Dependable Systems

Design for dependability: HW / SW hardening

Luca Cassano
luca.cassano@polimi.it
cassano.faculty.polimi.it/ds.html

Most of the material of these slides has been provided by Prof. Cristiana Bolchini, Politecnico di Milano, Italy

TOPIC QUESTIONS

How to harden computing systems?

What hardware/software techniques are available?

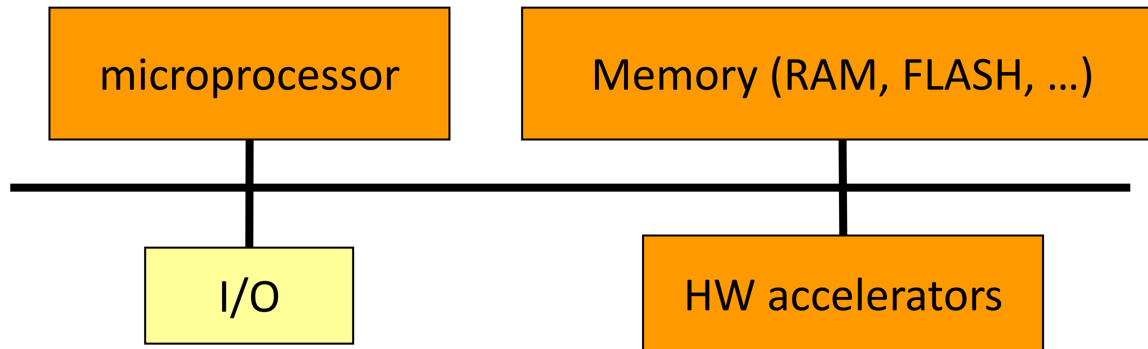
How can they be applied? Can they be applied?

What coverage do they offer?

How difficult it is to apply them?

Working Scenario

The considered hw/sw system architecture



Working Scenario | 2

Advanced architectures:

- Multi-processor systems
  - Several processors are connected through a common communication channel to a shared memory
- Distributed systems
  - Several nodes (processor, private memory and bus) are connected through a message-exchange bus

Working Scenario | 2

Advanced architectures:

- Multi-processor systems
  - Several processors are connected through a common communication channel to a shared memory
- Distributed systems
  - Several nodes (processor, private memory and bus) are connected through a message-exchange bus

Other kinds of contexts:

- Edge/Cloud/Fog computing
- Internet of Things (IoT)
- Cyber-Physical Systems (CPS)

Working Scenario | 3

Software functional structure

- Set of tasks that could be partitioned into two portions: **critical section** and **non-critical section**
  - Critical section: execution “area” producing sensible results for the system dependability
    - Worst case: all the tasks are included in the critical section

Working Scenario | 3

Software functional structure

- Set of tasks that could be partitioned into two portions: **critical section** and **non-critical section**
  - Critical section: execution “area” producing sensible results for the system dependability
    - Worst case: all the tasks are included in the critical section
- Depending on the system, tasks are mapped and scheduled at design-time or at run-time

Working Scenario | 3

Software functional structure

– Set of tasks that could be partitioned into two portions: **critical section** and **non-critical section**
  - Critical section: execution "area" producing sensible results for the system dependability
    – Worst case: all the tasks are included in the critical section
– Depending on the system, tasks are mapped and scheduled at design-time or at run-time

Hypothesis: the code is bug free

– A fault (temporary or permanent) affects the hardware and detection/tolerance is performed by acting both on the hardware and the software

Different approaches

System architectures can be hardened at different levels of abstraction

- Hardware level
- Architecture level
- Process level
- Software instruction level

- … a mix of the above approaches

Different approaches | 2

Acting at lower level

- Lower error detection latency
- More diagnosis information
- Simpler recovery

Different approaches | 2

Acting at lower level

- Lower error detection latency
- More diagnosis information
- Simpler recovery

Acting at higher level

- More flexible solution
- Reduced design cost and complexity
- Possibility to exploit COTS components

Hardware-level approaches

The processor structure is internally redesigned by applying hardening techniques (fault detection / tolerance)

Hardware-level approaches

The processor structure is internally redesigned by applying hardening techniques (fault detection / tolerance)

Specific techniques are selected w.r.t. the type of fault to deal with

- Permanent vs. transient

Hardware-level approaches

The processor structure is internally redesigned by applying hardening techniques (fault detection / tolerance)

Specific techniques are selected w.r.t. the type of fault to deal with

- Permanent vs. transient

The various units of the processor are hardened independently

- Functional units (ALU, fetch unit, ...)
- Register files and memories

Hardware-level approaches | 2

Hardening of the functional units (ALU, fetch unit, ...)

- Space redundancy is mainly used (DWC,TMR)
- Arithmetic codes is a viable approach for specific functional units (E.g.: residual codes for ALU)

Hardware-level approaches | 2

Hardening of the functional units (ALU, fetch unit, ...)

- Space redundancy is mainly used (DWC,TMR)
- Arithmetic codes is a viable approach for specific functional units (E.g.: residual codes for ALU)

Hardening of register files and memories

- Information redundancy (E.g.: EDC, ECC)

Hardware-level approaches | 2

Hardening of the functional units (ALU, fetch unit, ...)

- Space redundancy is mainly used (DWC,TMR)
- Arithmetic codes is a viable approach for specific functional units (E.g.: residual codes for ALU)

Hardening of register files and memories

- Information redundancy (E.g.: EDC, ECC)

An example of application of such approach is the **Leon2-FT** produced by **Gaisler** for **ESA**: a SEU tolerant microprocessor where FFs are protected by Triple Modular Redundancy and all internal and external memories are protected by error correction codes or parity bits.

Architecture-level hardening

Space Redundancy

The whole processor is replicated and its outputs are checked/voted

Some approaches:

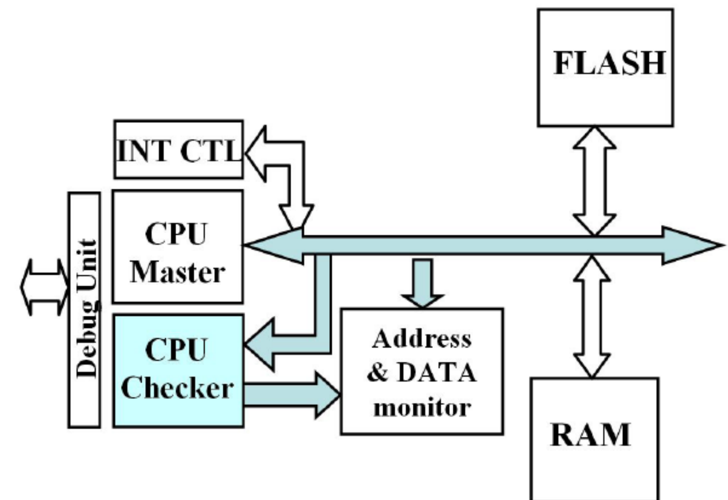
- Fault detection
  - Lock-Step Dual Processor
  - Loosely-Synchronized Dual Processor
  - Watchdog processor
- Fault tolerance
  - TMR – Triple Modular Redundancy
  - Dual Lock-Step Architecture

Architecture-level hardening | 2

Space Redundancy

Lock-Step Dual Processor

- Two processors execute the same code being strictly synchronized
- Bus and memories are protected with codes
- The interrupt controller is specifically designed with fault detection mechanisms

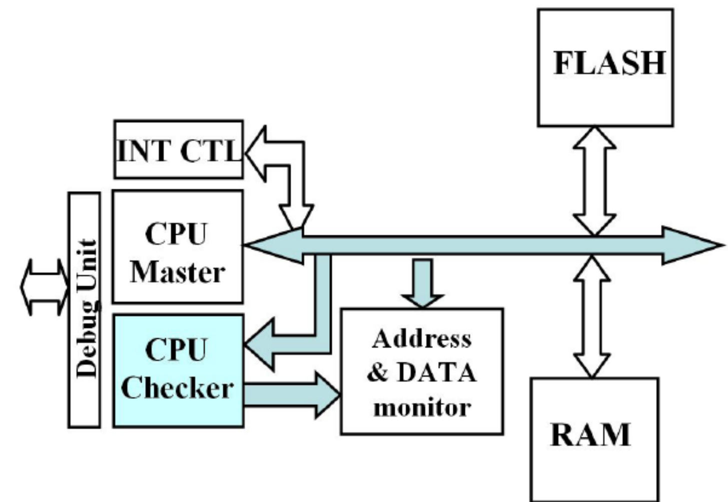


Architecture-level hardening | 2

Space Redundancy

Lock-Step Dual Processor

– Two processors execute the same code being strictly synchronized
– Bus and memories are protected with codes
– The interrupt controller is specifically designed with fault detection mechanisms



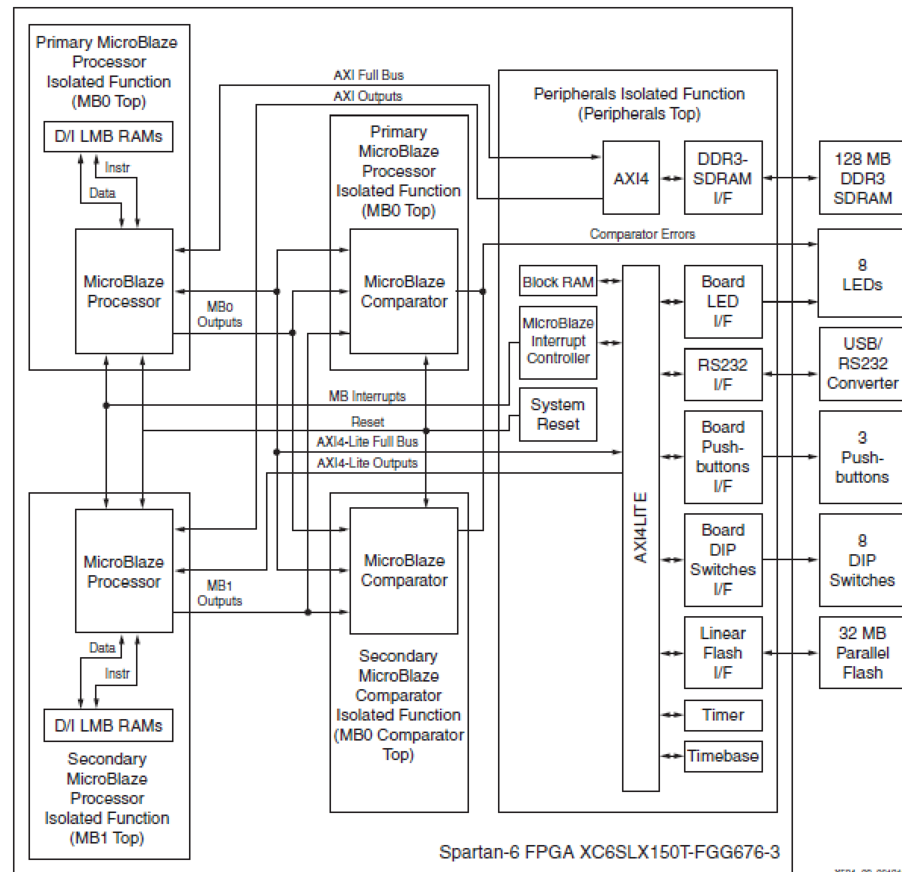
The solution is called fail-silent architecture (corrupted data are not emitted)

– Used as basic element for fault-tolerant distributed systems

Architecture-level hardening | 3

Space Redundancy

Example of Lock-Step Dual Processor: Xilinx Dual Lock-Step Processor

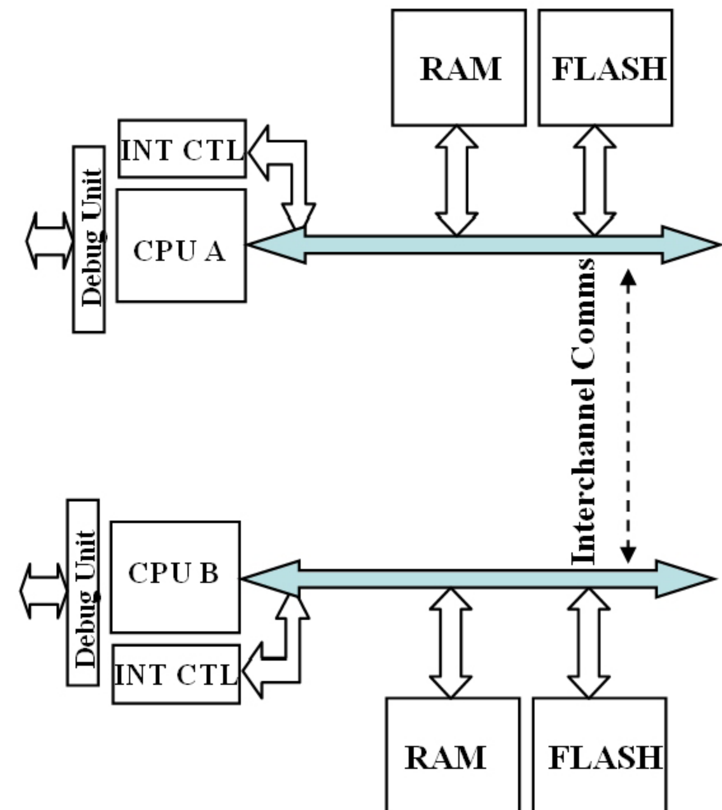


Architecture-level hardening | 4

Space Redundancy

Loosely-synchronized dual processor

- Two processors run independently
- The operating system is devoted to inter-process communication, synchronization and error detection



Architecture-level hardening | 5

Space Redundancy

Loosely-synchronized dual processor (cont.)

- Synchronization mechanisms must be protected with specific hardware/software mechanisms

Architecture-level hardening | 5

Space Redundancy

Loosely-synchronized dual processor (cont.)

- Synchronization mechanisms must be protected with specific hardware/software mechanisms
- After an error detection, self-testing and sanity-check can be performed to identify the faulty component

Architecture-level hardening | 5

Space Redundancy

Loosely-synchronized dual processor (cont.)

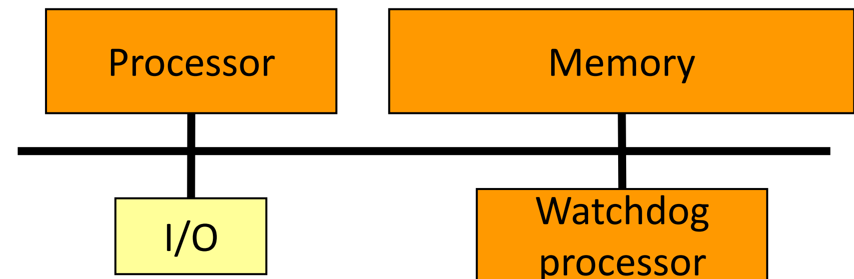
- Synchronization mechanisms must be protected with specific hardware/software mechanisms
- After an error detection, self-testing and sanity-check can be performed to identify the faulty component
- Two operational modes:
  - Critical applications: loosely-synchronized architecture featuring fault detection checks on synchronization
  - Non critical tasks: dual-core architecture

Architecture-level hardening | 6

Space Redundancy

Watchdog Processor

- The watchdog observes the behavior of the processor and performs a high-level anomaly detection
  - Execution statistics different from profiled ones (branch misses, branch prediction, ...)
  - Data values or memory addresses out of expected ranges
  - Timeout expiration

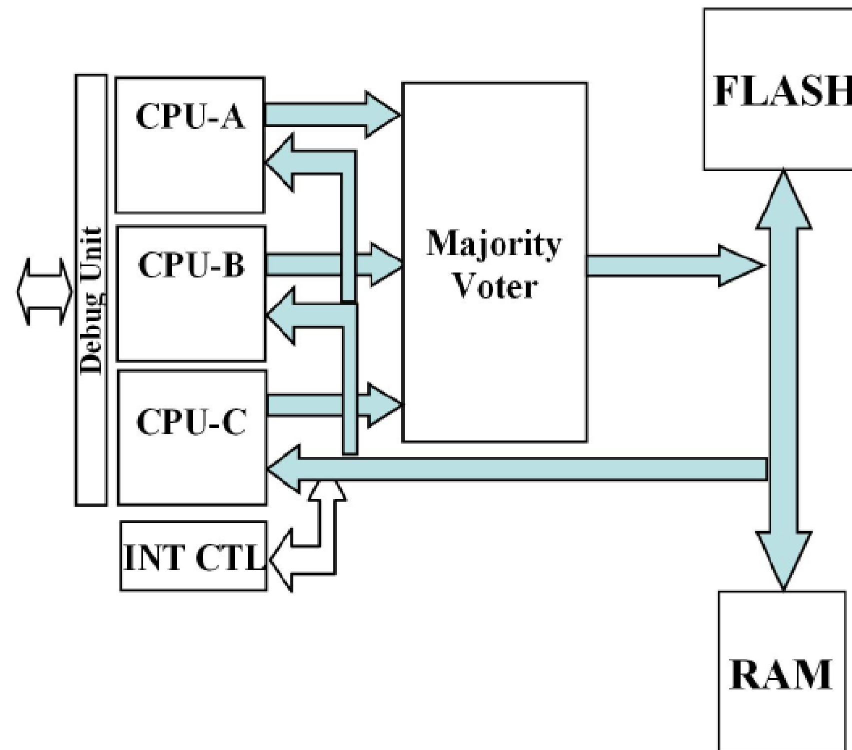


Architecture-level hardening | 7

Space Redundancy

TMR architecture

- It is a lock-step solutions with three processors



Architecture-level hardening | 8

Space Redundancy

Dual lock-step architecture

- Two dual lock-step nodes are connected
- Each node is fail-silent
- Two operational modes:
  - Fault detection for not-critical tasks (each dual lock-step executing a different code)
  - Fault tolerance for critical tasks (both dual lock-step executing the same code)
- This is a simple distributed system

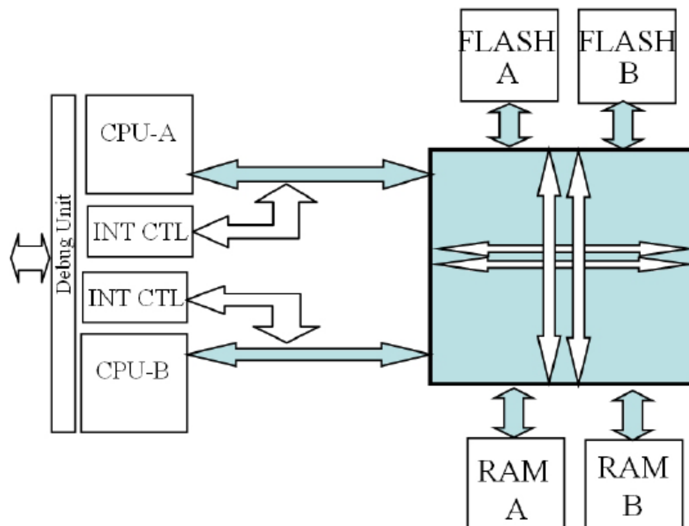
RAM
FLASH
INT CTL
CPU A Master
CPU A Checker
Debug Unit
Address & DATA monitor
Interchannel Comms
CPU B Checker
CPU B Master
Debug Unit
Address & DATA monitor
INT CTL
RAM
FLASH

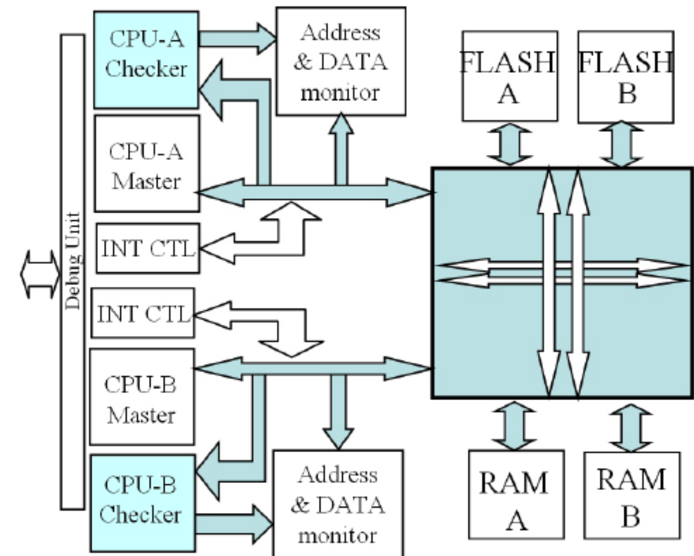
Architecture-level hardening | 9

Space Redundancy

Alternative solutions

– Memories can be shared between processor replicas, but…



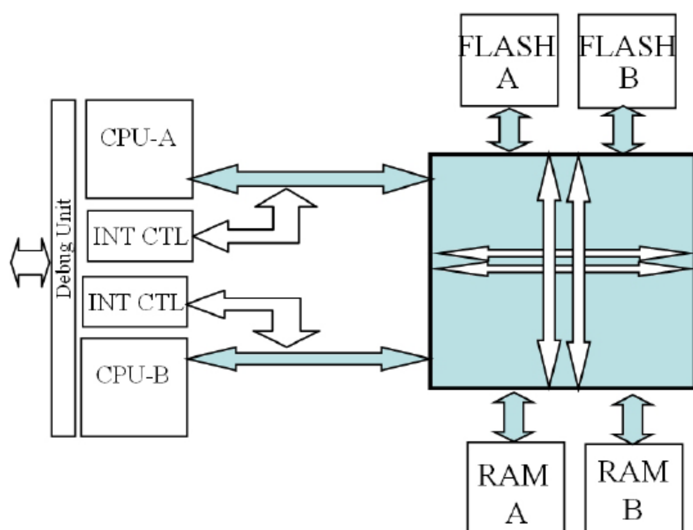


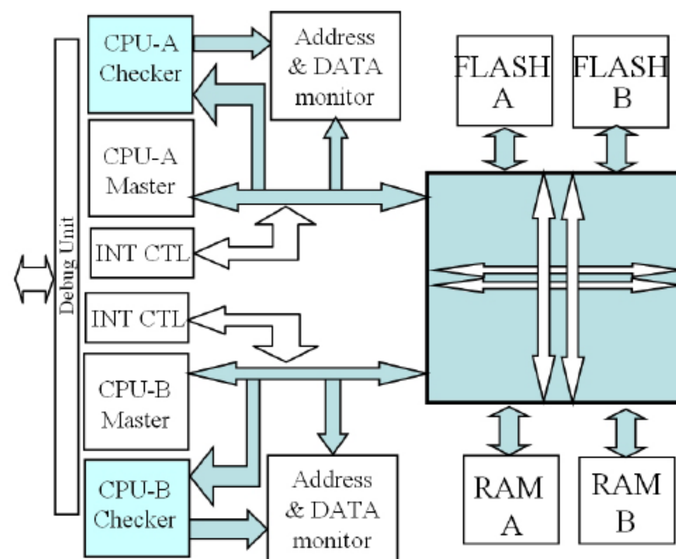
Architecture-level hardening | 9

Space Redundancy

Alternative solutions

- Memories can be shared between processor replicas, but…
- Issues with data protection in shared memory
  - A faulty task can corrupt data of the replica task





Process-level hardening

Space or Time Redundancy

Software processes can be replicated and the results compared

The operating system (or a hypervisor) manages the replicas' execution and result comparison, possibly by means of specific hardware components

Heterogeneous reliability requirements:

- The system executes both hardened processes and plain ones
- Many applications may tolerate a certain number of errors (e.g., image processing ones)

Process-level hardening | 2

The approach is widely-used due to the increasing diffusion of multi-core and many-core architectures

- Mix of both time and space redundancies

Redundancy can be applied

- At design time: replications are implemented in the source code of the program
- At run-time: the operating system decides to replicate tasks

Process-level hardening | 3

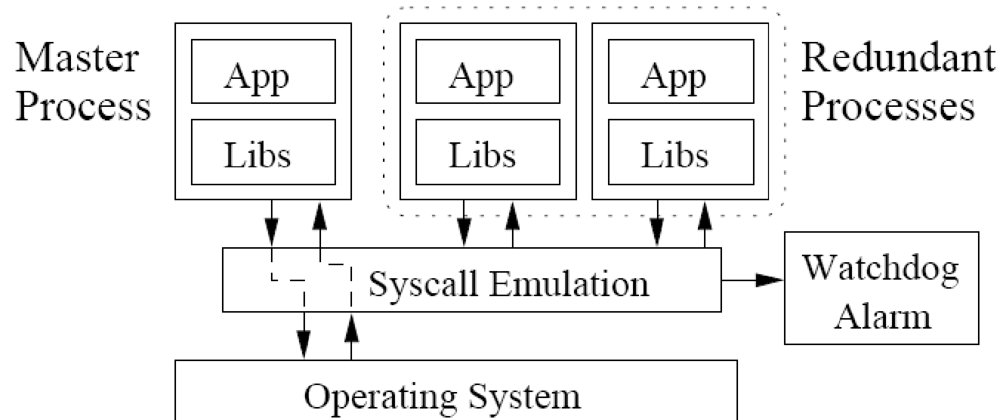
Issues:

- Isolation: a process cannot access memory spaces of other replicas
- The operating system must expose some kind of fault detection mechanisms

Process-level hardening | 4

Various alternatives:

- Run the same program or run diversified copies of the program **(SW diversity)**
  - a or b ←→ not(not(a) and not(b)) , a+b ←→ a – (– b)
- Check intermediate results
  - Possibly implemented by monitoring system calls



Mixed-level hardening

Mixed approaches can also be used:

- Fault detection is achieved at architectural-level
- Fault tolerance at process-level

The approach is mainly used on distributed systems

Approaches:

- Process replication
- Process re-execution
- Checkpointing
- Instruction-level hardening

Mixed-level hardening | 2

Process replication

- Faults are detected by the architecture that does not return any result (fail-silent) or that raises a warning
- Returned results are correct (at least we need results from one replica)

Process re-execution

- When a fault is detected, the process is re-started

Mixed-level hardening | 3

Checkpointing

- The operating system (or a HW mechanism) performs periodic checkpointing of the status
- When a fault is detected, the status of the system is restored to the previous checkpointing

Instruction-level hardening

Time and information redundancy can be applied in the software at instruction level

It is possible to act on both the source code or the assembler one

Instruction-level hardening | 2

Instructions types

- **Data**: they perform an elaboration
  - Assignment, sum, and, or, less then, equal,…
- **Control**: they allow the modification of the linear flow of the execution
  - Conditional instructions, loops (with initial condition, with final condition, with counting), calls.

It is necessary to manage both the execution and the data flows

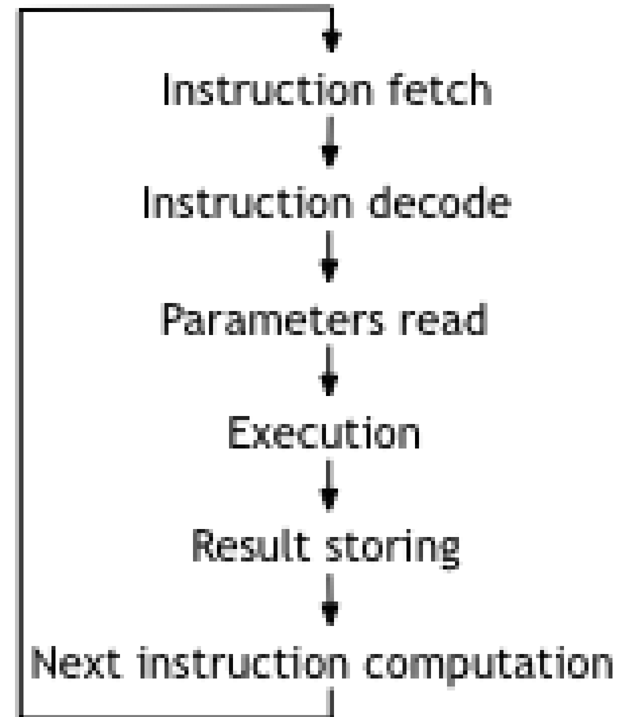
Instruction-level hardening | 3

Fault modeled as:

- corruption of the execution of a single step
- corruption of a stored value

Effects:

- erroneous execution of workflow
- data errors



Instruction-level hardening | 4

Errors while executing **data** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ A data processing instruction is transformed into: □ another data processing instruction: a wrong result is produced □ a control instruction: an erroneous jump to a random target is performed ▷ in the same basic block, or ▷ in a different basic block
○ Parameters read ○ Execution ○ Result storing	○ Instruction execution leads to a wrong result that is stored in memory
○ Next instruction computation	○ An erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block

Instruction-level hardening | 4

Errors while executing **data** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ A data processing instruction is transformed into: □ another data processing instruction: a wrong result is produced □ a control instruction: an erroneous jump to a random target is performed ▹ in the same basic block, or ▹ in a different basic block
○ Parameters read ○ Execution ○ Result storing	○ Instruction execution leads to a wrong result that is stored in memory
○ Next instruction computation	○ An erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block

Instruction-level hardening | 4

Errors while executing **data** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ A data processing instruction is transformed into: □ another data processing instruction: a wrong result is produced □ a control instruction: an erroneous jump to a random target is performed ▹ in the same basic block, or ▹ in a different basic block
○ Parameters read ○ Execution ○ Result storing	○ Instruction execution leads to a wrong result that is stored in memory
○ Next instruction computation	○ An erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block

Instruction-level hardening | 5

Errors while executing **control** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ The control instruction is transformed into: □ a data processing instruction: a wrong result is produced and no jump is performed □ another control instruction: an erroneous jump to a random target is performed ▹ in the same basic block, or ▹ in a different basic block
○ Parameters read ○ Execution ○ Next instruction computation	○ Jump target computation error: an erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block ○ Branch direction evaluation error: wrong branch direction is taken ○ The execution of a jump instruction is affected: no jump is performed
○ Result storing	○ No instruction is performed: the fault causes no error because it is not activated

Instruction-level hardening | 5

Errors while executing **control** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ The control instruction is transformed into: □ a data processing instruction: a wrong result is produced and no jump is performed □ another control instruction: an erroneous jump to a random target is performed ▷ in the same basic block, or ▷ in a different basic block
○ Parameters read ○ Execution ○ Next instruction computation	○ Jump target computation error: an erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block ○ Branch direction evaluation error: wrong branch direction is taken ○ The execution of a jump instruction is affected: no jump is performed
○ Result storing	○ No instruction is performed: the fault causes no error because it is not activated

Instruction-level hardening | 5

Errors while executing **control** instructions

Corrupted phase	Generated errors
○ Instruction fetch ○ Instruction decode	○ The control instruction is transformed into: □ a data processing instruction: a wrong result is produced and no jump is performed □ another control instruction: an erroneous jump to a random target is performed ▹ in the same basic block, or ▹ in a different basic block
○ Parameters read ○ Execution ○ Next instruction computation	○ Jump target computation error: an erroneous jump to a random instruction occurs: □ in the same basic block, or □ in a different basic block ○ Branch direction evaluation error: wrong branch direction is taken ○ The execution of a jump instruction is affected: no jump is performed
○ Result storing	○ No instruction is performed: the fault causes no error because it is not activated

Instruction level hardening | 3

Data instructions

- Instructions are duplicated and results are compared duplication and comparison
- The technique implies that all variables must be duplicated
- Example:

Original code	*Modified Code*

`int a,b;`
 	
`a = b;`
	`int` a_0, b_0, a_1, b_1`;`
	
	a_0 `=` b_0`;`
	a_1 `=` b_1`;`
	`if (`b_0 `!=` b_1`)`
	`error();`

`a = b + c;`	a_0 `=` b_0 `+` c_0`;`
	a_1 `=` b_1 `+` c_1`;`
	`if((`b_0`!=`b_1`)||(`c_0`!=`c_1`))`
	`error();`

Instruction level hardening | 4

Data instructions (cont.)

- **Selective instruction duplication**
  - Identification of the trade-off between the reliability level and performance degradation
    1. Code Reliability Analysis
    2. Code reordering
    3. Selective Variable duplication

Instruction level hardening | 5

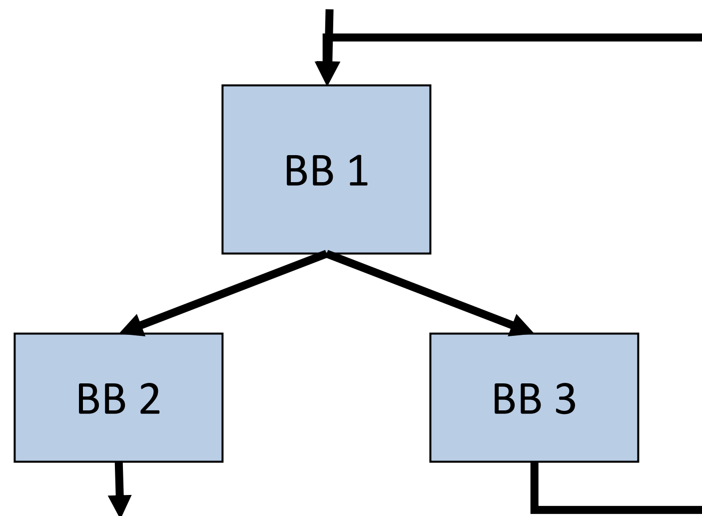
Selective instruction duplication (cont.)

- Code Reliability Analysis
  - Each variable is associated with a reliability-weight:
    - The longer the lifetime the higher the probability of being corrupted
    - The more is the number of descendents the higher is the number of variables on which an erroneous value could be propagated
- Code reordering
  - Functional equivalent code but with and improved global reliability weight
- Selective variable duplication

Instruction level hardening | 6

Control instructions

- Main concept: Basic Block (BB)
  - basic block is a sequence of program statements that contains no labels and no branches
    from this definition a BB can only be executed completely and in sequence



Instruction level hardening | 7

Control instructions (cont.)

- A program can be described by a sequence of BBs
- Control Flow Errors are:
  - illegal branches
  - wrong branches
  - branches in the middle of a BB
  - branches from the middle of a BB

Instruction level hardening | 8

Control instructions (cont.)

- Two **signatures** are introduced to control if
  - the BB has been successfully completed
  - The control flow jumps from the end of a BB to the beginning of a correct BB

- The two signatures are:
  - a first instruction is introduced at the beginning of each BB
  - a second one is inserted at the end of the BB

Instruction level hardening | 9

Control instructions (cont.)

– Logic schema: intra-block signature
  - Identify erroneous jumps

Instruction level hardening | 10

Procedure duplication

– The body of the procedure is hardened

Original code	*Modified code*
`int res,a;`	`int res0, res1, a0, a1;`
`res = search (a);`	`search(a0, a1, &res0, &res1);`
`…`	`…`
`int search (int p)`	`void search(int p0,int p1,int *r0,int *r1)`
`{ int q;`	`{ int q0, q1;`
`…`	`…`
`q = p + 1;`	`q0 = p0 + 1;`
	`q1 = p1 + 1;`
	`if (p0 != p1) error();`
`…`	`…`
`return(1);`	`*r0 = 1;`
	`*r1 = 1;`
	`return;`
`}`	`}`

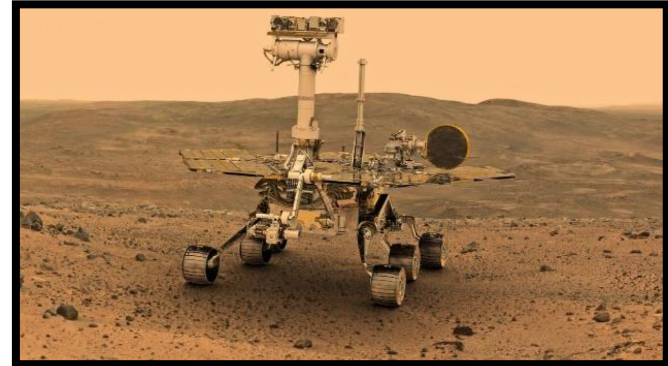
Procedure call duplication

– The procedure is called twice with the original parameter and the replicated ones

Application-level fault detection/management

Working scenario

Image Processing and Machine Learning for perception and decision tasks in mission/-safety-critical systems



Application-level fault detection/management

GOAL: reduce reliability-related costs

STRATEGY: avoid re-computation when the corrupted output can still be used, exploiting context inherent tolerance to some degree of inexactness

APPROACH: evaluate the produced output (final or intermediate) w.r.t. a “usability” concept and re-process it only if strictly necessary

correct/corrupted

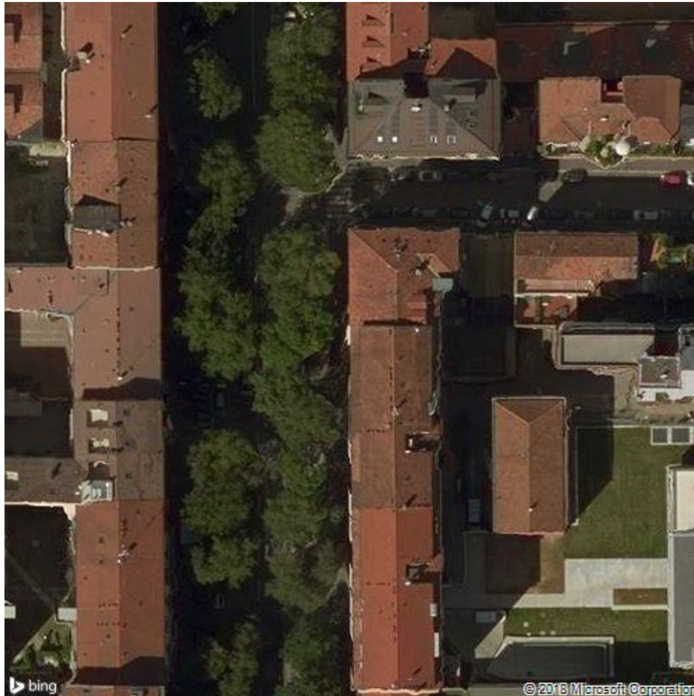
vs.

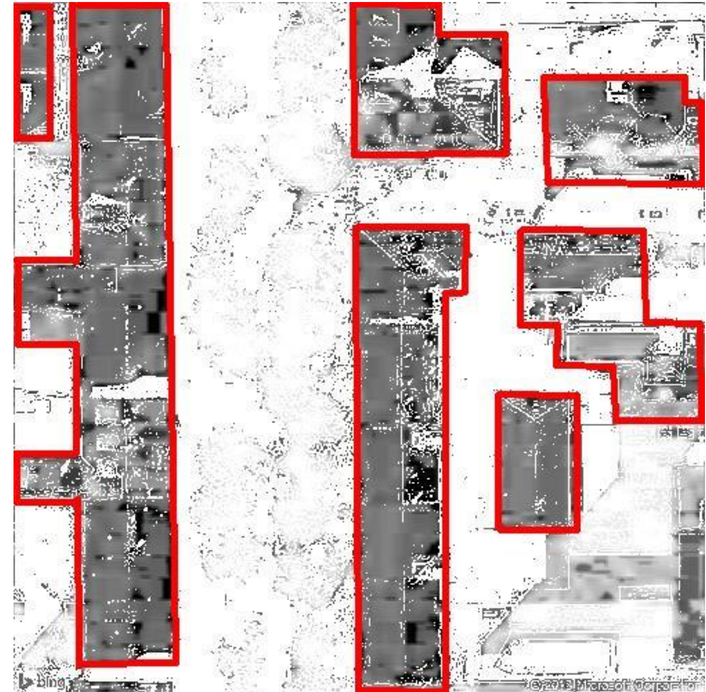
usable/unusable

Application-level fault detection/management

Example application

Building identification in aerial pictures





Application-level fault detection/management

corrupted/usable

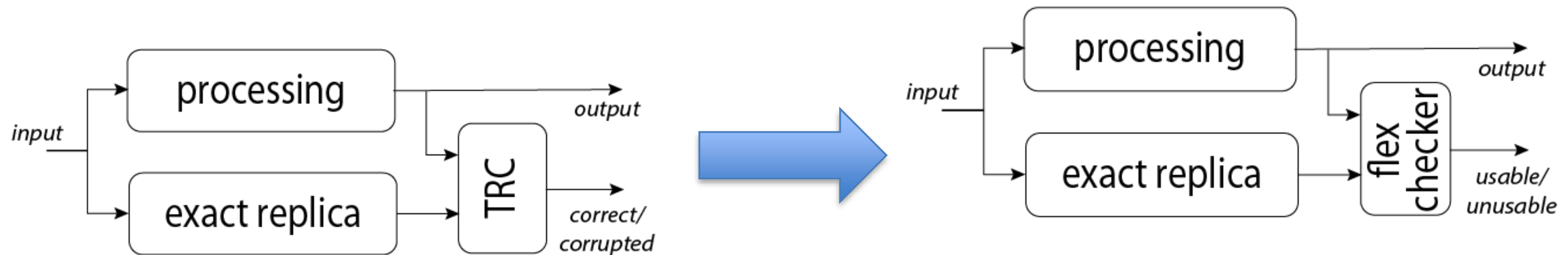
A

B

corrupted/unusable

Application-level fault detection/management

Strategy: innovative lightweight fault impact management



- Traditional redundancy
- CNN-based usability-oriented flexible checking

TOPIC QUESTIONS

How to harden computing systems?

What hardware/software techniques are available?

How can they be applied? Can they be applied?

What coverage do they offer?

How difficult it is to apply them?

TOPICS

The hardening of a system can be performed at different abstraction levels

Applicability

Trade-off w.r.t. various parameters