3), r7
; -----------------------------------------------------------------
;                CHECKER CORE                          MAIN CORE
; -----------------------------------------------------------------
```

CHECKER CORE:

```
LOAD R7, 0xFFFF
LOAD R6, 0xFFFE
SW 0(R3), R6

LOOPx32:
  CALL _WALKING_BIT R6
  SW 0(R3), R6
LOAD R7, 0xFFFF
LOAD R6, 0xFFFE
SW 0(R3), R6

LOOPx32:
  CALL _WALKING_BIT R7
  SW 0(R3), R6
```

MAIN CORE:

```
LOAD R7, 0xFFFF
LOAD R6, 0xFFFE
SW 0(R3), R7

LOOPx32:
  CALL _WALKING_BIT R6
  SW 0(R3), R7
LOAD R7, 0xFFFF
LOAD R6, 0xFFFE
SW 0(R3), R7

LOOPx32:
  CALL _WALKING_BIT R7
  SW 0(R3), R7
```

Experimental Results – DCLS OR1200

Self-test mechanism	Area w.r.t. Lockstep [%]	Coverage [%]	Duration [clock cycles]	Flash Occupation [Bytes]
Hardware	4.47	99.7	500	0
STL	0	72.0	43,976	18,828
Hybrid	2.10	99.5	5,970	4,300

Hybrid self-test – Conclusions

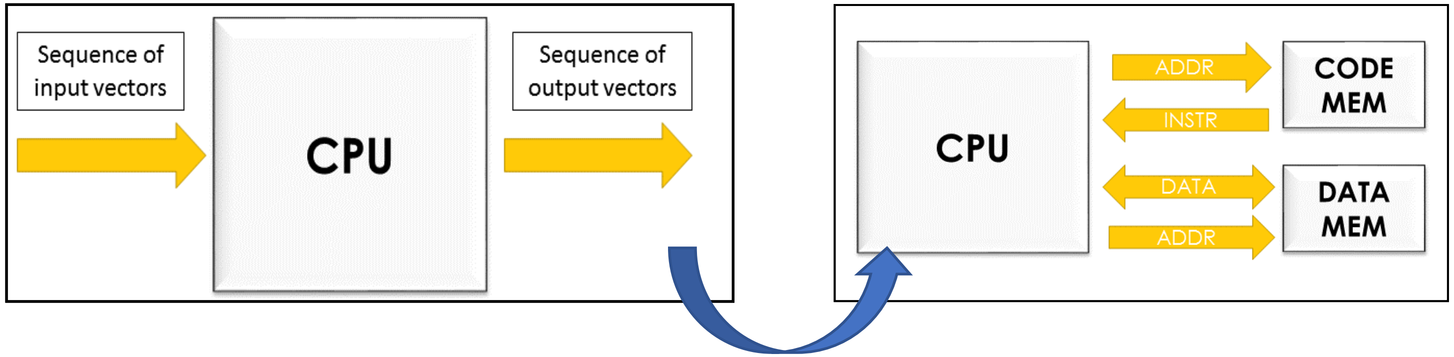
- Hybrid solution halves the area overhead w.r.t a pure hardware-based solution;
- Test patterns are not anymore fixed, and can be updated during device lifetime;
- Future directions: reduce test application time.

Outline

- Problem Statement
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - Deterministic cache-based execution of Software Test Libraries
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - **Functional fault grading for Software Test Libraries**
  - JTAG-based fault emulation platform

Problem Formulation – Fault grading of STLs

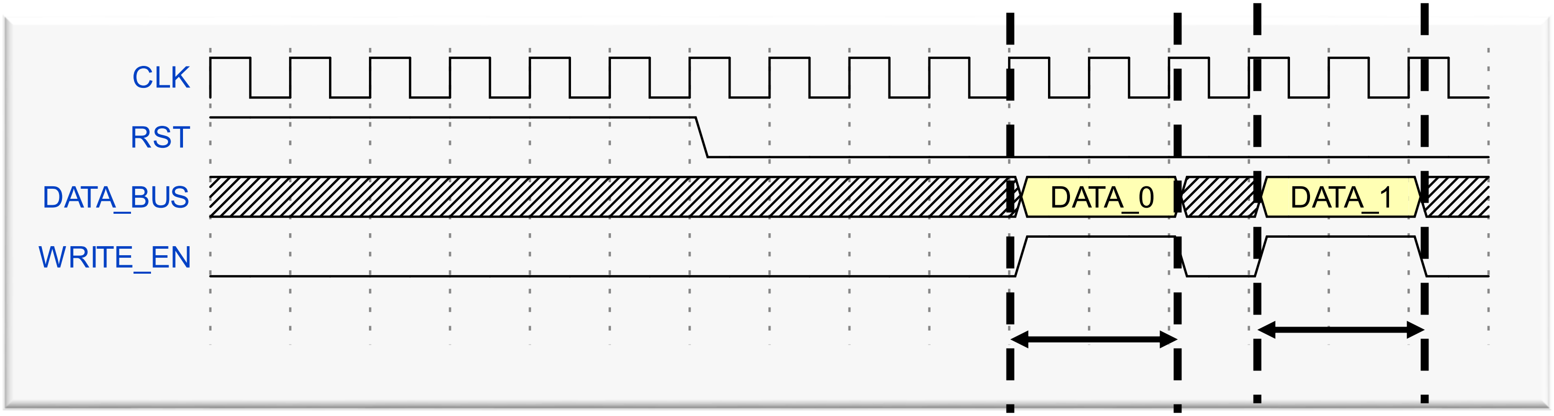
- Fault Grading of self-test mechanism represents a major bottleneck when the complexity of the system increases;
- Critical for STLs development – **lot of fault simulations**;
- From classical sequential circuit fault simulation (**fast**) to Functional fault simulation (**slow**).



Functional fault simulation concepts – Observability

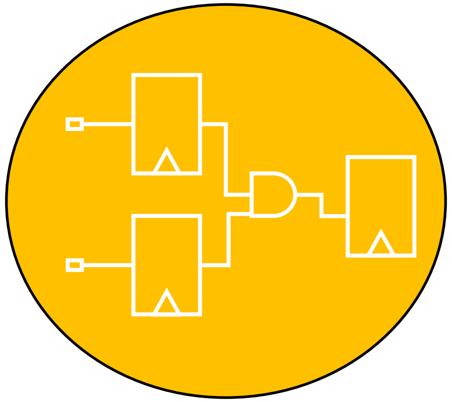
- To grade a self-test procedure, **observability selection** plays a key role:
  - **Which** signals to observe;
  - **When** to observe such signals.

Observation window

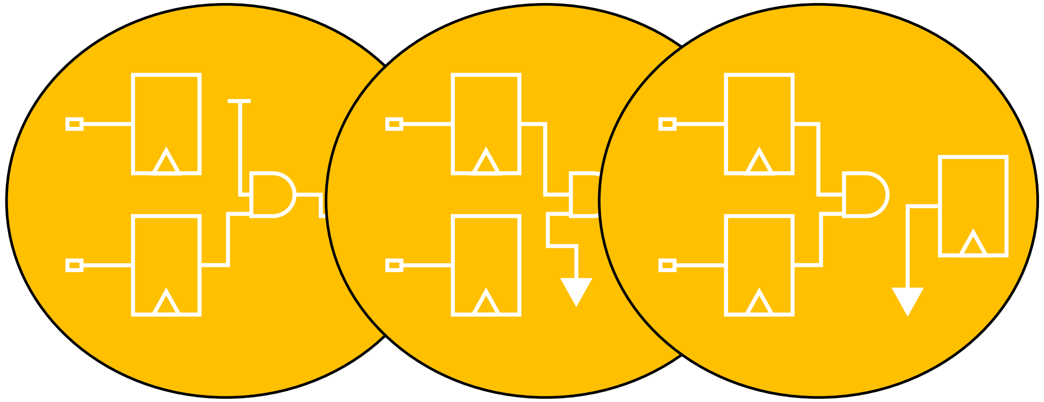


Functional fault simulation concepts – Fault Dropping

- Fault dropping: reduce computational effort.



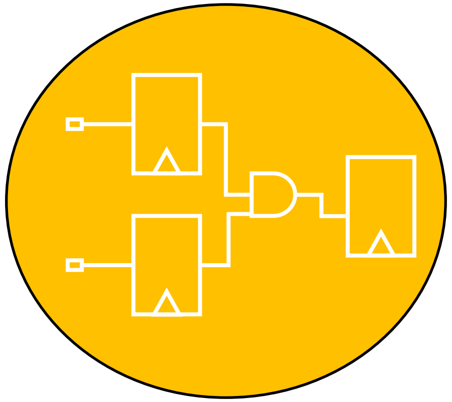
1 Golden Machine



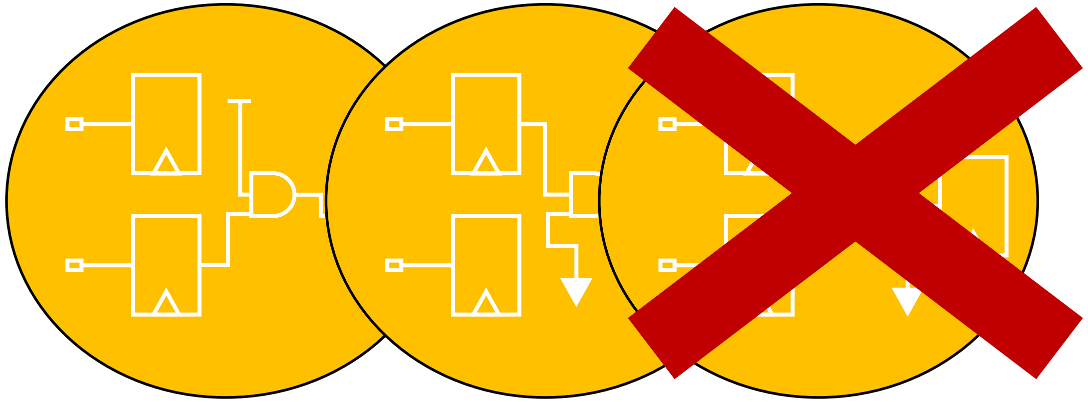
N Faulty Machine

Functional fault simulation concepts – Fault Dropping

- Fault dropping: reduce computational effort.



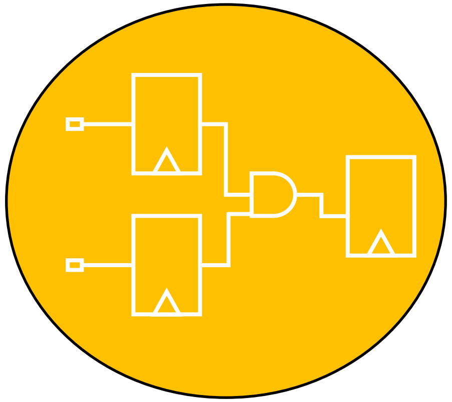
1 Golden Machine



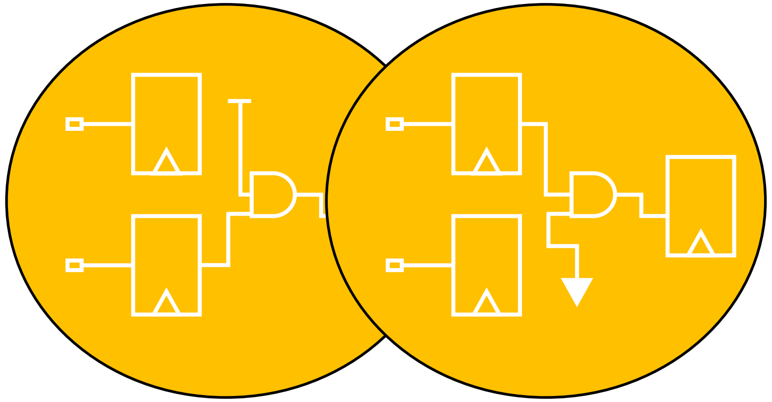
N Faulty Machine

Functional fault simulation concepts – Fault Dropping

- Fault dropping: reduce computational effort.



1 Golden Machine



N - 1 Faulty Machine

Basic Functional fault simulation

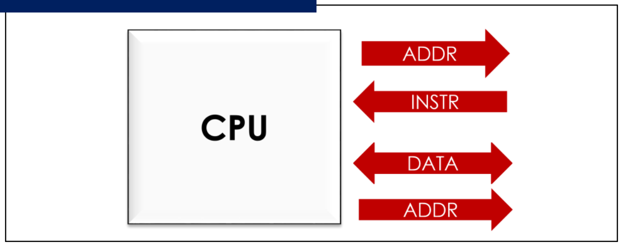
- Observability selection: check memory content (e.g., test signature) **at the end of self-test program execution**;
- Fault dropping **not exploited at all** ➔Huge run time!

- Set of techniques to be used during **the entire STL development flow;**
- Based on optimal placement of observation windows to enable fault dropping (trading off execution time for accuracy).

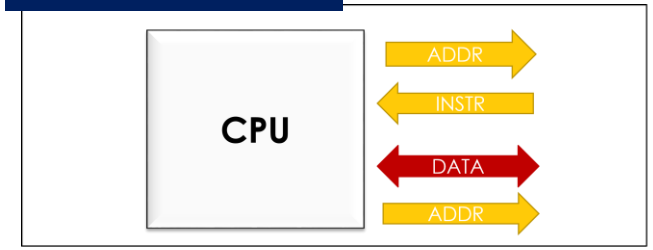
Self-Test Program Fault Simulations (STP-FSIMs)

- Basic techniques:

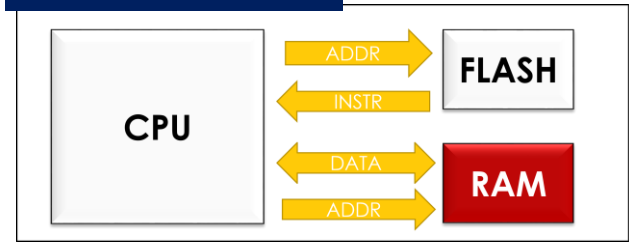
STP-FSIM0



STP-FSIM1

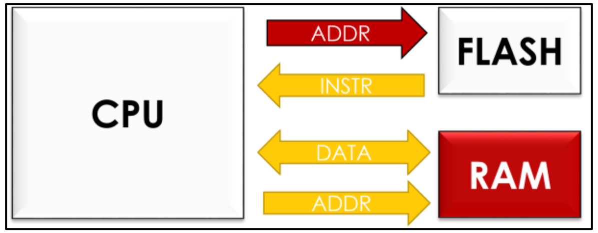


STP-FSIM2

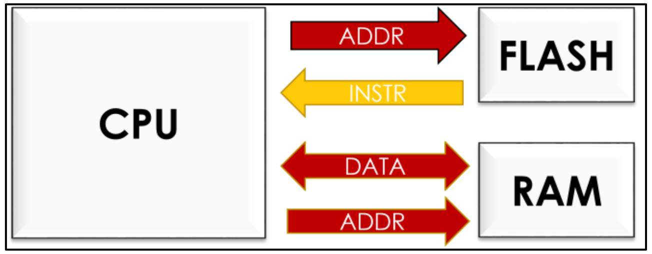


- Optimized techniques:

STP-FSIM3



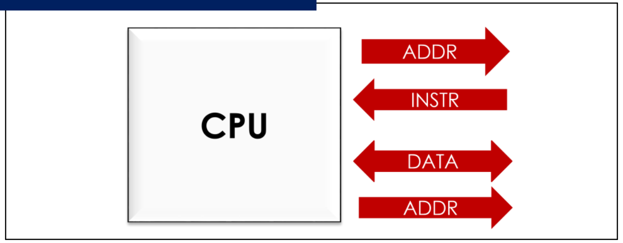
STP-FSIM4



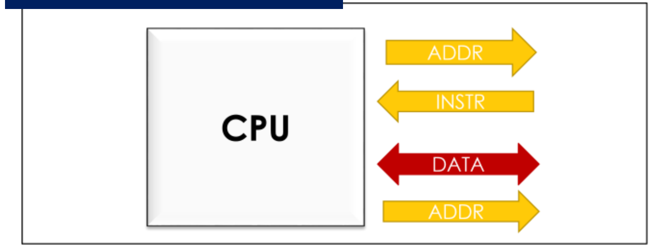
Self-Test Program Fault Simulations (STP-FSIMs)

- Basic techniques:

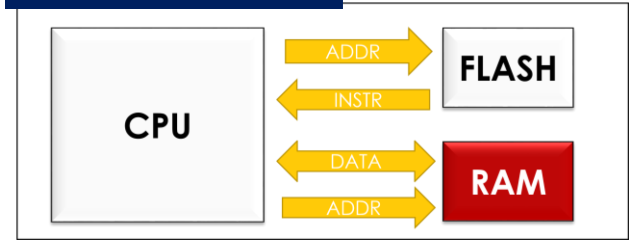
STP-FSIM0



STP-FSIM1

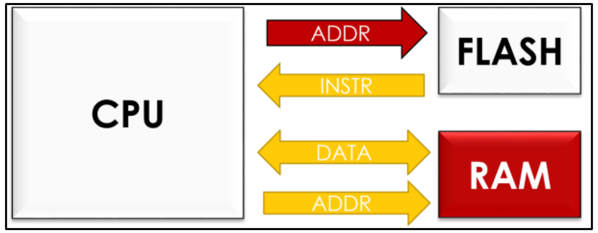


STP-FSIM2

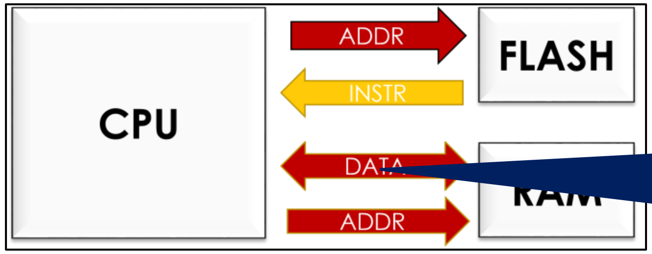


- Optimized techniques:

STP-FSIM3



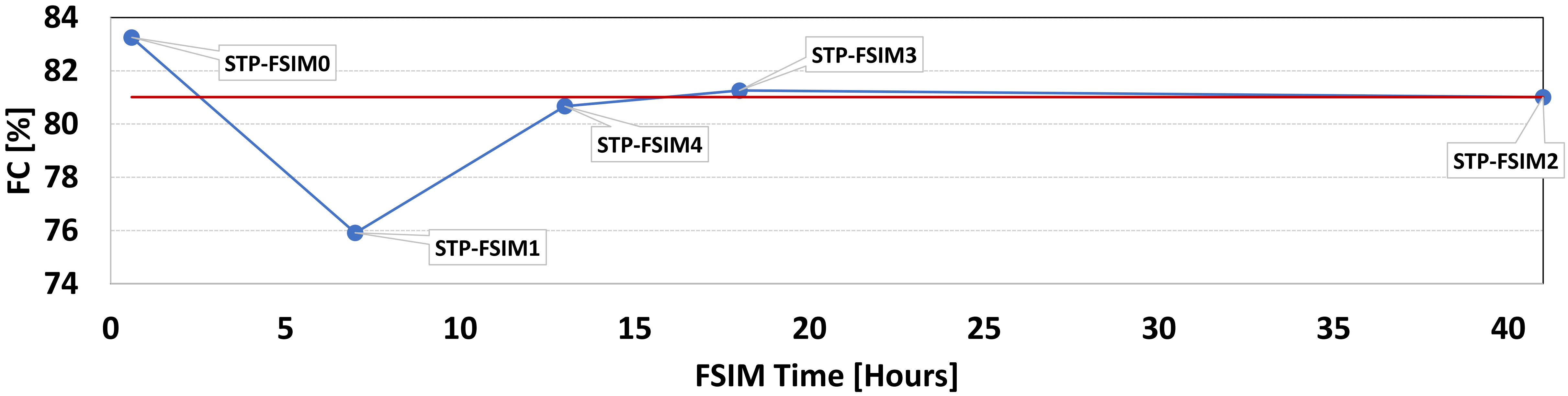
STP-FSIM4



Observed when accessing **DATA MEM ONLY**

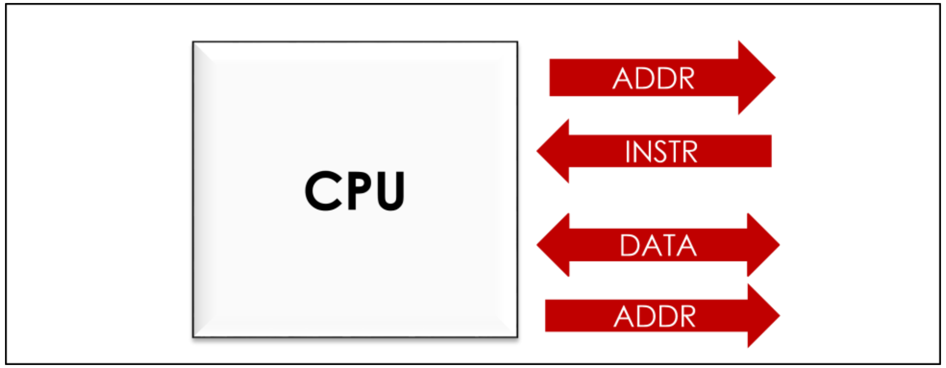
Experimental Results – STP-FSIMs on OR1200

- Functional Fault simulation time greatly reduced: 56-68%;
- For optimized techniques, limited loss of accuracy in the final fault coverage;



Fault grading of STLs for DCLS

- STL for lockstep: check occurrence of faults **in exclusively one of the two domains** (i.e., either Main or Checker);
- Faults **detected by downstream comparators – signature not required**;
- STP-FSIM0 (basic sequential fault simulation) models this behavior – can be used **without any loss of coverage**.



Fault grading of STLs – Conclusions

- STP-FSIMs to be used in different phases of STL development:
  - Quickest methods for early phases;
  - Longest for final grading;

- In case of DCSL, the quickest (STP-FSIM0) can be always used.

Outline

- Problem Statement
- On-line self-test mechanisms
  - Software Scheduler for Software Test Libraries
  - Deterministic cache-based execution of Software Test Libraries
  - Hybrid self-test mechanisms for Lockstep CPUs
- Improvements of functional fault grading methodologies
  - Functional fault grading for Software Test Libraries
  - **JTAG-based fault emulation platform**

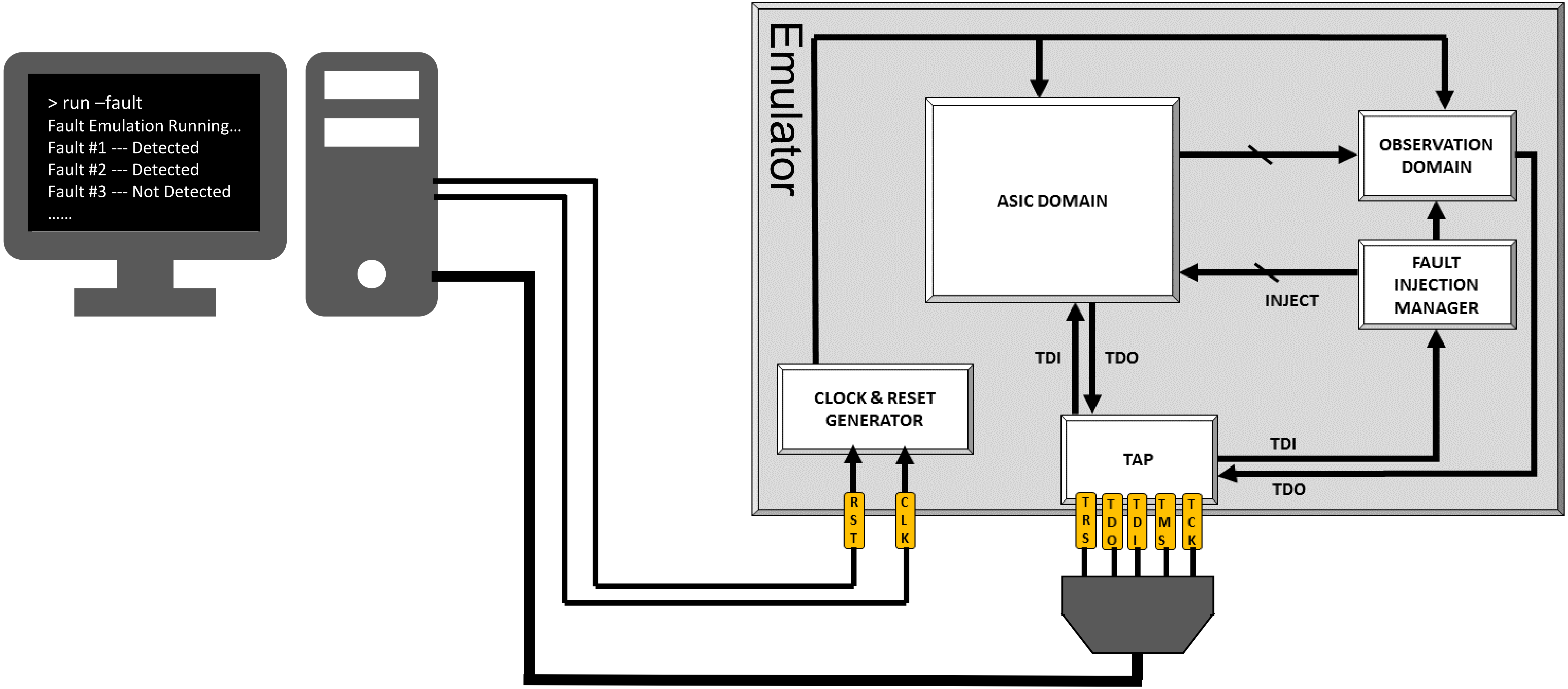
Fault Emulation – Problem Formulation

- To address limitations of fault simulation, **emulation** can be exploited;
- **Applicable not only to STLs**;
- Research community focused on how to inject faults;
- Focus:
  - **how to (efficiently) observe** to cope with slow external interfaces;

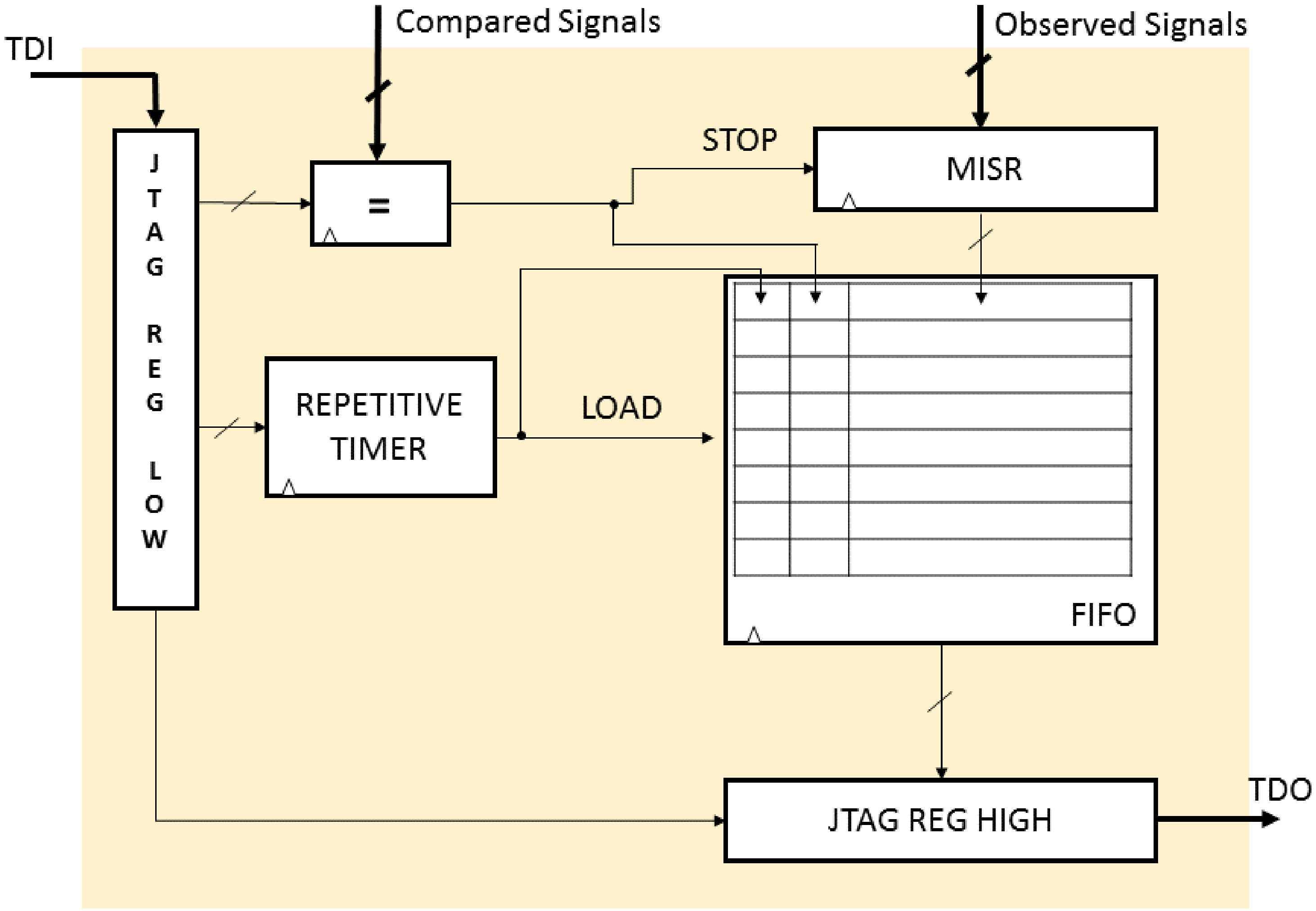
Proposed fault emulation platform

- **JTAG wire-wise compatible** – virtually 0 pins added to the design;
- Allows **for periodic fault dropping in hardware**:
  - Stop emulation as soon as mismatch is detected on the outputs – reduced fault emulation time;

The platfrom



Observation Domain



Implementation details

- Gate-level 8051 MCU (65nm CMOS technology) – instrumented for stuck-at faults (50k) injections;
- Workload: Fibonacci series (~2k clock cycles)
- Emulation time: 88 seconds with fault dropping (90 without);
- FPGA Zynq 7000 Xilinx utilization:
  - LUT: ~53% (~12% without instrumentation);
  - Flip-flops: ~4.5% (~1.2% without instrumentation);
- Observation domain:
  - LUT ~2%, FFs ~1% (with 32x34 FIFO).

FPGA-based emulation – Conclusions

- MISR and on-chip FIFO ideal for coping with slow external interfaces;
- Effectiveness limited mainly due to the short benchmark considered;
- Future directions:
  - Further benchmarking;
  - Automatize fault detection directly in hardware with dedicated FSM (programmable via JTAG).

Thesis Conclusions

- STLs parallel execution in MPSoCs:
  - Multiple shared resources;
  - Both heterogeneous/homogeneous MPSoCs;
  - Uncertain fault coverage and unstable signature never reported elsewhere;
- Hybrid approaches to the on-line self-test: merge the best of two worlds;
- Fault grading of self-test mechanisms – overcome limitations of simulation-based approaches via hardware emulation.

Thank you for your attention!

Backups

Decentralized Selfish Scheduler – DSS

Decentralized Selfish Scheduler – DSS

DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {TP2, TP3}

DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {TP2, TP3}

Decentralized Selfish Scheduler – DSS

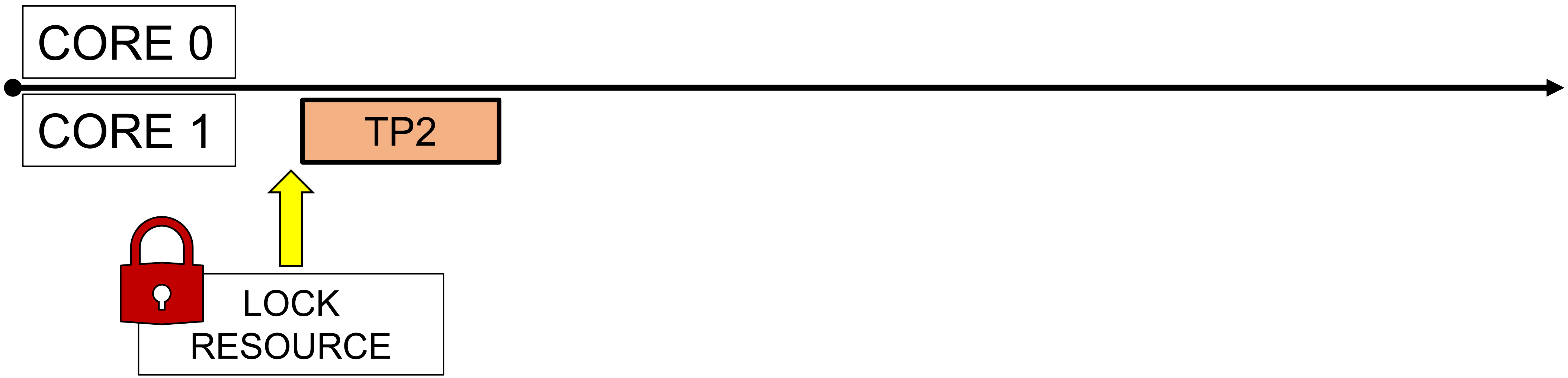
DSS CORE 1

TestTable = {**TP2, TP3**, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {**TP2, TP3**}

DSS CORE 0

TestTable = {**TP2, TP3**, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {**TP2, TP3**}

Decentralized Selfish Scheduler – DSS



DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP3, TP1 , TP4, TP5}
ShareResource = {TP2, TP3}

DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {TP2, TP3}

Decentralized Selfish Scheduler – DSS

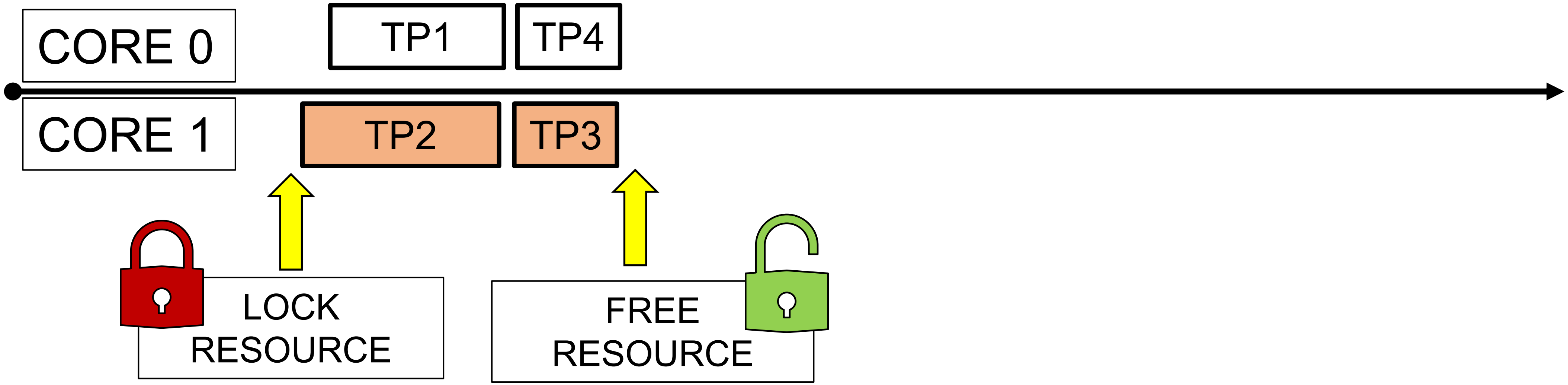
DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP1 , TP4, TP5}
ShareResource = {TP2, TP3}

DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP4, TP5}
ShareResource = {TP2, TP3}

Decentralized Selfish Scheduler – DSS



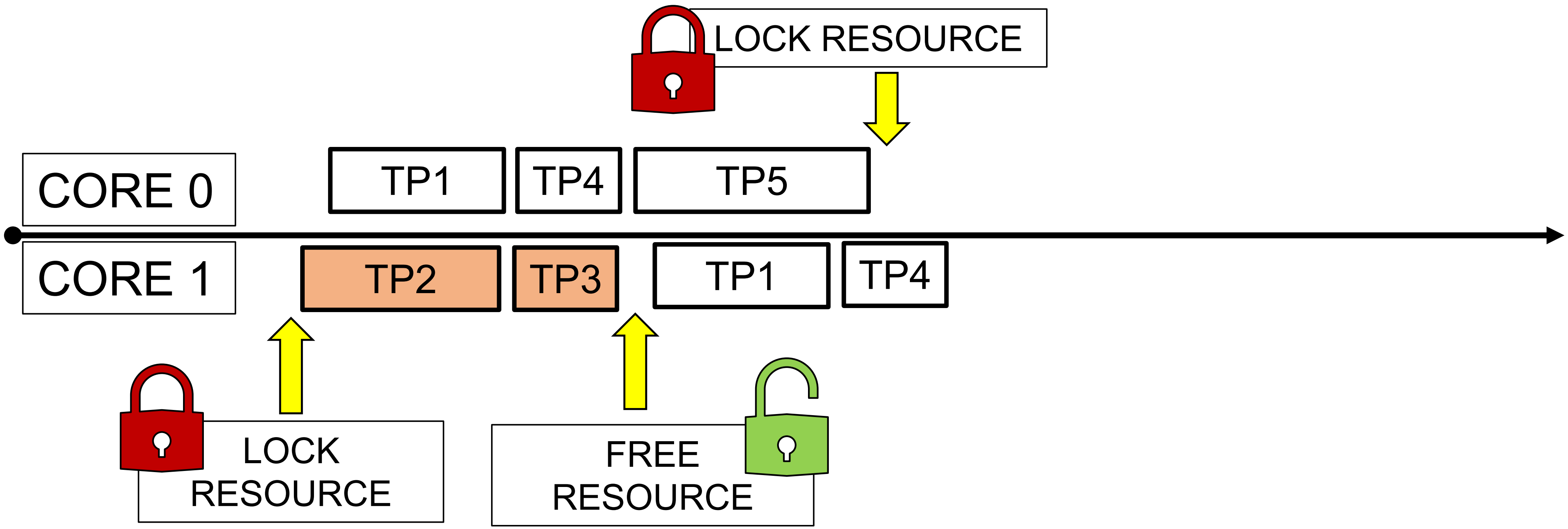
DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP1, TP4, TP5}
ShareResource = {TP2, TP3}

DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3, TP5}
ShareResource = {TP2, TP3}

Decentralized Selfish Scheduler – DSS



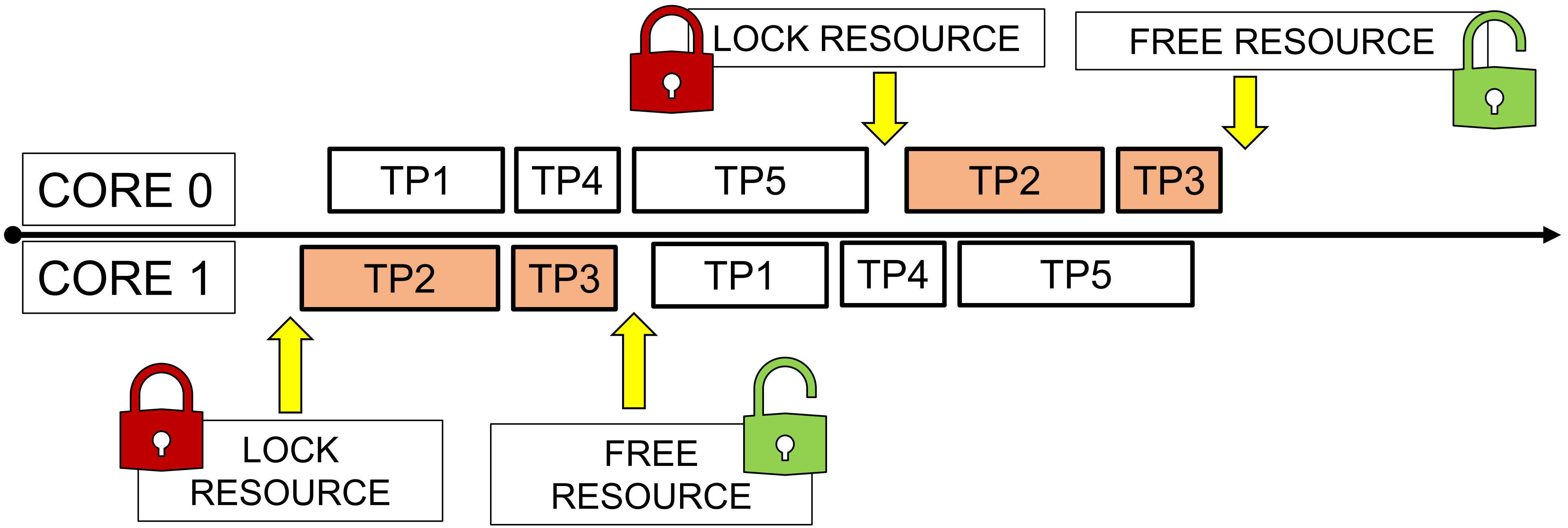
DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP5}
ShareResource = {TP2, TP3}

DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {TP2, TP3}
ShareResource = {TP2, TP3}

Decentralized Selfish Scheduler – DSS



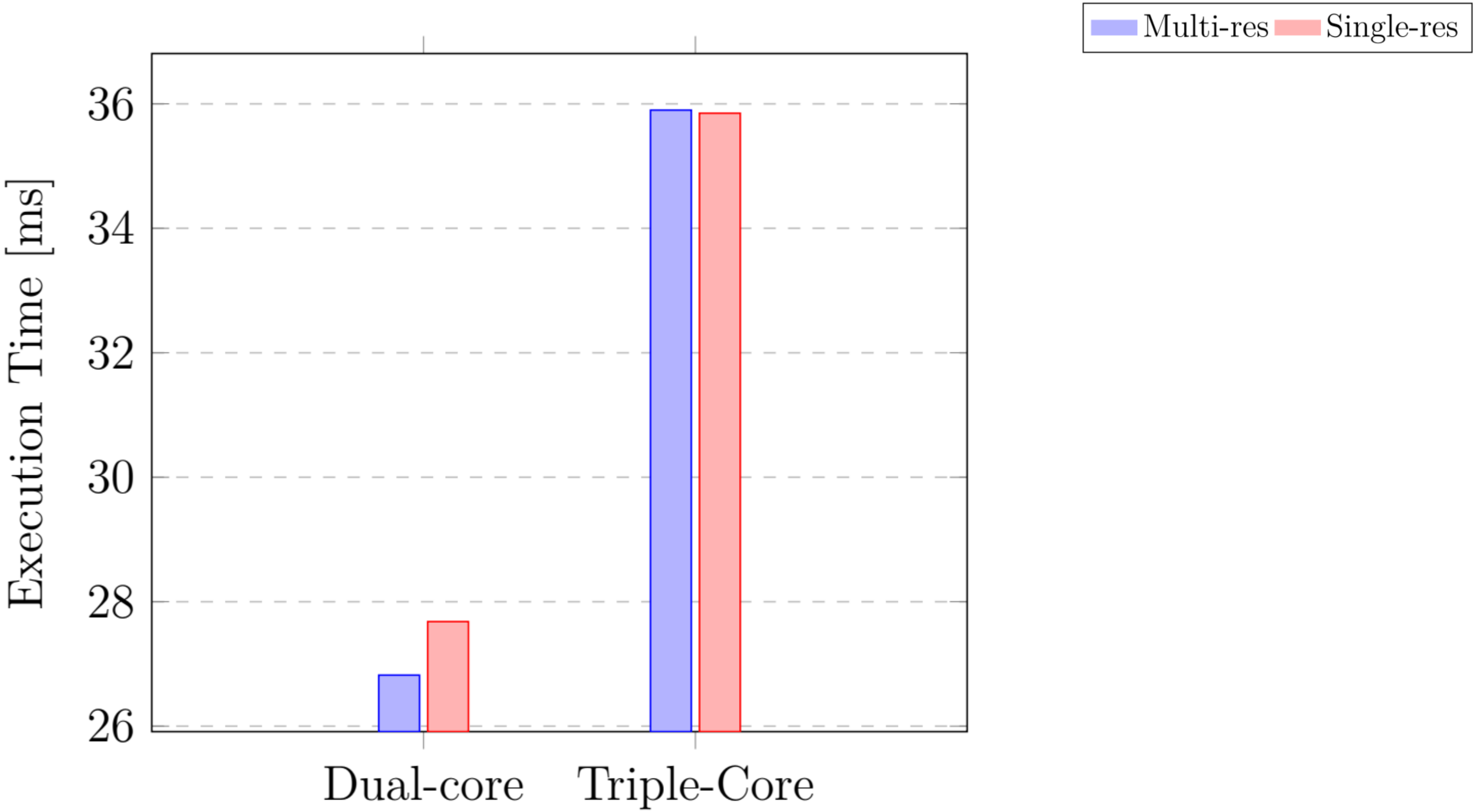
DSS CORE 1

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {}
ShareResource = {TP2, TP3}

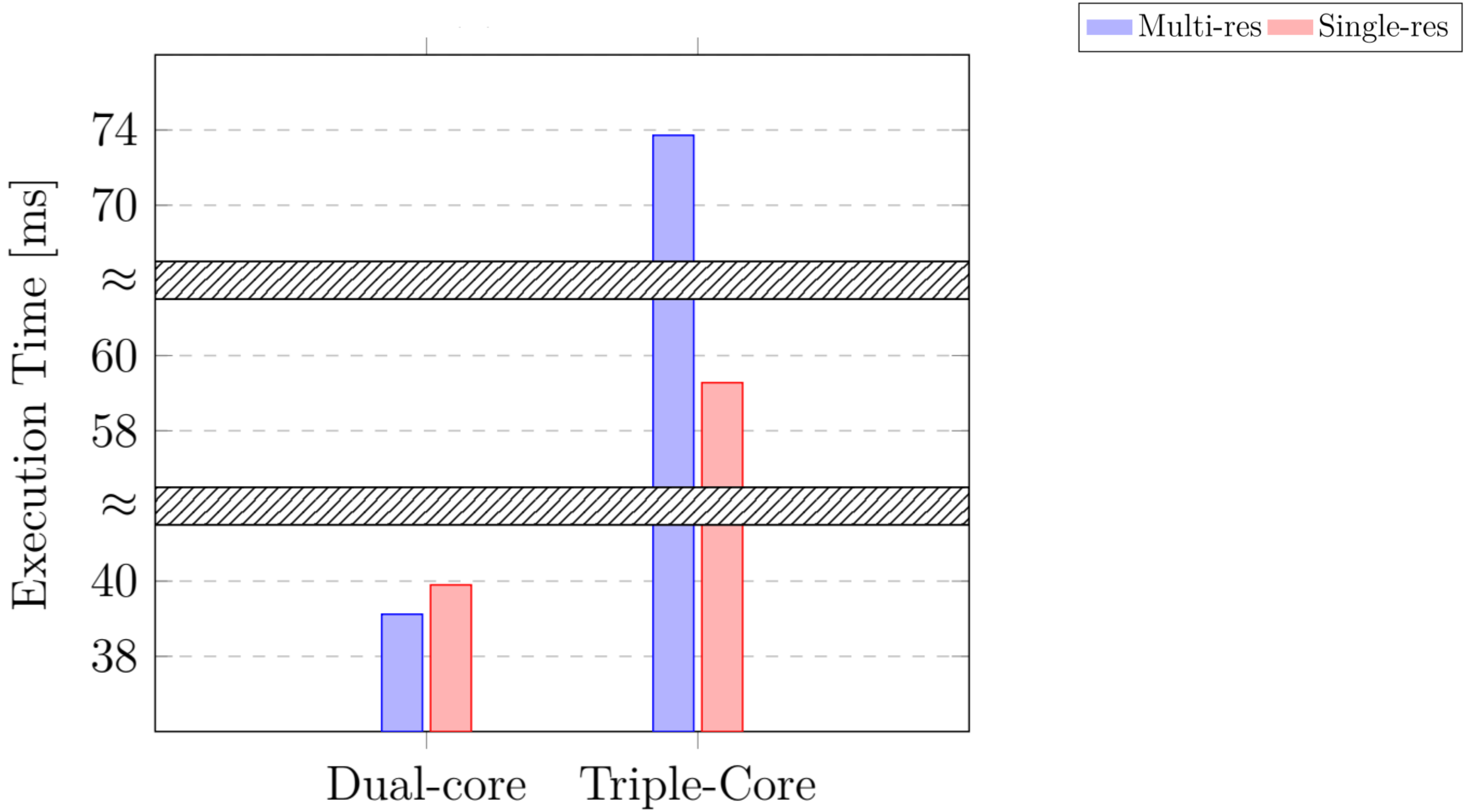
DSS CORE 0

TestTable = {TP2, TP3, TP1, TP4, TP5}
PendingList = {}
ShareResource = {TP2, TP3}

Multi-resource heterogenous MPSoC



Multi-resource homogeneous MPSoC



Proposed approach – Hybrid self-test

- **Hardware-assisted software self-test** of comparators;
- Exploit software flexibility combined with specialized hardware;
- Possibly trade-off area savings at the expenses of execution time.

Software Scheduler for STLs – Challenges

- Boot-time tests create parallelization difficulties due to shared resources (e.g., the shared portion of system RAM):