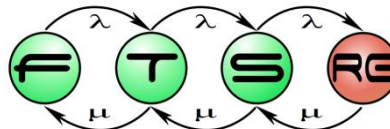


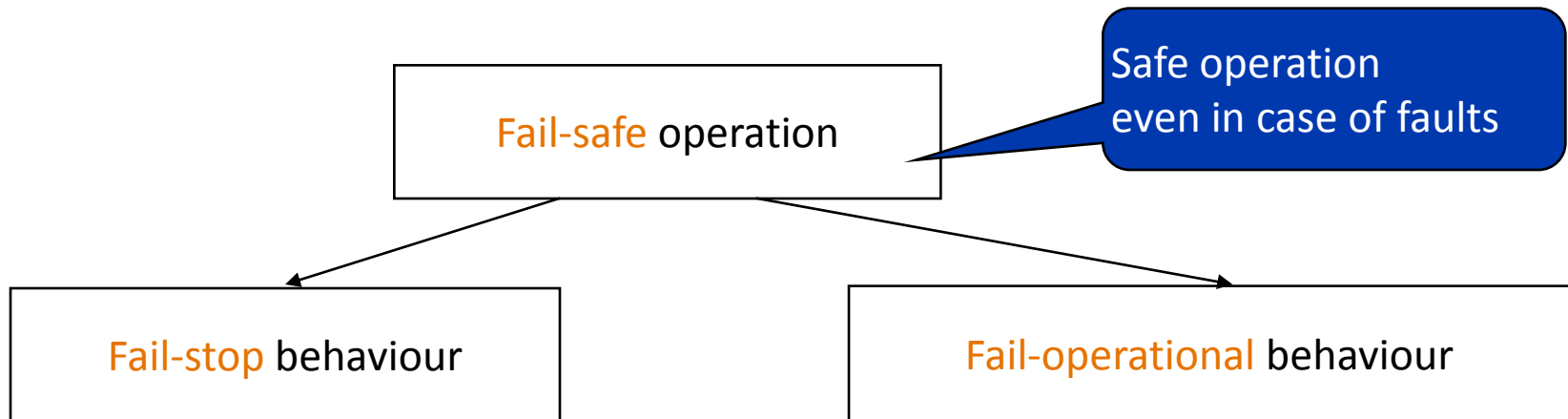
Design of the architecture of safety-critical systems

Ákos Horváth

Based on István Majzik's slides
Dept. of Measurement and Information Systems



Objectives

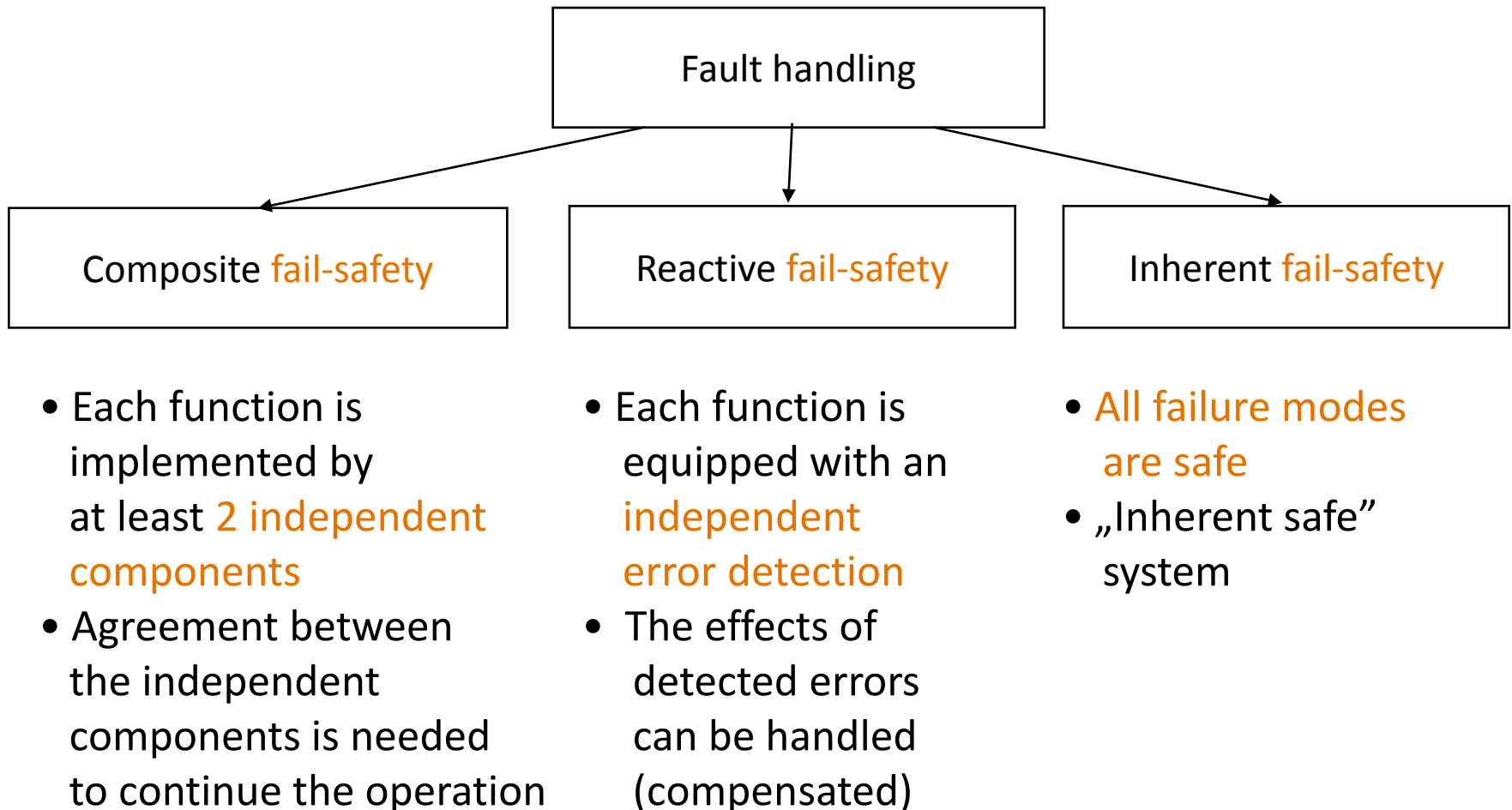


- Stopping (switch-off) **is a safe state**
- In case of a detected error the system has to be stopped
- **Detecting errors** is a critical task

- Stopping (switch-off) **is not a safe state**
- Service is needed even in case of a detected error
 - full service
 - degraded (but safe) service
- **Fault tolerance** is required

Architectural solutions (overview)

■ Safety in case of single random hardware faults

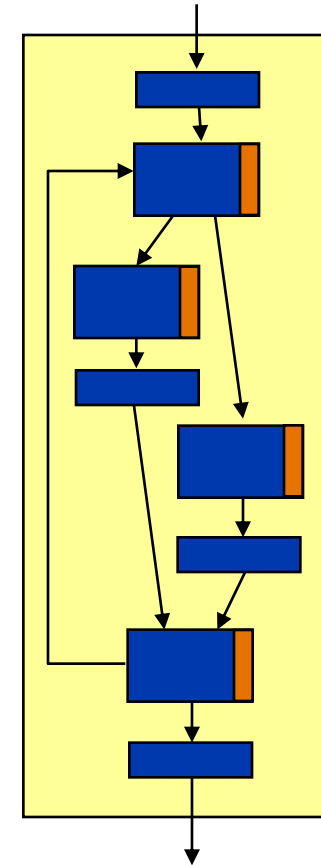


Typical architectures for fail-stop operation



1. Single channel architecture with built-in self-test

- Single processing flow
- Scheduled **hardware self-tests**
 - After switch-on: Detailed self-test to detect permanent faults
 - In run-time: On-line tests to detect latent permanent faults
- Scheduled **software self-tests**
 - Typically application dependent techniques
 - Checking the control flow, data acceptance rules, timeliness properties
- Disadvantages:
 - Fault coverage of the self-tests is limited
 - Fault handling (e.g., switch-off) shall be performed by the same channel

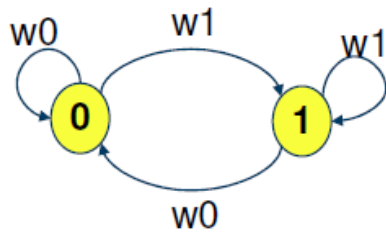


Implementation of on-line error detection

- **Application dependent** (ad-hoc) techniques
 - Acceptance checking (e.g., for ranges of values)
 - Timing related checking (e.g., too early, too late)
 - Cross-checking (e.g., using inverse function)
 - Structure checking (e.g., in linked list structure)
- **Application independent** mechanisms
 - Hardware supported on-line checking
 - CPU level: Invalid instruction, user/supervisor modes etc.
 - MMU level: Protection of memory ranges
 - Generic architectural solutions
 - Two-channel execution with comparison
 - Two-channel execution with safety bag

Example: Testing memory cells

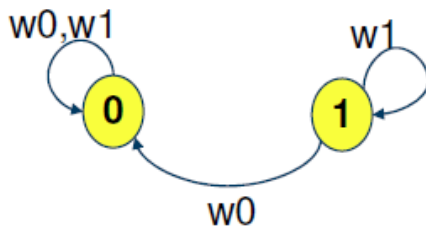
States of a correct cell:



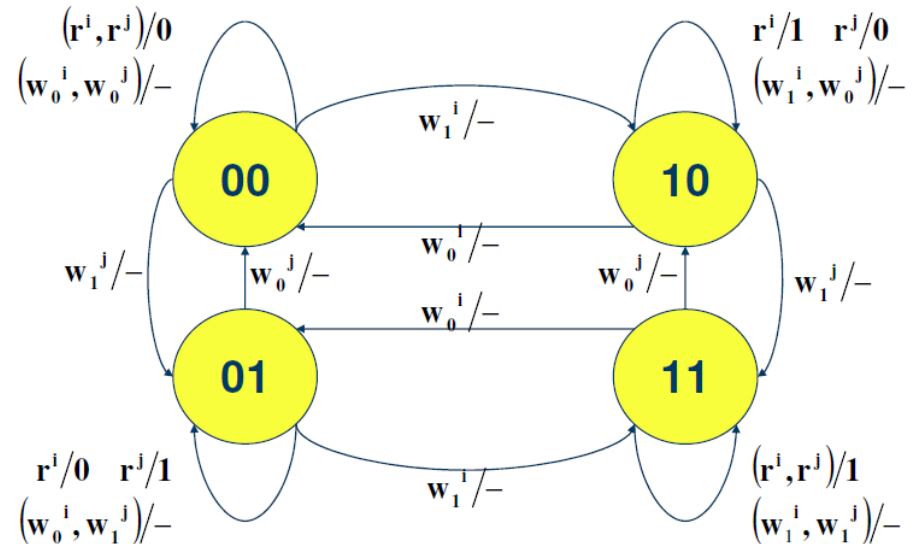
Stuck-at 0/1 faults:



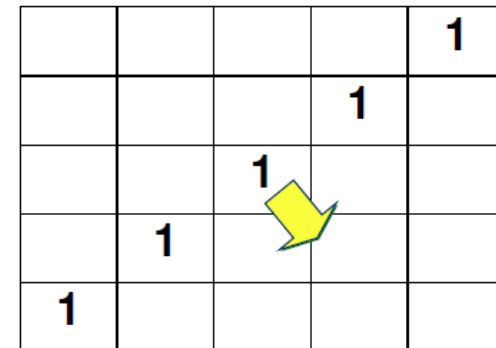
Transition fault:



State transitions to check stuck faults:



„March” algorithms:



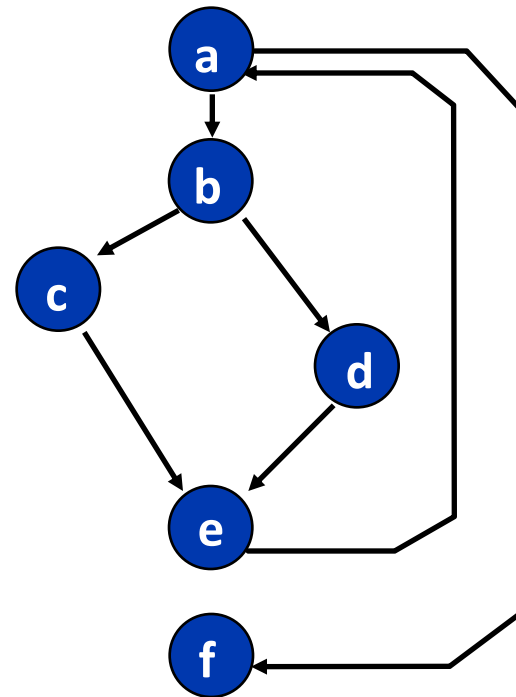
Example: Software self-test

- Checking the correctness of execution paths
 - On the basis of the program control flow graph

Source code:

```
a: for (i=0; i<MAX; i++) {  
b:   if (i==a) {  
c:     n=n-i;  
      } else {  
d:     m=m-i;  
      }  
e:   printf(“%d\n”,n);  
      }  
f:   printf(“Ready.”)
```

Control flow graph:



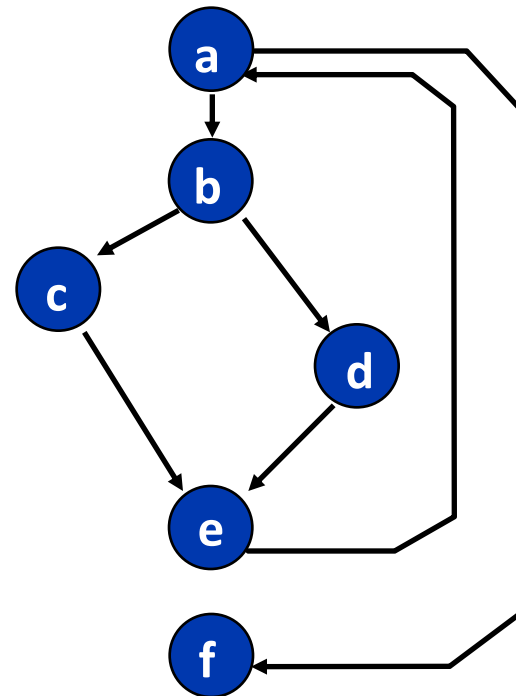
Example: Software self-test

- Checking the correctness of execution paths
 - On the basis of the program control flow graph
 - Actual run: Checked on the basis of assigned signatures

Instrumented source code:

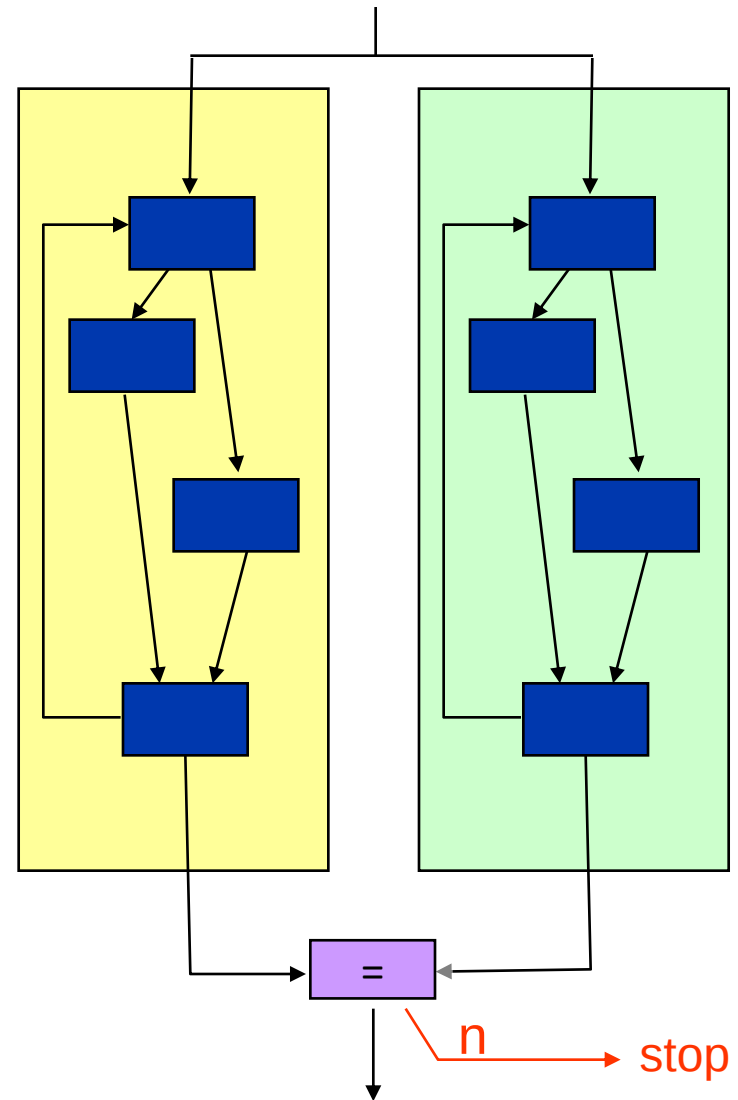
```
a: S(a); for (i=0; i<MAX; i++) {  
b:   S(b); if (i==a) {  
c:     S(c); n=n-i;  
      } else {  
d:     S(d); m=m-i;  
      }  
e:   S(e); printf(“%d\n”,n);  
      }  
f:   S(f); printf(“Ready.”)
```

Control flow graph (reference):

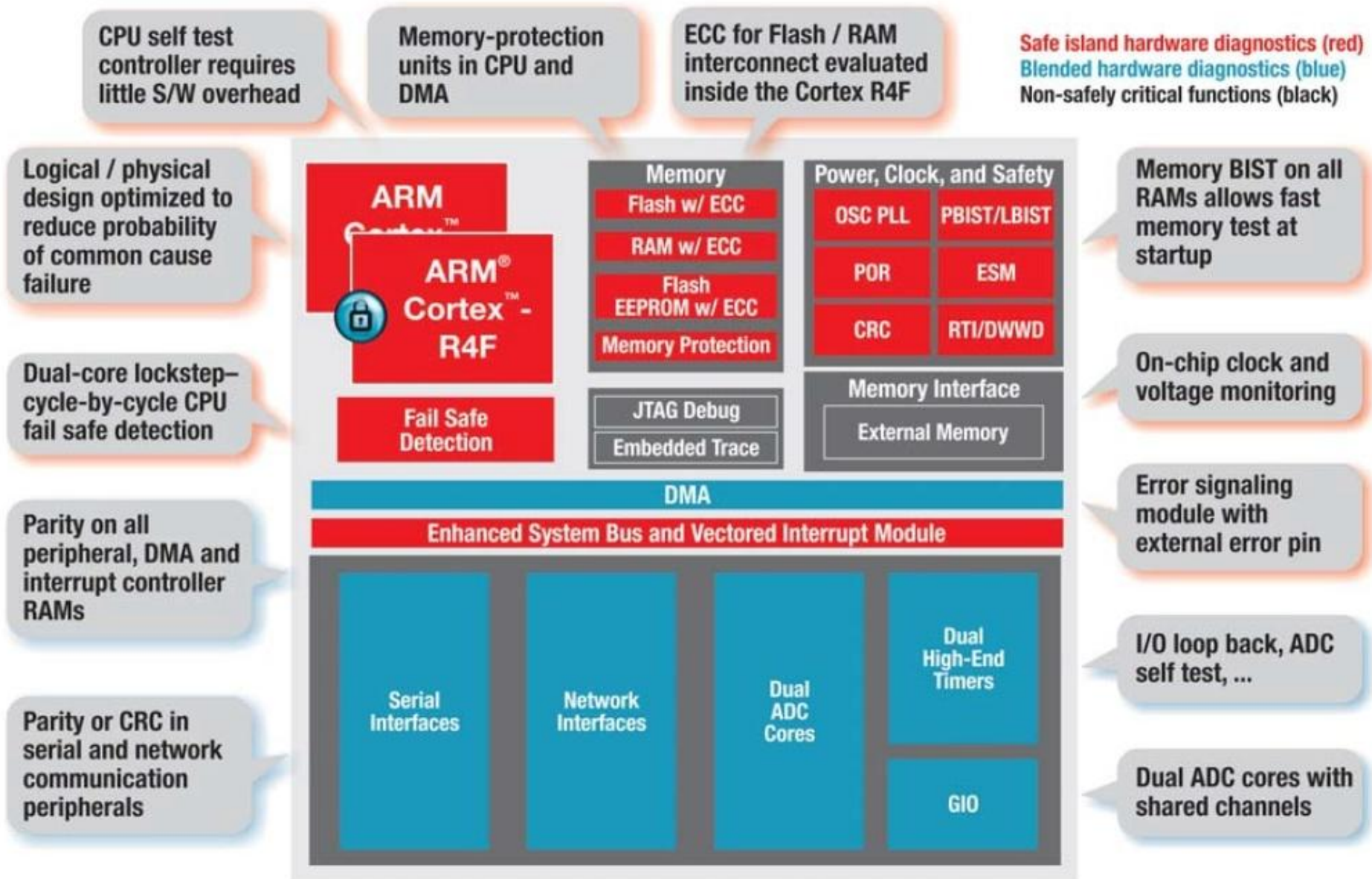


2. Two-channels architecture with comparison

- Two or more processing channels
 - Shared input
 - Comparison of outputs
 - Stopping in case of deviation
- High error detection coverage
- The comparator is a critical component (but simple)
- Special way of comparison:
 - Performed by the operator
- Disadvantages:
 - Common mode faults
 - Long detection latency

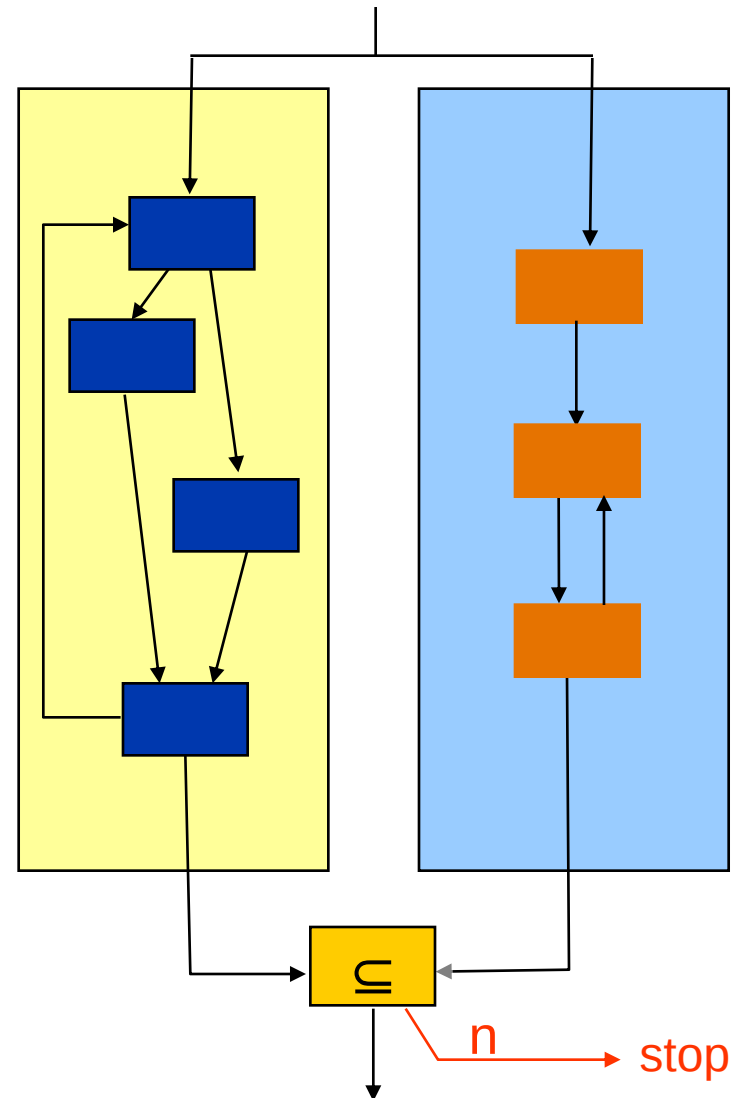


Example: TI Hercules Safety Microcontrollers

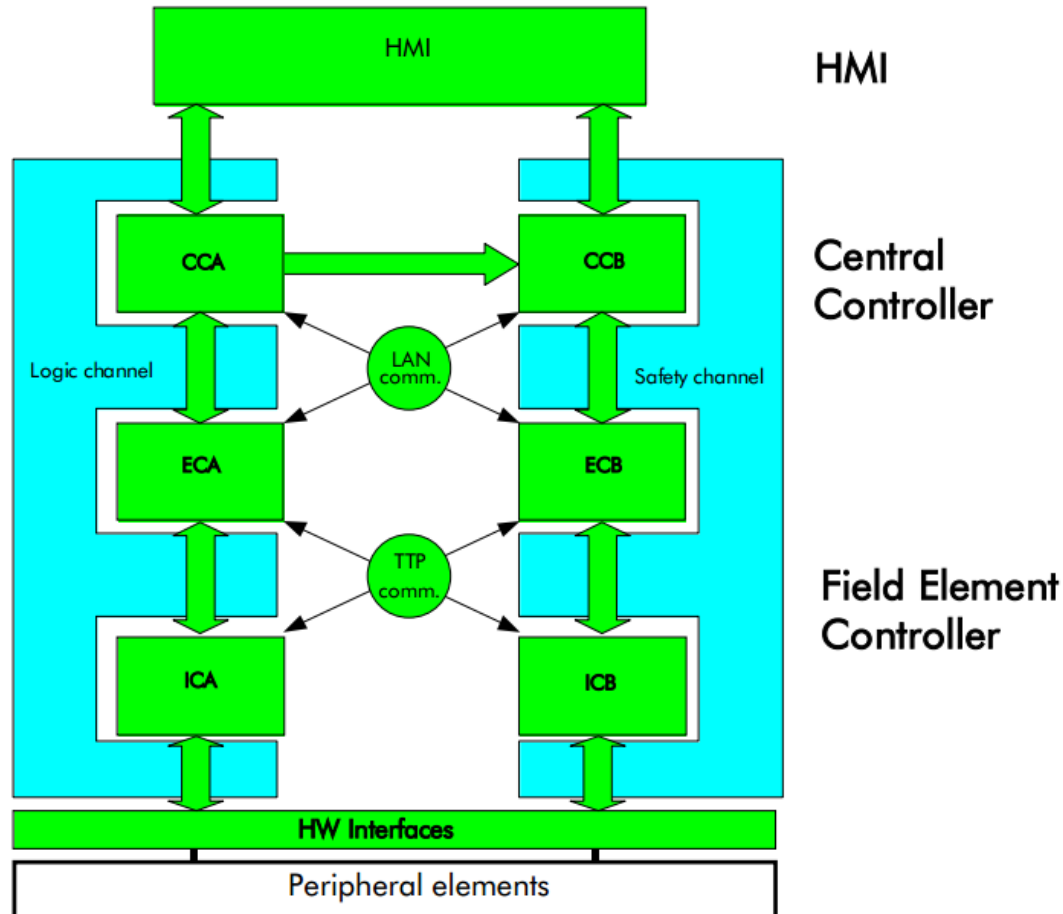


3. Two-channels architecture with safety bag

- Independent second channel
 - „Safety bag”: only safety checking
 - Diverse implementation
 - Checking the output of the primary channel
- Example:
 - Elektra railway interlocking system
 - Rules are implemented to check the primary channel



Example: Alcatel (Thales) Elektra



Two channels:

- **Logic channel:**
CHILL (CCITT High Level Language) procedure-oriented programming language
- **Safety channel:**
PAMELA (Pattern Matching Expert System Language) rule-based language

Typical architectures for fault-tolerant systems



Objectives for fault tolerant behaviour

Fail-safe operation

Fail-stop behaviour

- Stopping (switch-off) is a safe state
- In case of a detected error the system has to be stopped
- Detecting errors is a critical task

Fail-operational behaviour

- Stopping (switch-off) is not a safe state
- Service is needed even in case of a detected error
 - full service
 - degraded (but safe) service
- Fault tolerance is required

Fault tolerant systems

- **Fault tolerance:** Providing (safe) service in case of faults
 - Autonomous error handling during operation (instead of stopping)
 - Intervening into the **fault** → **failure** chain
- **Basic condition: Redundancy**
Extra resources to replace (the service of) faulty components
 - Hardware
 - Software
 - Information
 - Time

} redundancy (sometimes joint appearance)
- **Types of redundancy**
 - **Cold:** The redundant component is inactive in fault-free case
 - **Warm:** The redundant component has reduced load in fault-free case
 - **Hot:** The redundant component is active in fault-free case

Forms of redundancy

1. Hardware redundancy

- Extra hardware components
 - Inherent in distributed systems
 - Planned for fault tolerance

2. Software redundancy

- Extra software modules

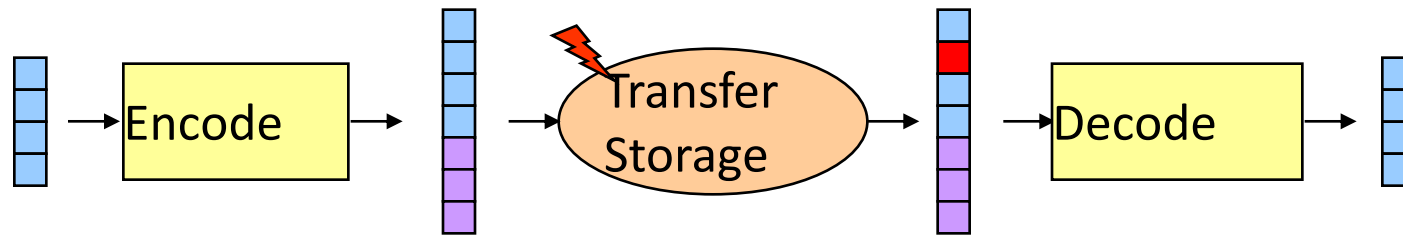
3. Information redundancy

- Extra information
 - Example: Error correcting codes (ECC)

4. Time redundancy

- Repeated execution (to handle transient faults)

Example: Error detecting and correcting codes



- **Error detecting codes (EDC):** Only detection of errors
 - Parity bit: Increasing the Hamming-distance, 1 bit error can be detected
 - Checksum: Using in case of files, messages
- **Error correcting codes (ECC):** Identifying and correcting errors
 - Higher Hamming distance: Errors can be corrected
 - E.g.: (7,4) bit Hamming code: 1 bit error corrected, 1 or 2 bit errors detected
 - Information blocks: More difficult codes are used
 - E.g.: (255, 223) byte Reed-Solomon code: 16 byte errors can be corrected
- **Limited error correction capability**
 - Information storage: In long time, more errors can occur than the number of errors that can be corrected by the applied codes
 - Basic idea: Periodic reading, correcting and writing back the information

4 data bits,
3 redundant
bits

How to use the redundancy?

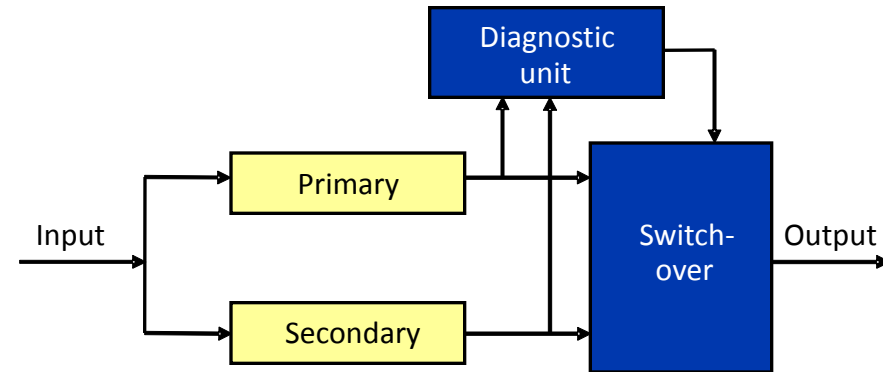
- **Hardware design faults:** (< 1%)
 - Hardware redundancy, with design diversity
 - Often are neglected (wide-spread components are used)
- **Hardware permanent operational faults:** (~ 20%)
 - Hardware redundancy (e.g., redundant processor)
- **Hardware transient operational faults:** (~ 70-80%)
 - Time redundancy (e.g., instruction retry)
 - Information redundancy (e.g., error correcting codes)
 - Software redundancy (e.g., checkpointing and recovery)
- **Software design faults:** (~ 10%)
 - Software redundancy, with design diversity

1. Fault tolerance for hardware permanent faults

Replication:

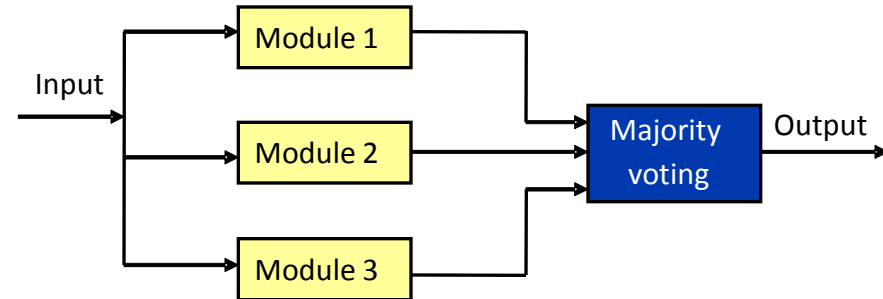
■ Duplication:

- With comparison:
Error detection only!
- With diagnostic support:
Fault tolerance by switch-over



■ TMR: Triple Modular Redundancy

- Masking the failure by majority voting
- Voter is a critical component (but simple)



■ NMR: N-modular redundancy

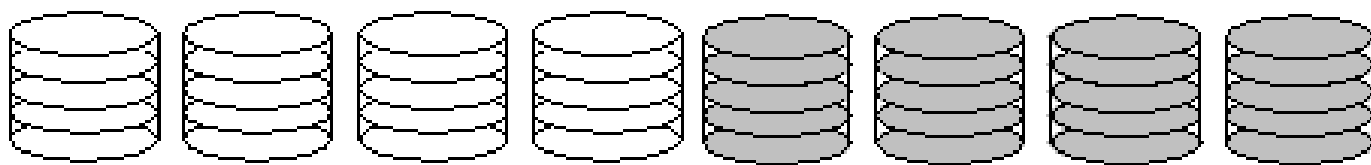
- Masking the failure by majority voting
- Goal: Surviving a mission time with high probability
- Airborne and space systems: 4MR, 5MR

Implementation of the replication

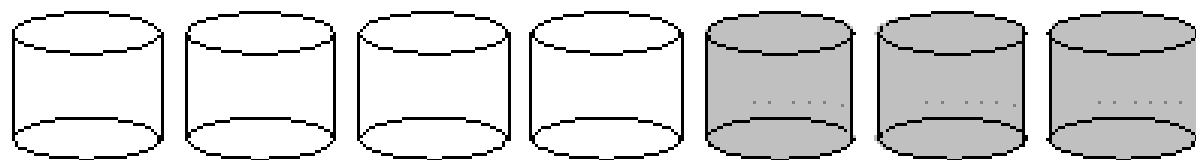
- **Equipment/server level:**
 - Servers: High availability server clusters
 - E.g., Linux HA Clustering, Windows Server Failover Clustering
 - Software support: Failover and failback
- **Board level:**
 - Run-time reconfiguration: “Hot-swap”
 - E.g., CompactPCI, HDD, power supply
 - Software support: monitoring, reconfiguration
- **Component level:**
 - Replication of components: TMR
 - Self-checking circuits (processing encoded information)

Example: RAID disk configura- tions

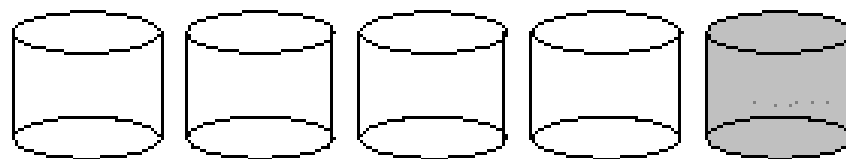
(Redundant
Array of
Independent
Disks)



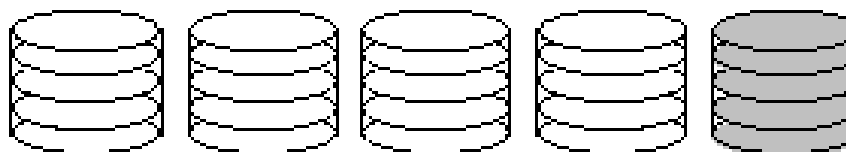
RAID-1: Mirroring (duplicated disks)



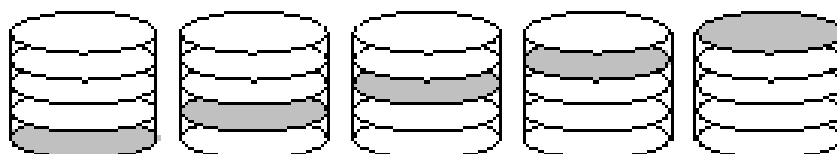
RAID-2: Bit-level ECC (error correcting codes)



RAID-3: Bit-level parity (assumption: faulty disk is identified)



RAID-4: Block-level parity (to improve performance)



RAID-5: Block-level parity (to avoid bottleneck of the parity disk)

2. Fault tolerance for transient hardware faults

- Basic approach: Software supported fault tolerance
 - Repeated execution will avoid transient faults
 - The handling of fault effects is important
 - Transient faults are handled by setting a fault-free state and continuing the execution from that state (potentially with repeated execution)
- Four phases of operation:
 - 1) Error detection
 - 2) Damage assessment
 - 3) Recovery
 - 4) Fault treatment and continuing service

The four phases of operation 1/4

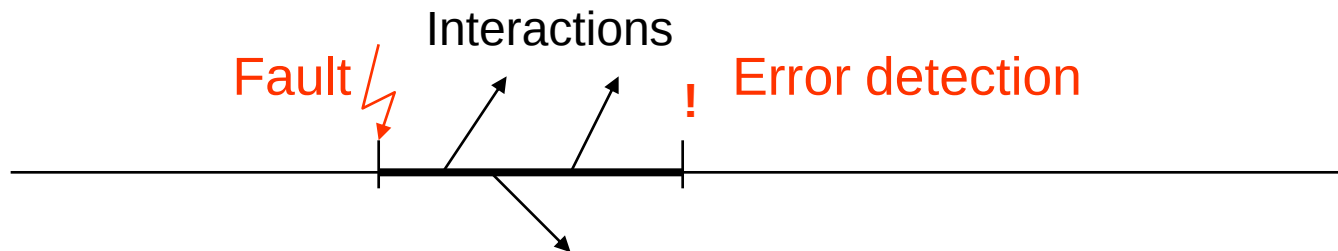
1) Error detection:

- Application independent mechanisms:
 - E.g., detecting illegal instructions at CPU level
 - E.g., detecting violation of memory access restrictions
- Application dependent techniques:
 - Acceptance checking
 - Timing related checking
 - Cross-checking
 - Structure checking
 - Diagnostic checking
 - ...

The four phases of operation 2/4

2) Damage assessment:

- Motivation: Errors can **propagate among the components** between the occurrence and detection of errors



- Limiting error propagation: **Checking interactions**
 - Input acceptance checking (to detect external errors)
 - Output credibility checking (to provide „fail-silent” operation)
 - Checking and logging resource accesses and communication
- Estimation of components affected by a detected error
 - Analysis of interactions (during the latency of error detection)

The four phases of operation 3/4

3) Recovery from an erroneous state

■ Forward recovery:

- Setting an error-free state by **selective correction**
- Dependent on the detected error and estimated damage
- Used in case of anticipated faults

■ Backward recovery:

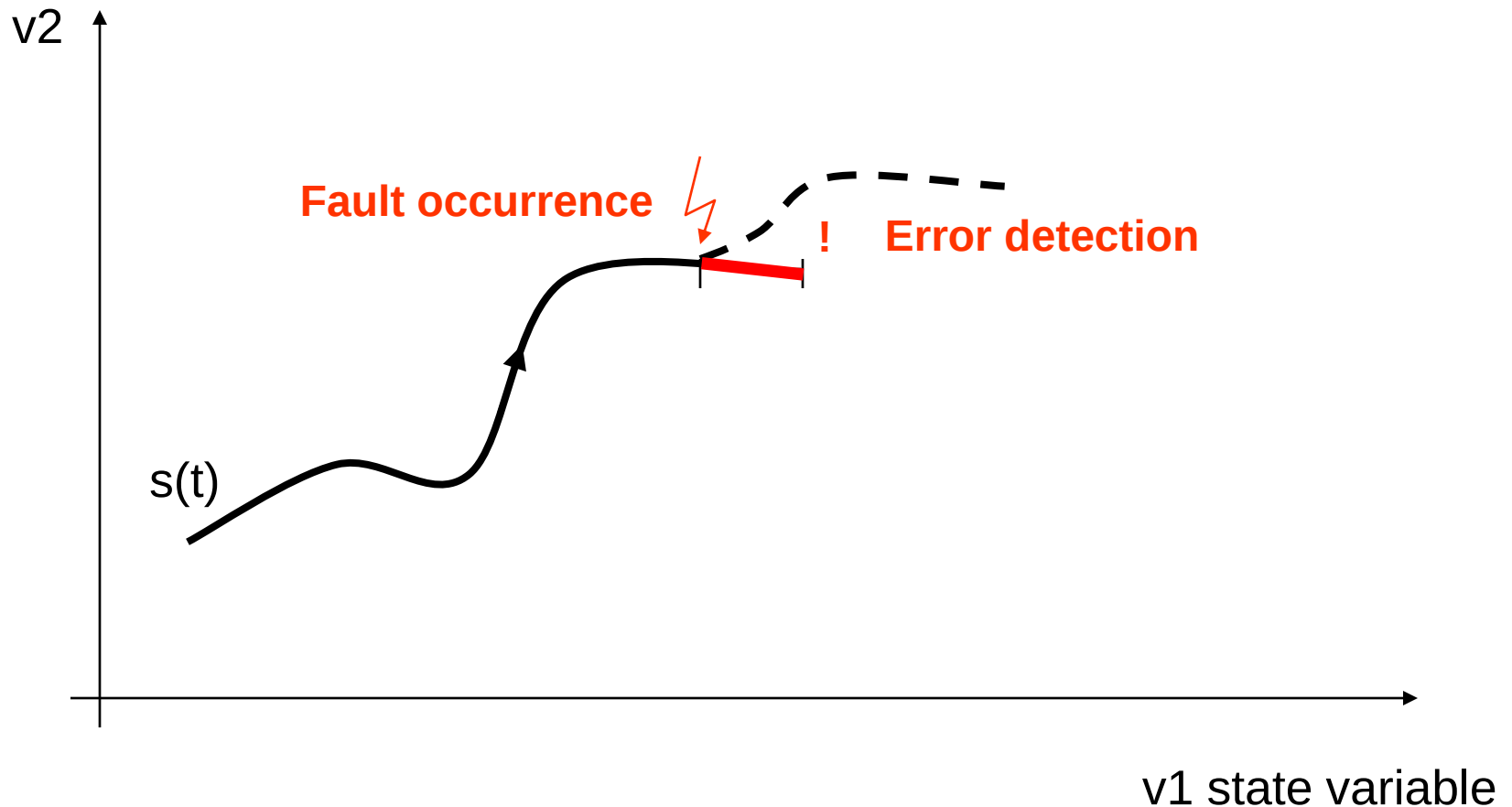
- Restoring a prior error-free state (saved earlier)
- Independent of the detected error and estimated damage
- State shall be saved and restored for each component

■ Compensation:

- The error can be handled by using redundant information

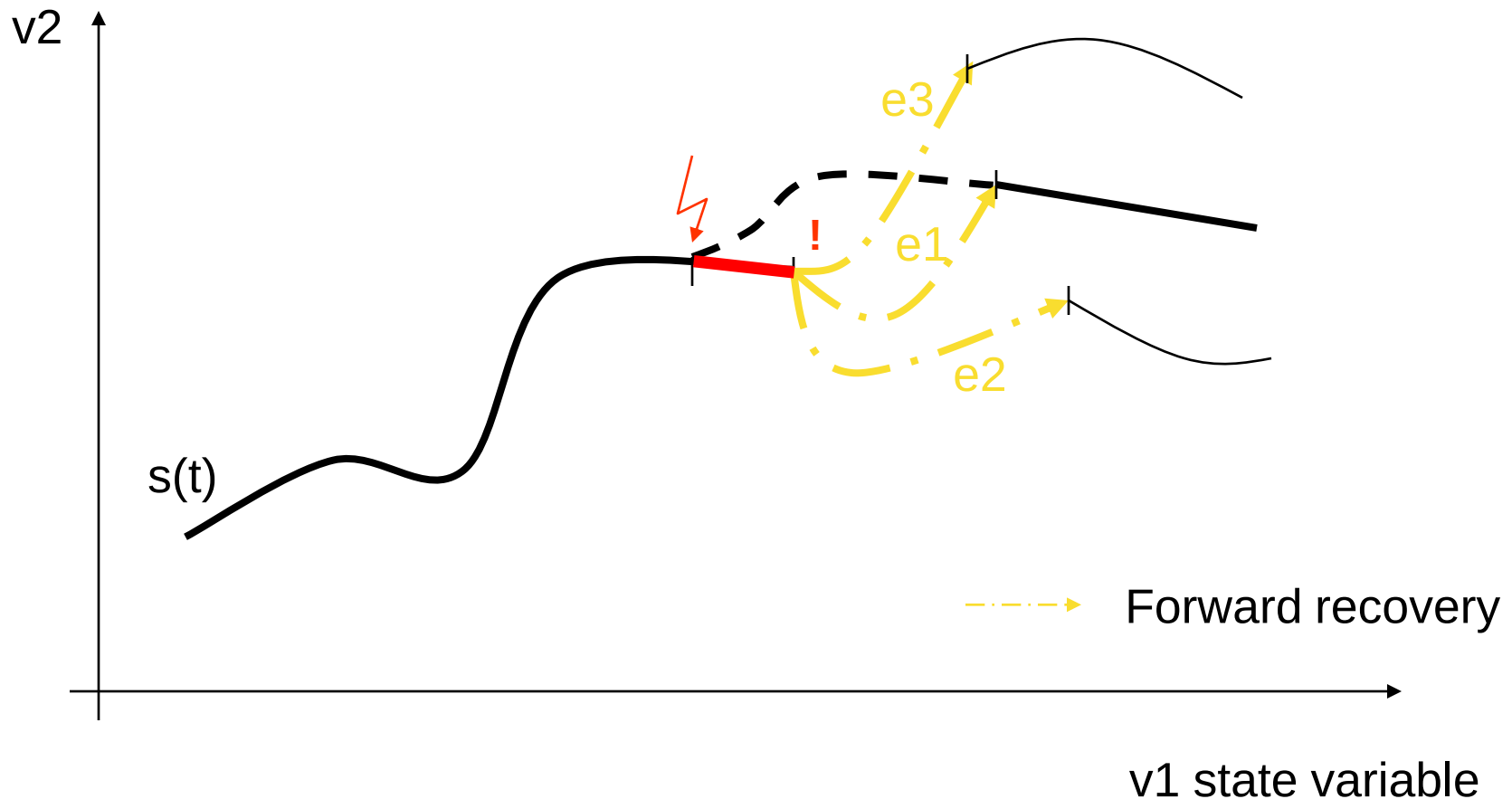
Types of recovery

- State space of the system (example): Error detection



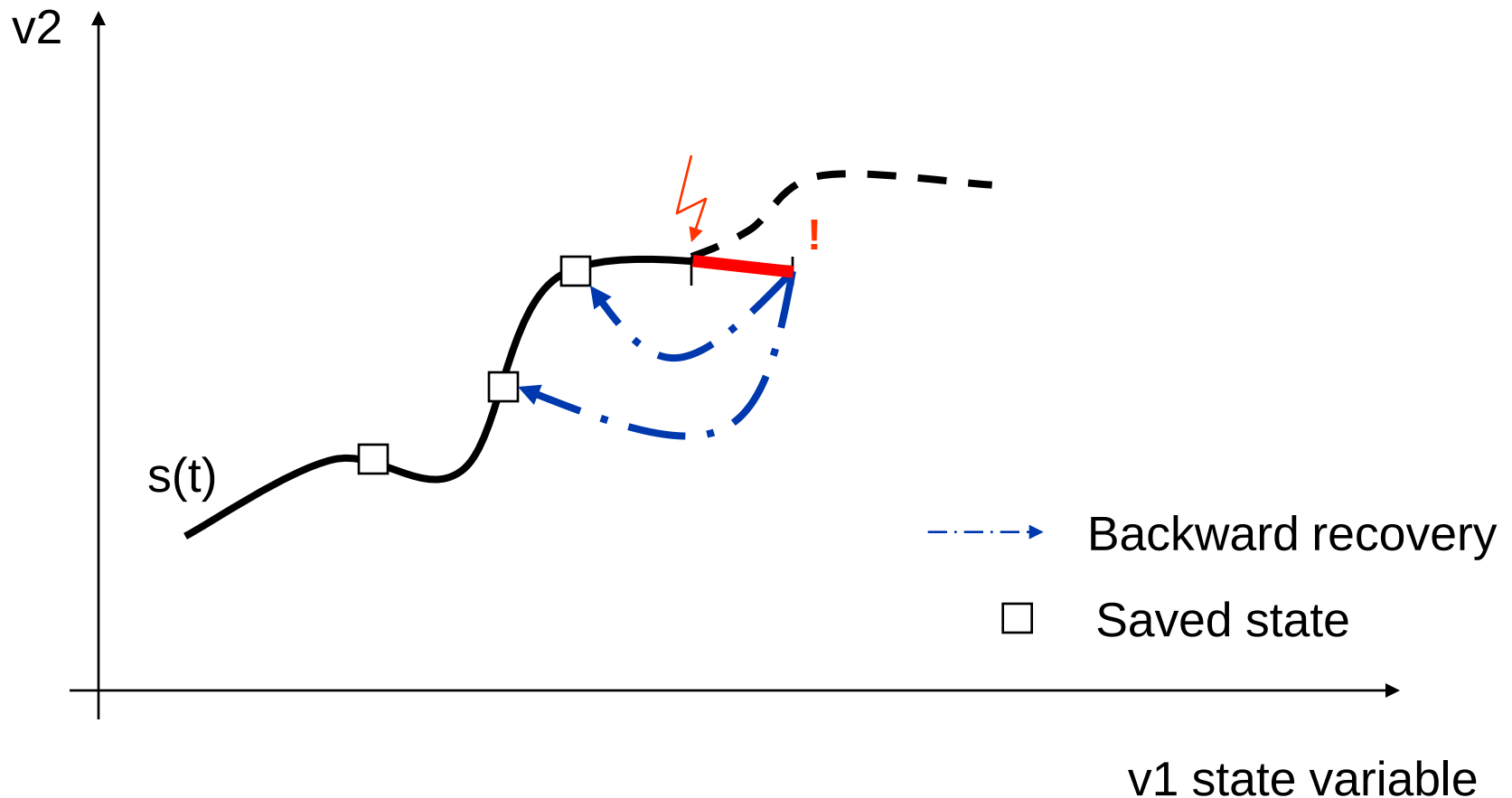
Types of recovery

- State space of the system: Forward recovery



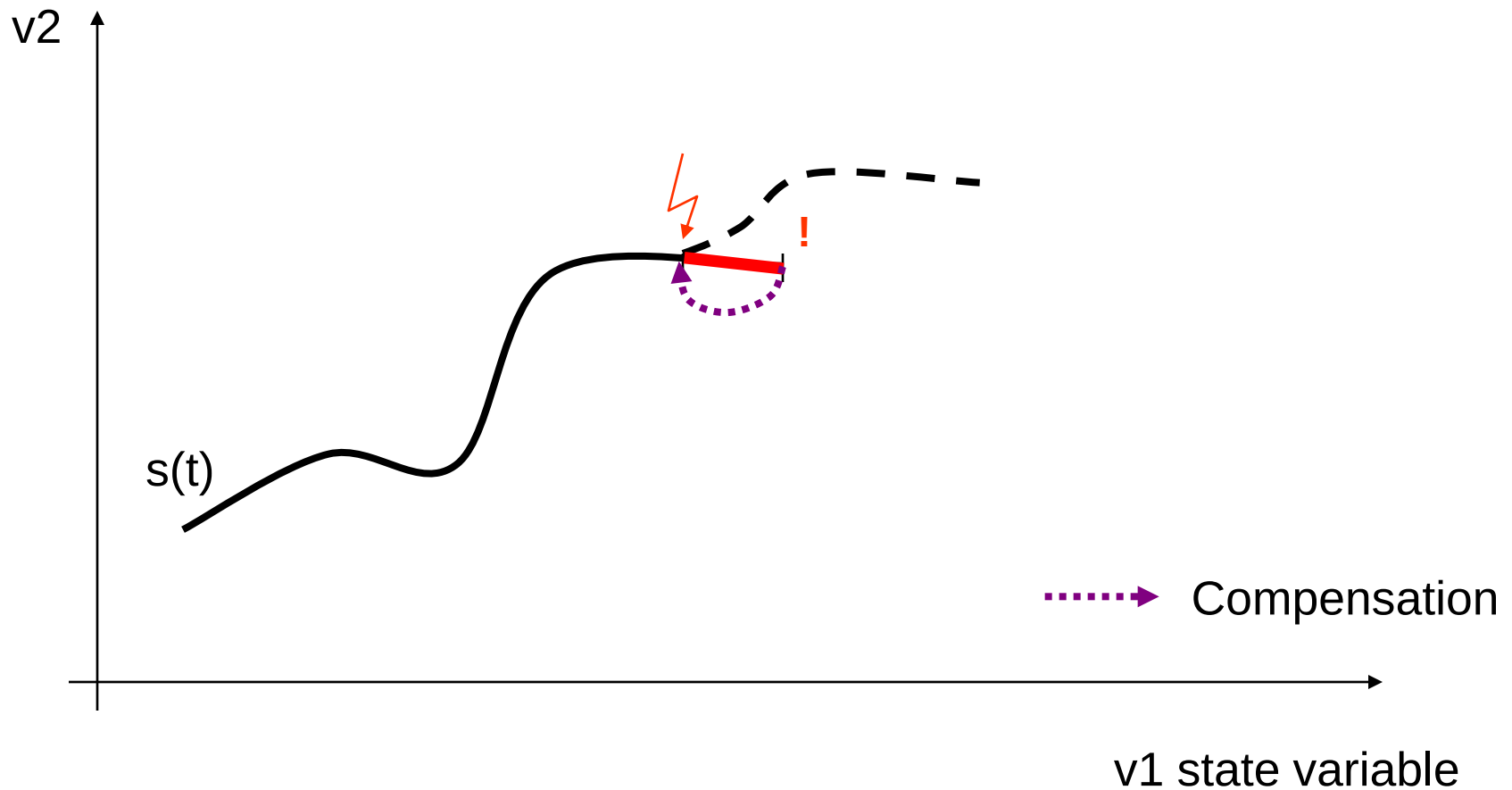
Types of recovery

- State space of the system: Backward recovery



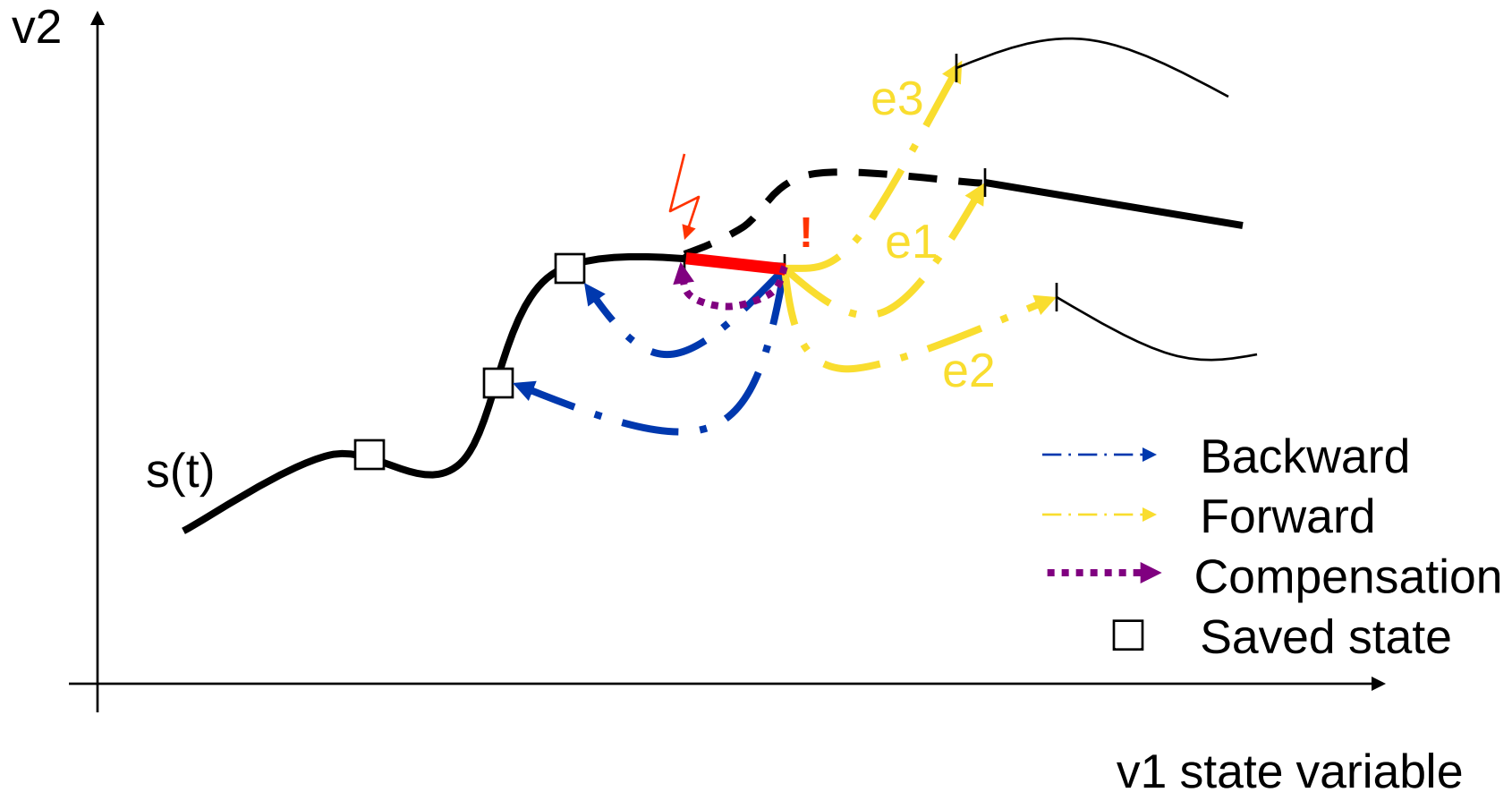
Types of recovery

- State space of the system: Compensation



Types of recovery

- State space of the system: Types of recovery



Backward recovery

■ Based on saved state

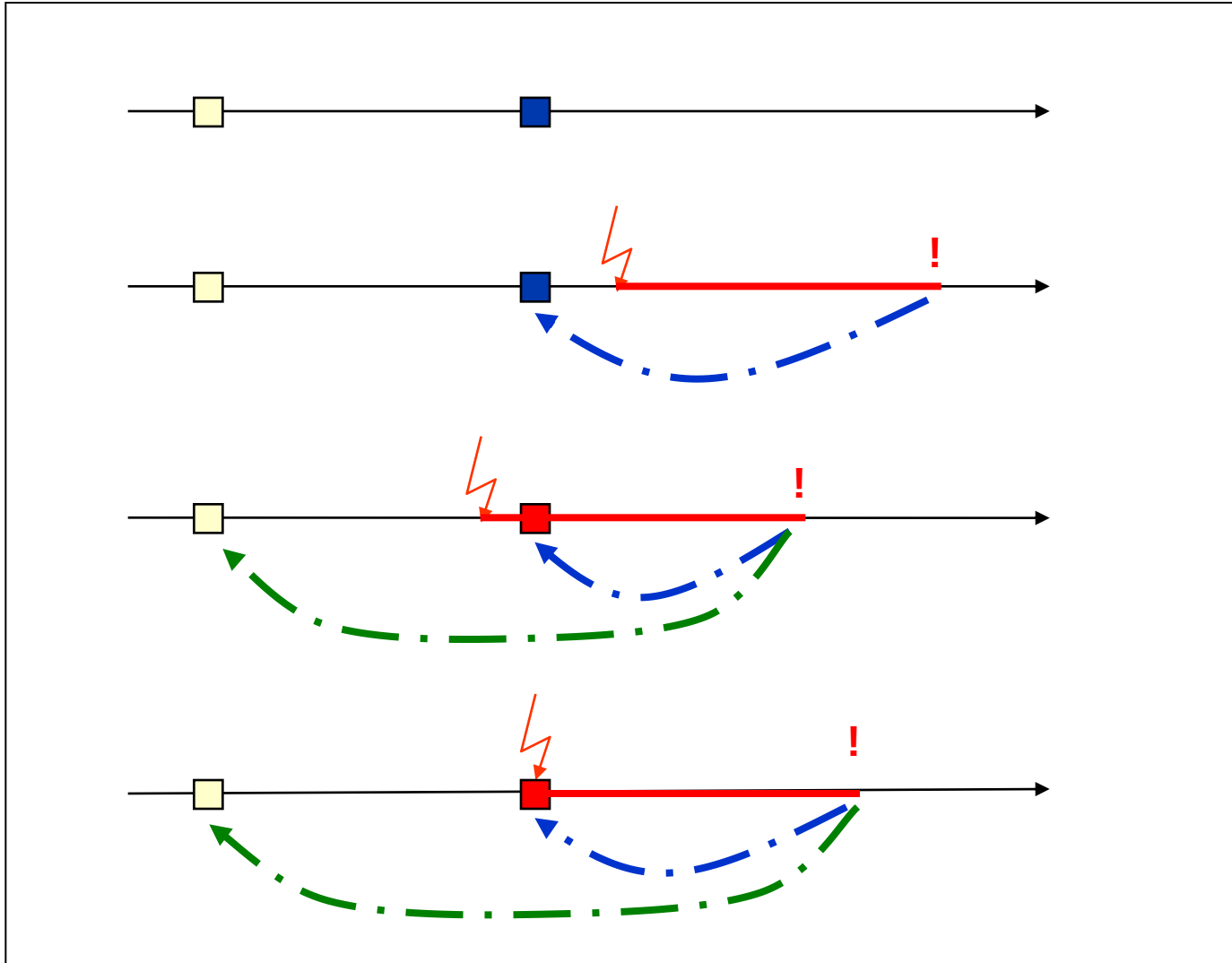
- **Checkpoint:** The saved state
- Checkpoint operations:
 - Saving the state: periodically, after messages; into stable storage
 - Recovery: restoring the state from the stable storage to memory
 - Discarding: after having more recent saved state(s)
- Analogy: “autosave”

■ Based on operation logs

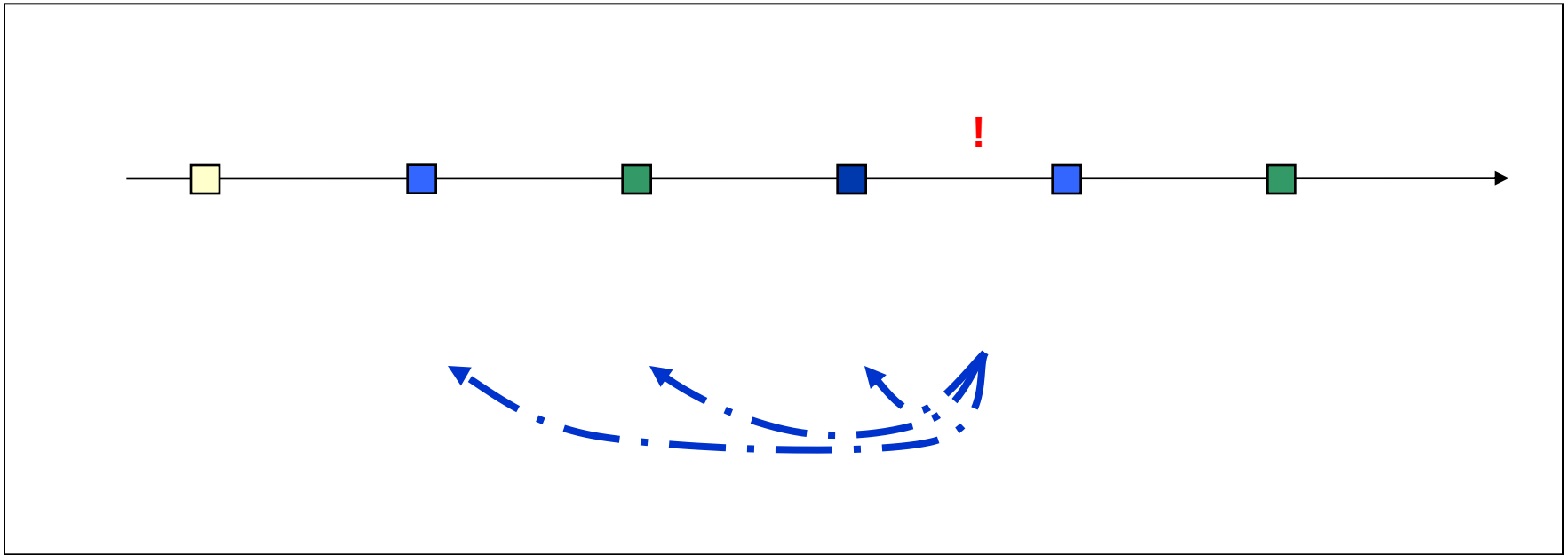
- Error to be handled: unintended operation
- Recovery is performed by the withdrawal of operations
- Analogy: “undo”

■ It is possible to combine the two mechanisms

Scenarios of backward recovery



Checkpoint intervals



Aspects of optimizing checkpoint intervals:

- Stable storage is slow (-> overhead) and has limited capacity
- Computation is lost after the last checkpoint
- Long error detection latency increases the chance of damaged checkpoints

The four phases of operation 4/4

4) Fault treatment and continuing service

■ Transient faults:

- Handled by the forward or backward recovery

■ Permanent faults:

Recovery becomes unsuccessful (the error is detected again)

The faulty component shall be localized and handled:

- Diagnostic checks to localize the fault
- Reconfiguration
 - Fault tolerance: Replacing the faulty component using redundancy
 - Degraded operation: Continuing only the safety related services
- Repair and substitution

4. Fault tolerance for software faults

- Repeated execution is not effective for design faults
- Redundancy with **design diversity** is required!

Variants: redundant software modules with

- diverse algorithms and data structures,
- different programming languages and development tools,
- separated development teams

in order to reduce the probability of common failures

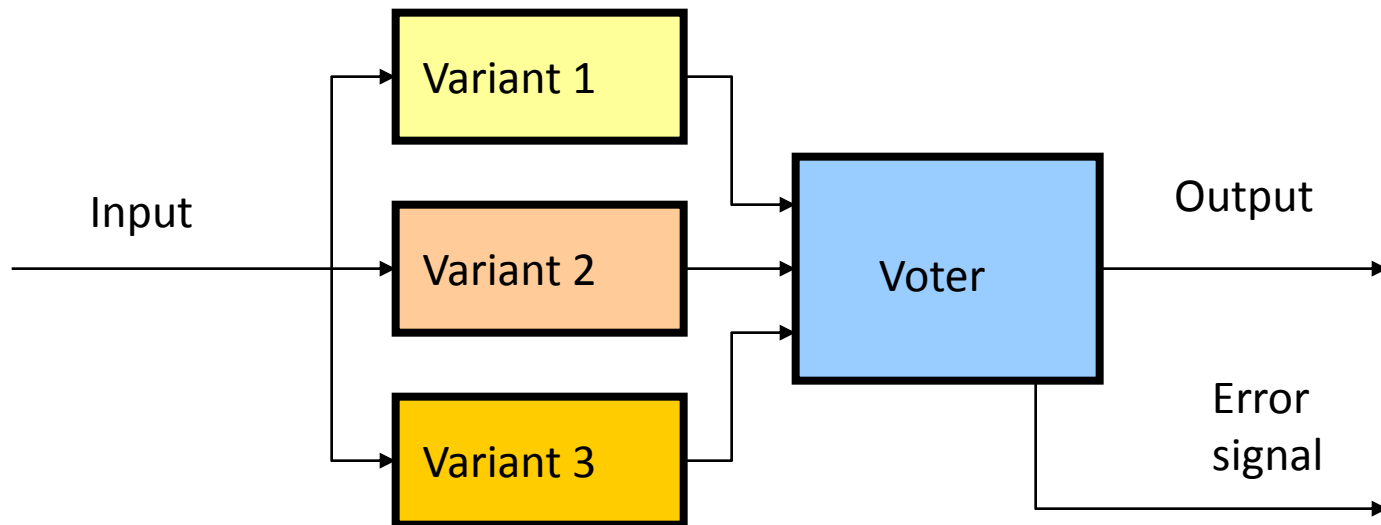
- Execution of variants:
 - N-version programming
 - Recovery blocks

N-version programming

- **Active** redundancy:

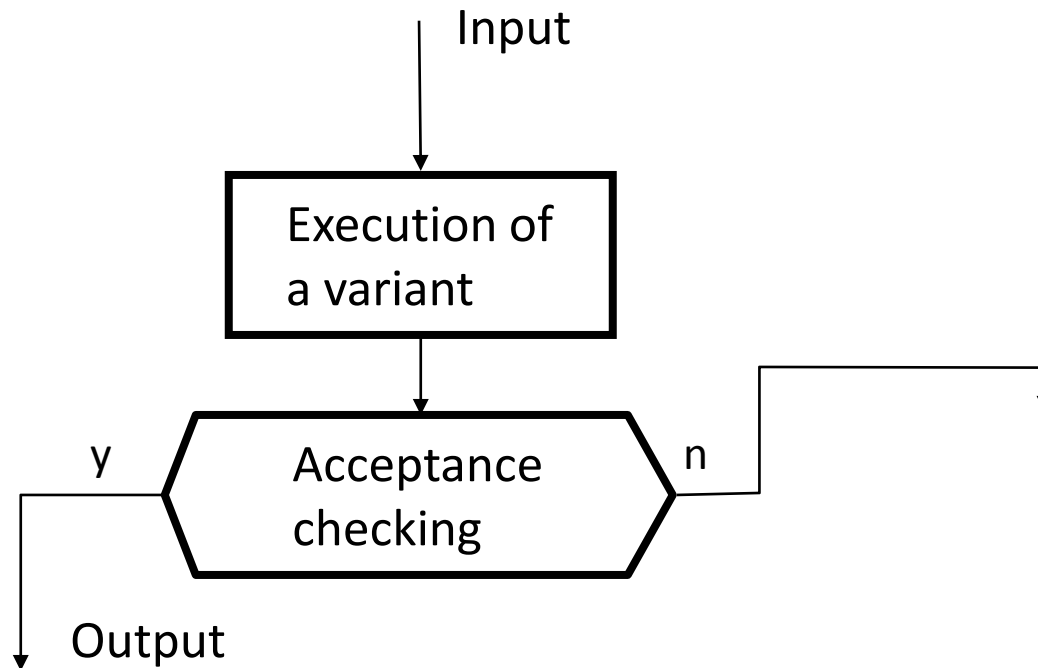
Each variant is executed (in parallel)

- The same inputs are used
- Majority voting is performed on the output
 - Acceptable range of difference shall be specified
 - The voter is a single point of failure



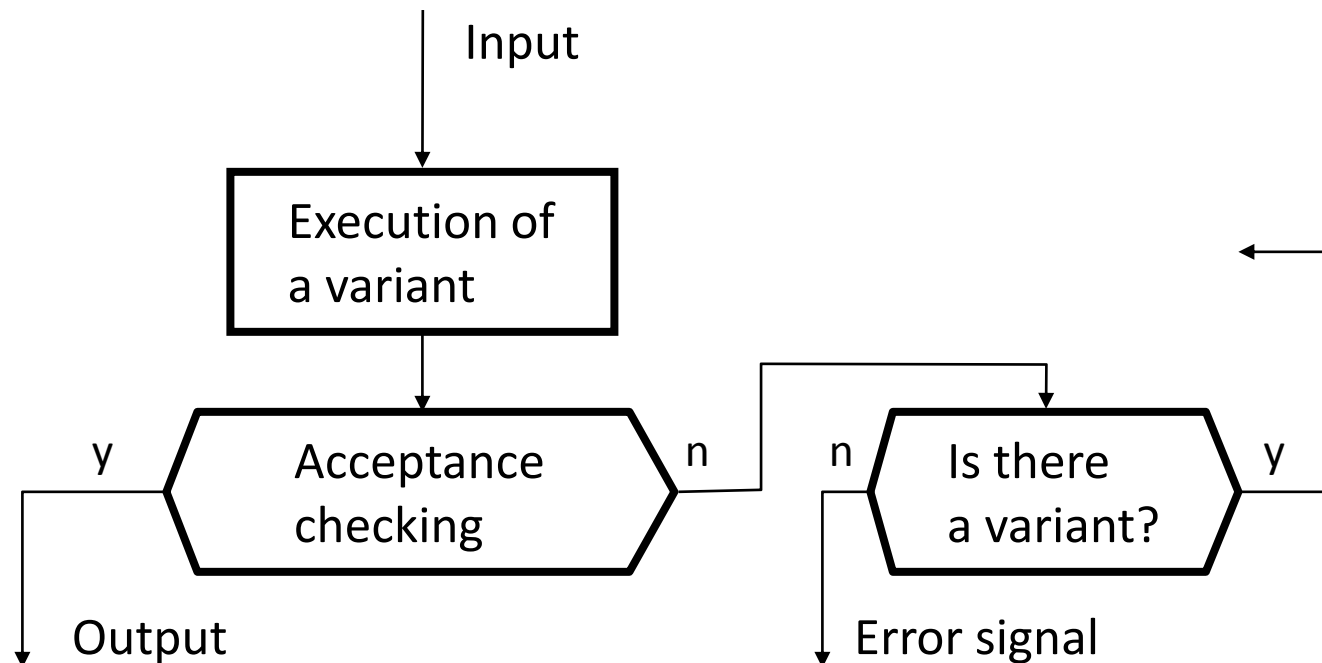
Recovery blocks

- **Passive** redundancy: Activation only in case of faults
 - The primary variant is executed first
 - **Acceptance checking** performed on the output of the variants
 - In case of a detected error another variant is executed



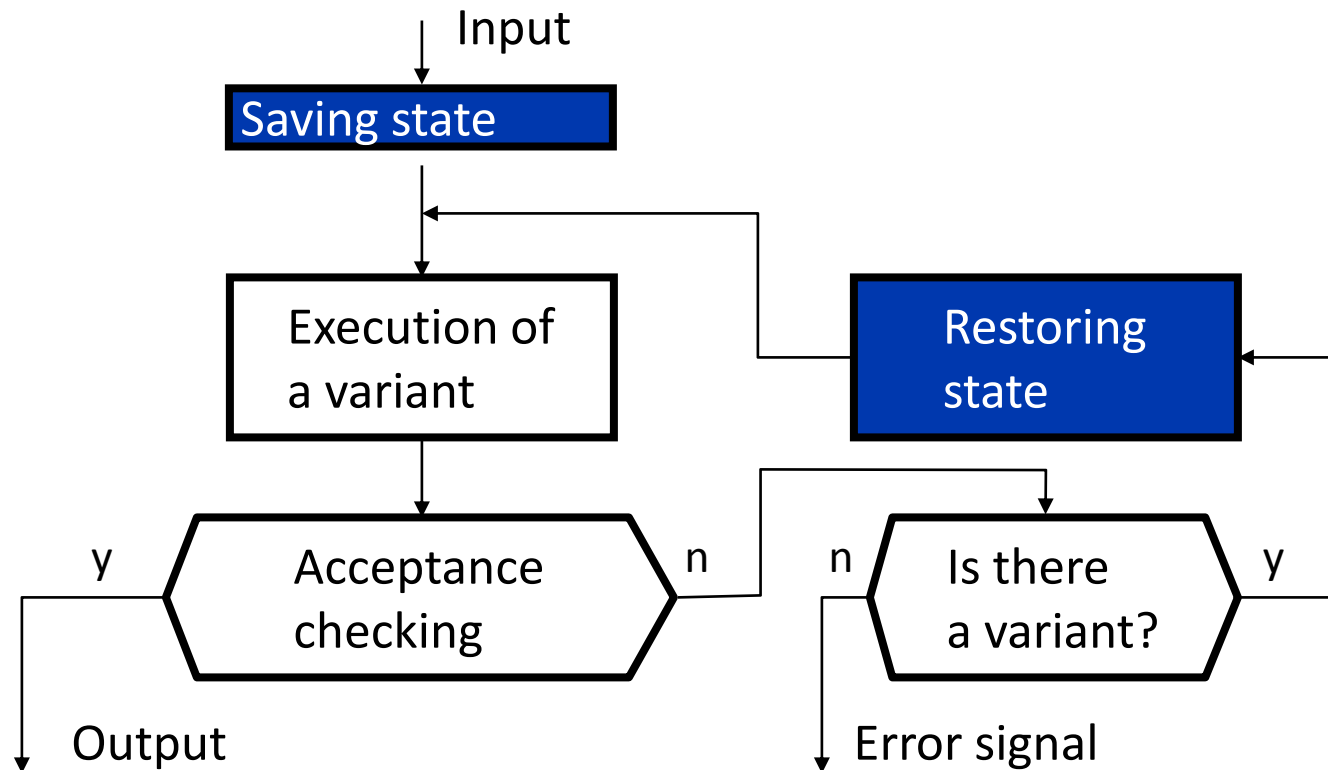
Recovery blocks

- **Passive** redundancy: Activation only in case of faults
 - The primary variant is executed first
 - **Acceptance checking** performed on the output of the variants
 - In case of a detected error another variant is executed



Recovery blocks

- **Passive** redundancy: Activation only in case of faults
 - The primary variant is executed first
 - **Acceptance checking** performed on the output of the variants
 - In case of a detected error another variant is executed

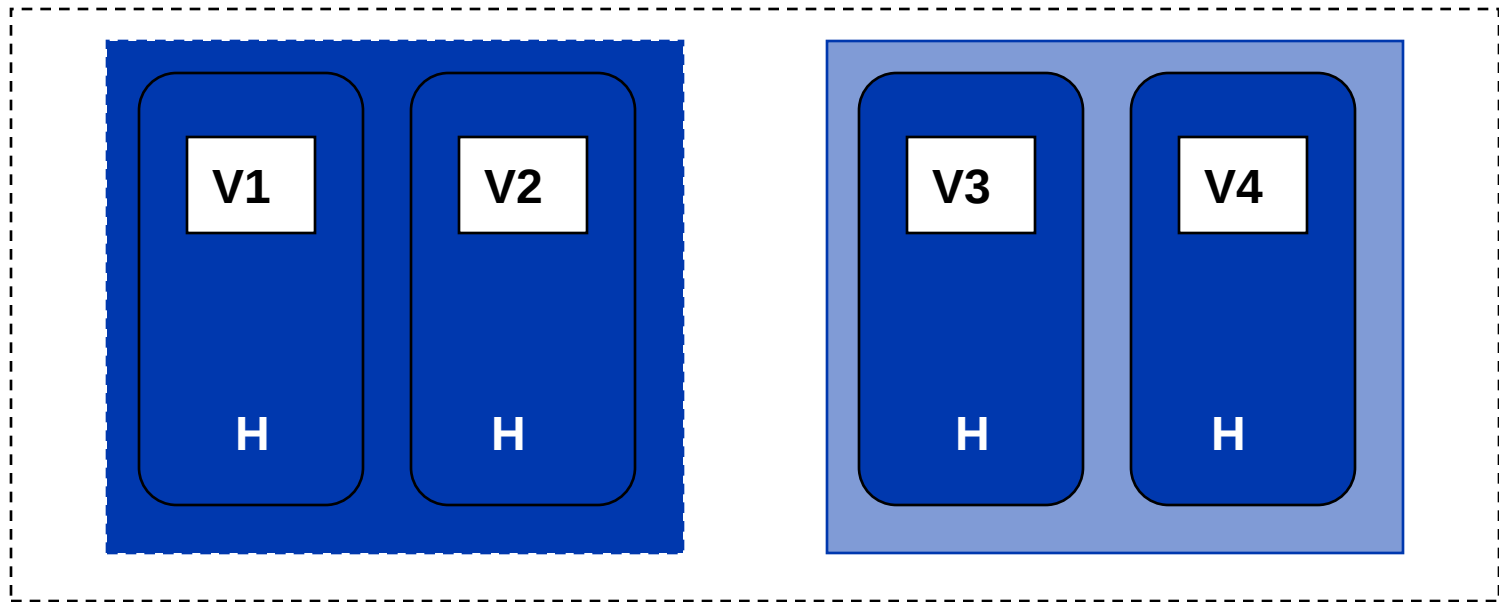


Comparison of the techniques

Property/Type	N-version prog.	Recovery blocks
Error detection	Majority voting, relative	Acceptance checking, absolute
Execution of variants	Parallel	Serial
Execution time	Slowest variant (or time-out)	Depending on the number of faults
Activation of redundancy	Always (active)	Only in case of fault (passive)
Tolerated faults	$[(N-1)/2]$	$N-1$
Fault handling	Masking	Recovery

Example: Airbus A-320, self-checking blocks

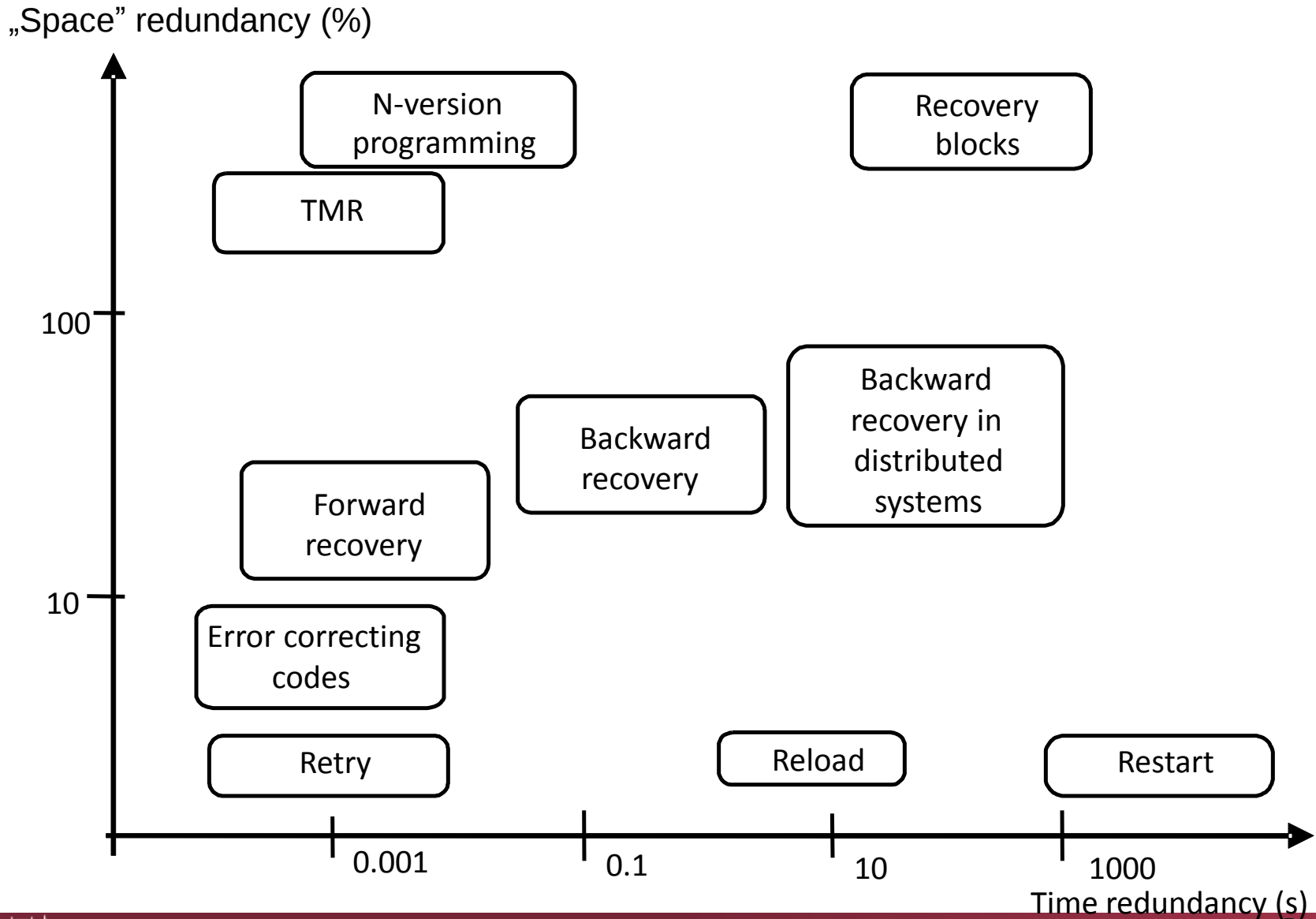
- Pair-wise self-checking execution
- Primary pair is active, switch-over in case of a fault
- Permanent hardware fault:
The pair with repeatedly detected fault will switch off



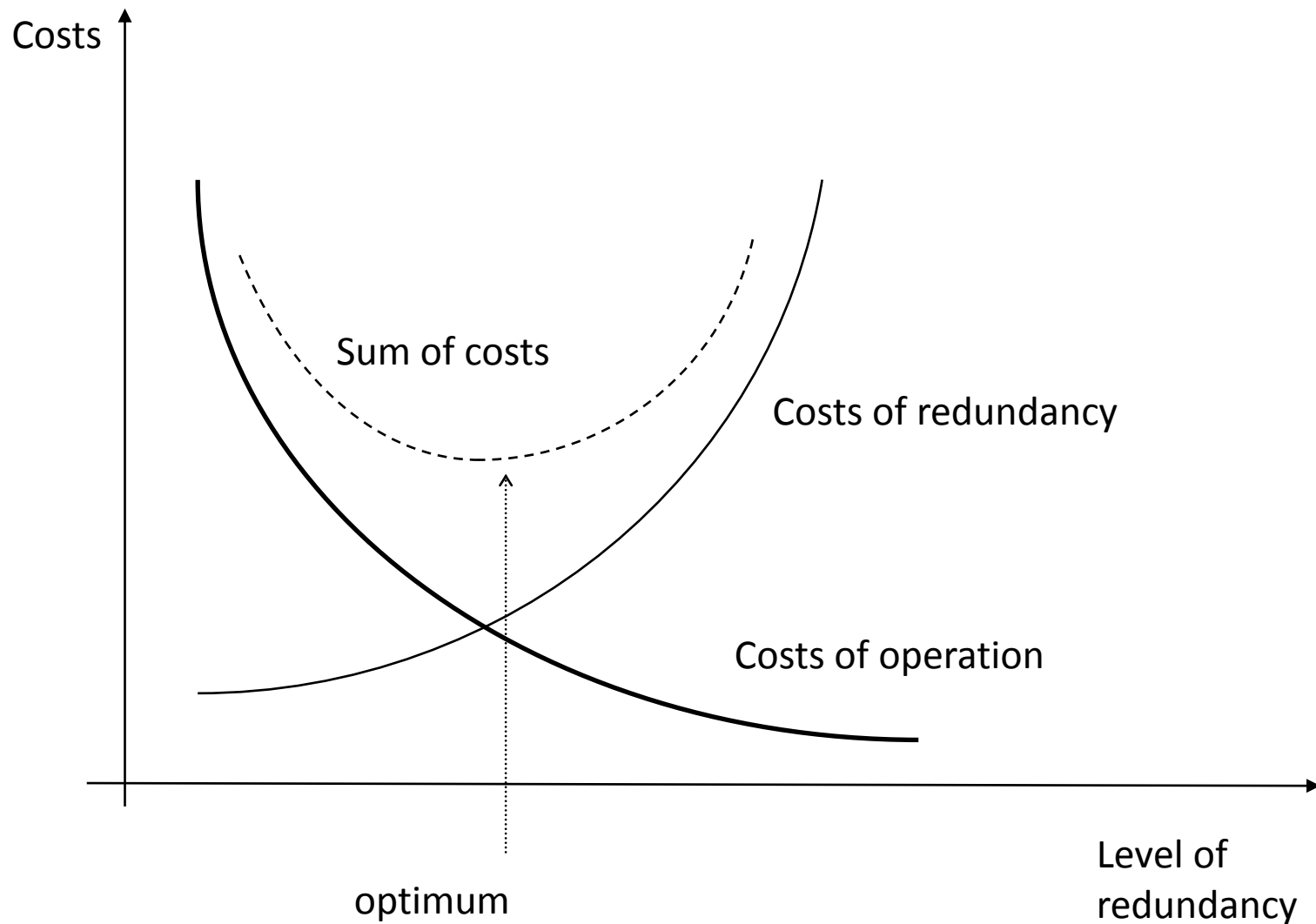
Summary



Redundancy in space (resources) and time



Costs of redundancy and operation (faults)



Summary: Types of redundancy

1. Hardware redundancy

- Replicas are used to tolerate permanent faults

2. Software redundancy

- Variants (NVP, RB) are used to tolerate design faults
- Software is used to tolerate transient hardware faults:
 - Forward recovery
 - Backward recovery

3. Information redundancy

- Faults in information storage and transfer are corrected by error correcting codes

4. Time redundancy

- Repeated execution is used in case of transient faults

Summary: Techniques of fault tolerance

1. Hardware design faults

- Diverse redundant components are used

2. Hardware permanent operational faults

- Replicated components are used

3. Hardware transient operational faults

- Software techniques for fault tolerance
 1. Error detection
 2. Damage assessment
 3. Forward or backward recovery (or compensation)
 4. Fault treatment
- Information redundancy: Error correcting codes
- Time redundancy: Repeated execution

4. Software design faults

- Variants as diverse redundant components (NVP, RB)