



Hardware-middleware support for adaptive and reliable MPSoCs

Leandro Fiorin (USI)
leandro.fiorin@usi.ch

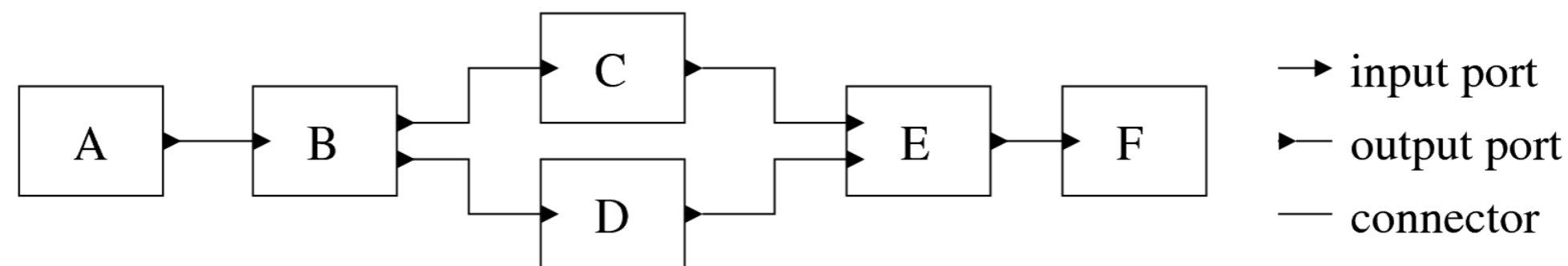
**“Though this be madness, yet there is method in't”
Hamlet Act 2, scene 2, 193-206**

- Adaptivity and fault tolerance: the MADNESS' objectives
- MADNESS: short background
- Fault models and fault-tolerance techniques
- Fault tolerant strategies for Networks-on-Chip
- Fault tolerant strategies for processing elements
- Future activities

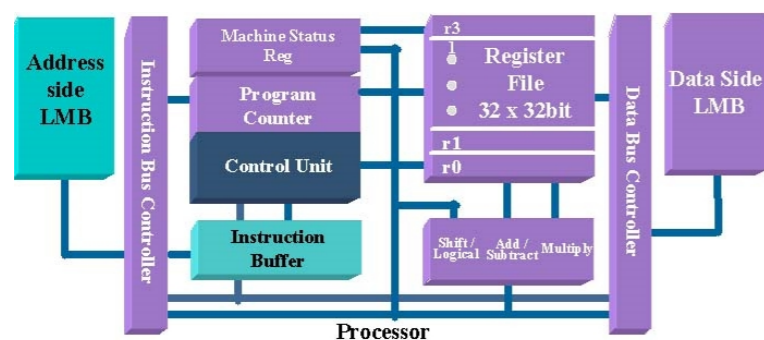
- QoS demands of applications, as well as demands in terms of dependability of the system, require a certain degree of adaptivity of the system.
- Need to extend the static approach to embedded MPSoC design for addressing adaptivity requirements: new methodologies and corresponding tool support for adaptive (i.e., run-time) mapping of application tasks to the underlying architecture resources.
- Research at several levels:
 - **scenario-based DSE** incorporating the notion of adaptive behavior at all levels;
 - **middle-ware support** – using e.g. the developed run-time DSE mechanisms – for realizing run-time mapping in actual MPSoC implementations prototyped on FPGAs;
 - **hardware support** for adaptivity, allowing run-time monitoring and management of HW resources, as well as facilitating rapid migration of tasks (threads) from one component to another at run-time;
 - **runtime reconfigurability** of IPs e NoC.

- Probability that some manufacturing defect will escape end-of-production testing or that faults will become evident during normal operation has to be taken into account.
- “Traditional” approaches (involving massive redundancy) hardly adoptable in constrained devices (e.g. embedded devices).
- Due to time-to-market requirements, solutions leading to continued availability in the presence of faults must be developed within the normal design flow.
- **Objectives:**
 - To build a system-level methodology supporting continuity of service by providing self-checking policies and system-level reconfigurations policies.
 - To define and implement an optimal architecture for the monitoring and detection of faults in the behavior of the system (including a self-monitoring of elements of the NoC).
 - To define reconfiguration methodologies reacting to hard faults affecting the NoC, as the ones incapacitating other IPs present in the MPSoC, with the final target of dynamically mapping the application onto the MPSoC.

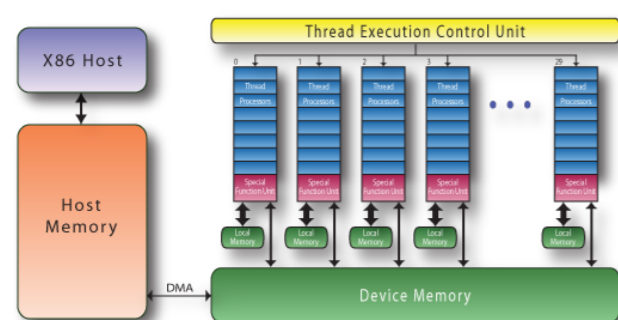
- Applications modeled as Kahn Process Networks (KPNs).
- A KPN is a stream-oriented model of computation based on the idea of organizing an application into streams and computational blocks.
- Streams represent the flow of data, while computational blocks represent operations on a stream of data.
- A KPN application consists of a set of parallel running tasks (application components) connected through non-blocking write and blocking read unbounded FIFO queues (connectors) that carry tokens between input and output ports of application components.
- A token is a typed message.



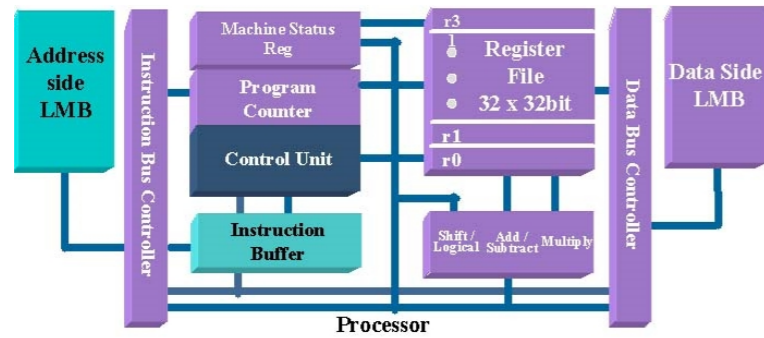
- During the MADNESS project, we distinguish between three main types of components:



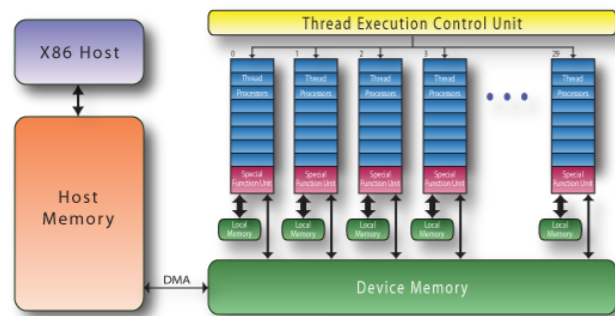
Processing elements



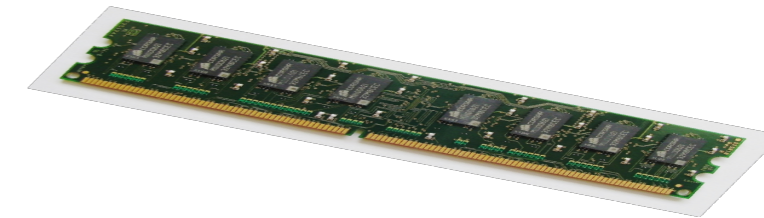
- During the MADNESS project, we distinguish between three main types of components:



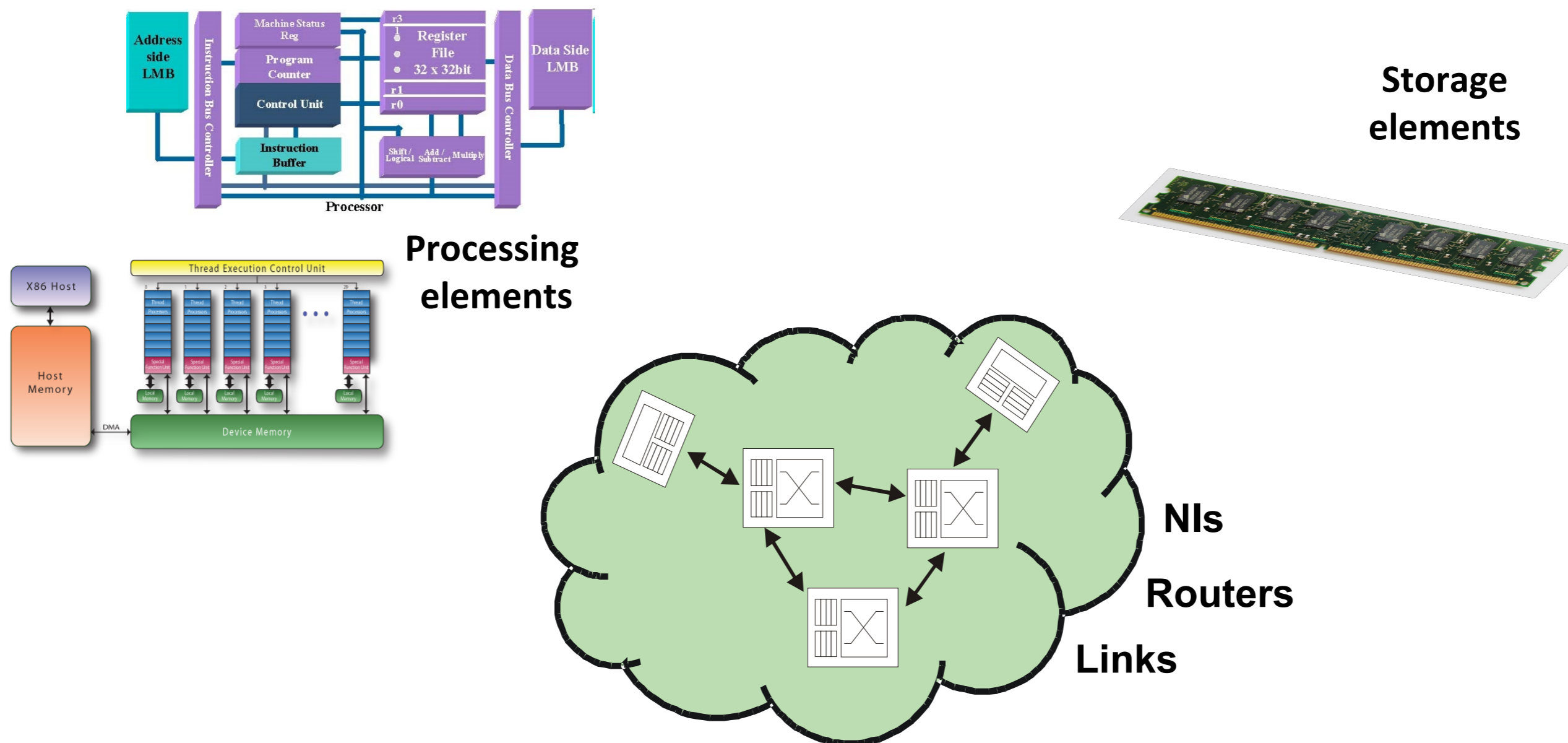
Processing elements



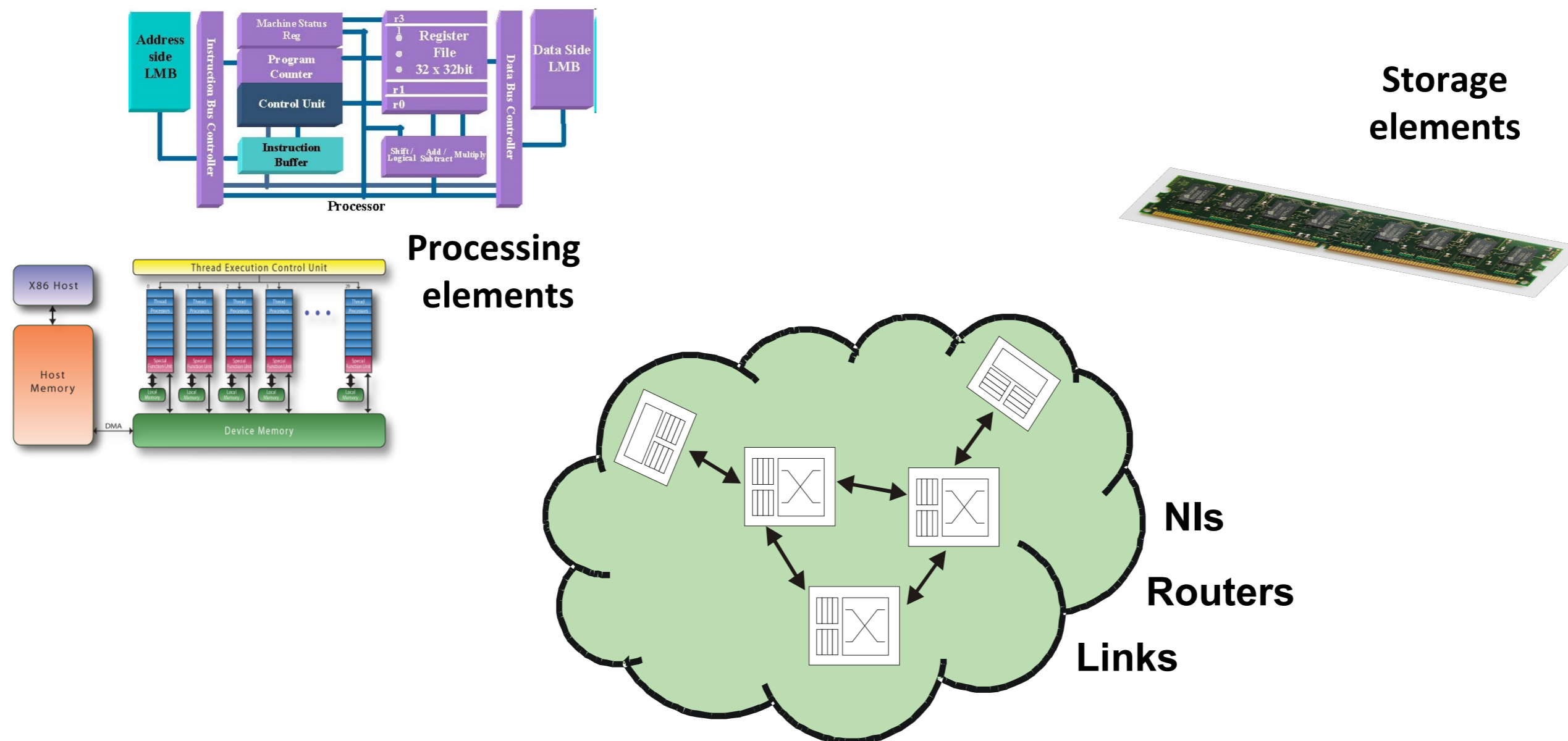
Storage elements



- During the MADNESS project, we distinguish between three main types of components:

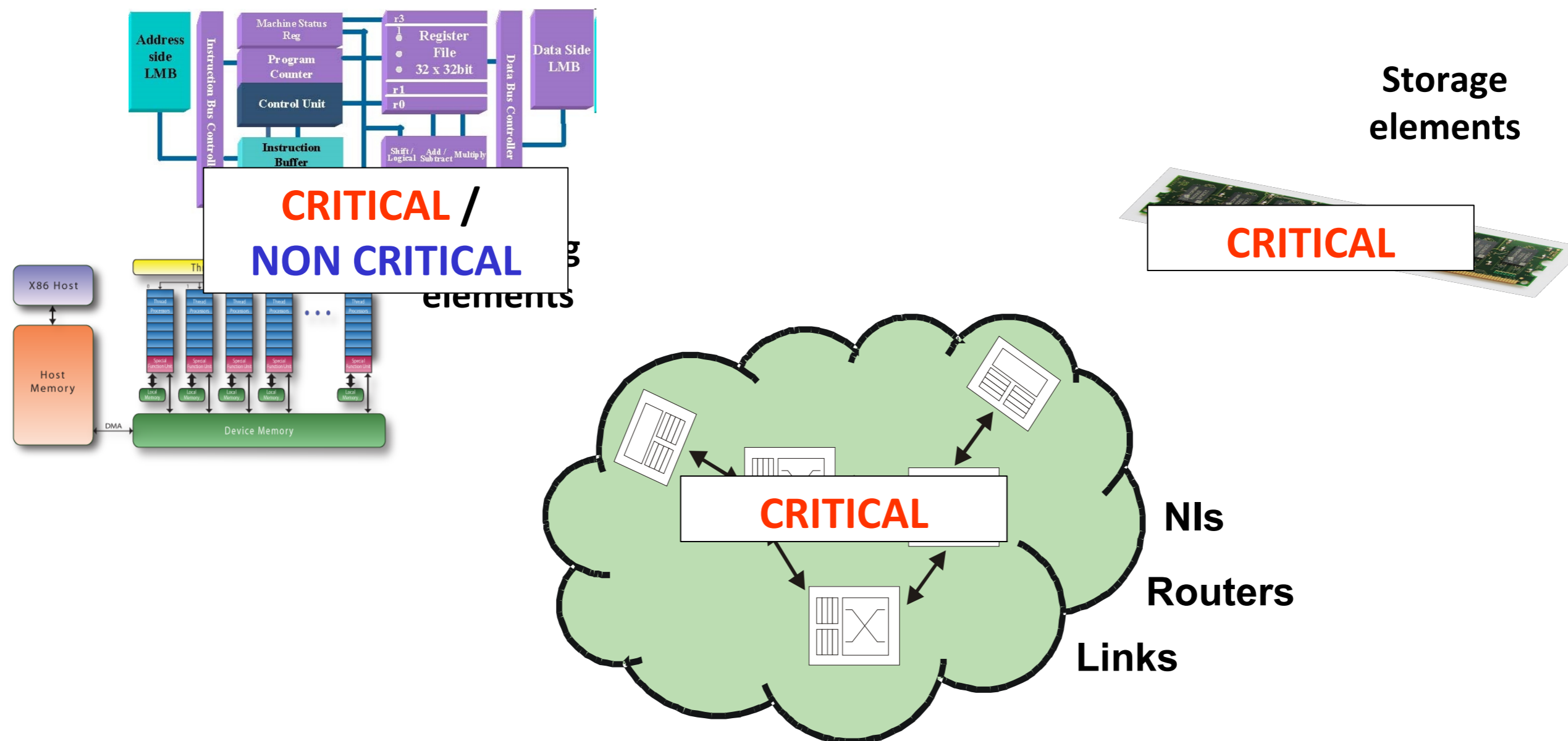


- During the MADNESS project, we distinguish between three main types of components:



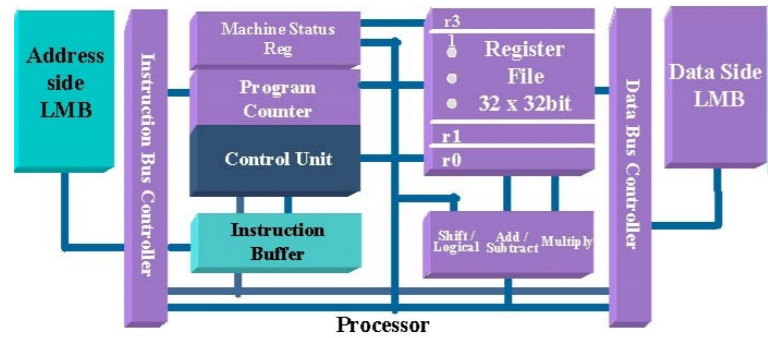
- **Assumptions:** single-fault (for individual unit or set of units), hard-faults vs soft-faults, critical vs non critical, acceptable error propagation for non-critical components.
- **CRITICAL:** concurrent self-checking.
- **NON CRITICAL:** semi-concurrent self-checking/self-testing.

- During the MADNESS project, we distinguish between three main types of components:

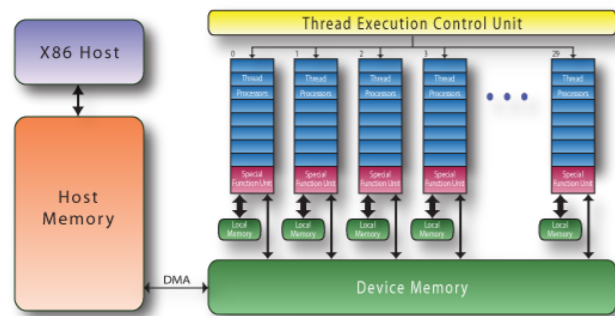


- **Assumptions:** single-fault (for individual unit or set of units), hard-faults vs soft-faults, critical vs non critical, acceptable error propagation for non-critical components.
- **CRITICAL:** concurrent self-checking.
- **NON CRITICAL:** semi-concurrent self-checking/self-testing.

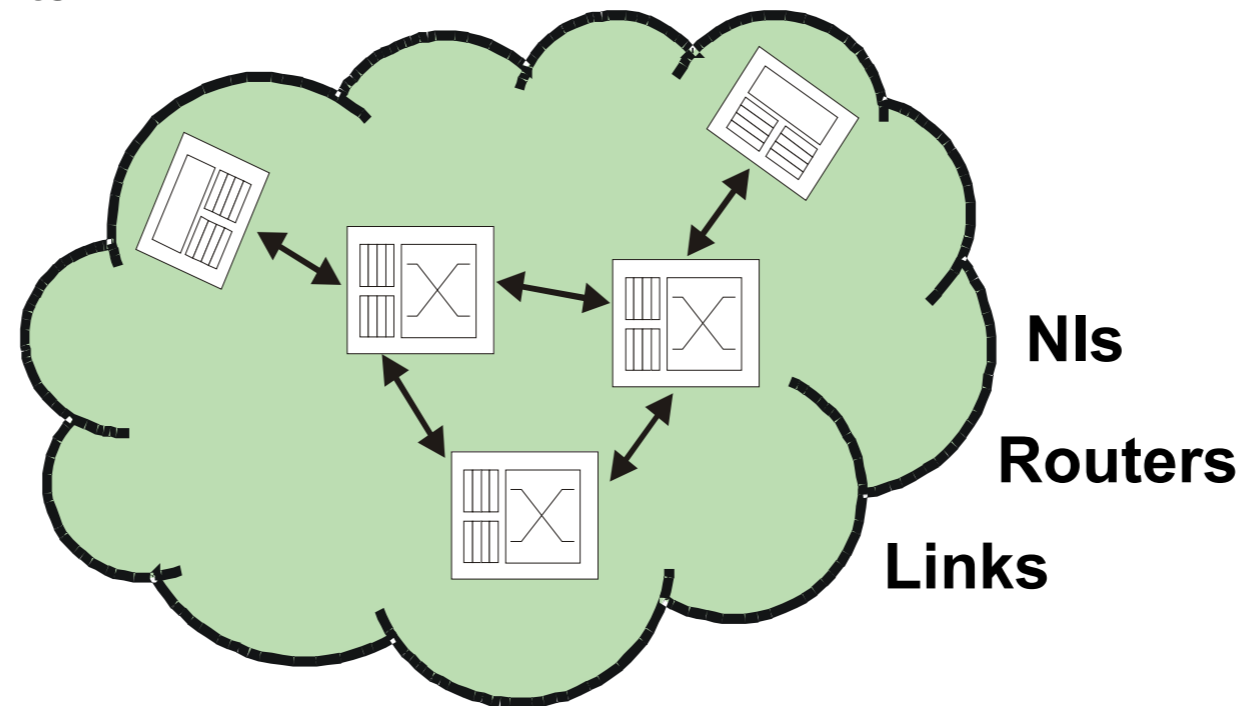
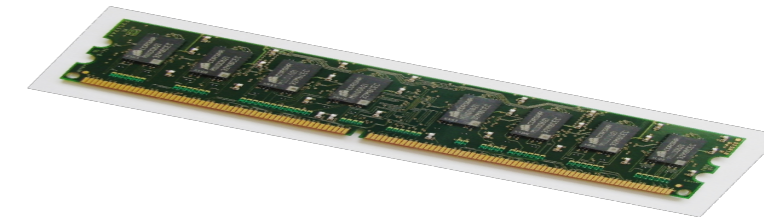
- Self-checking and fault-tolerance techniques:



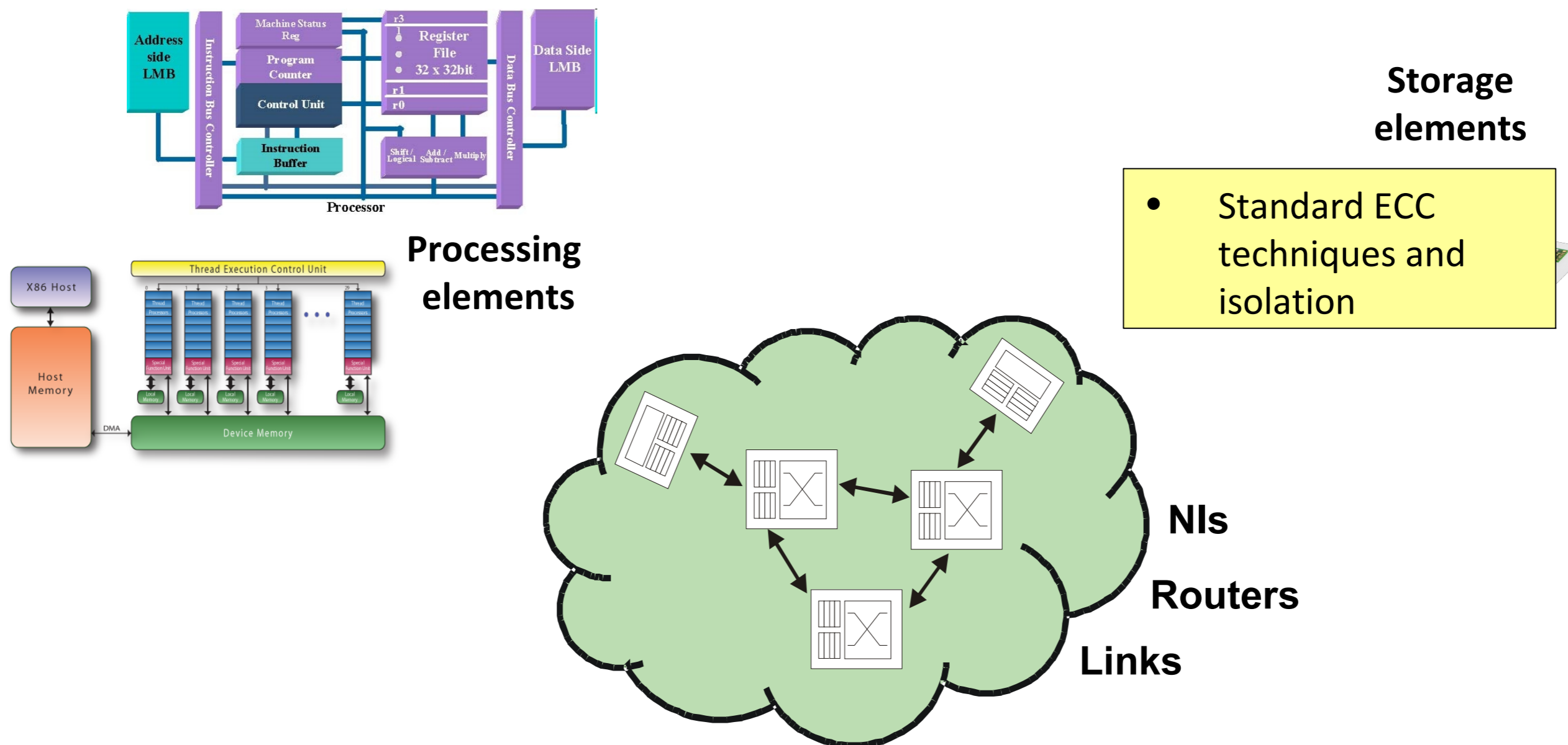
Processing elements



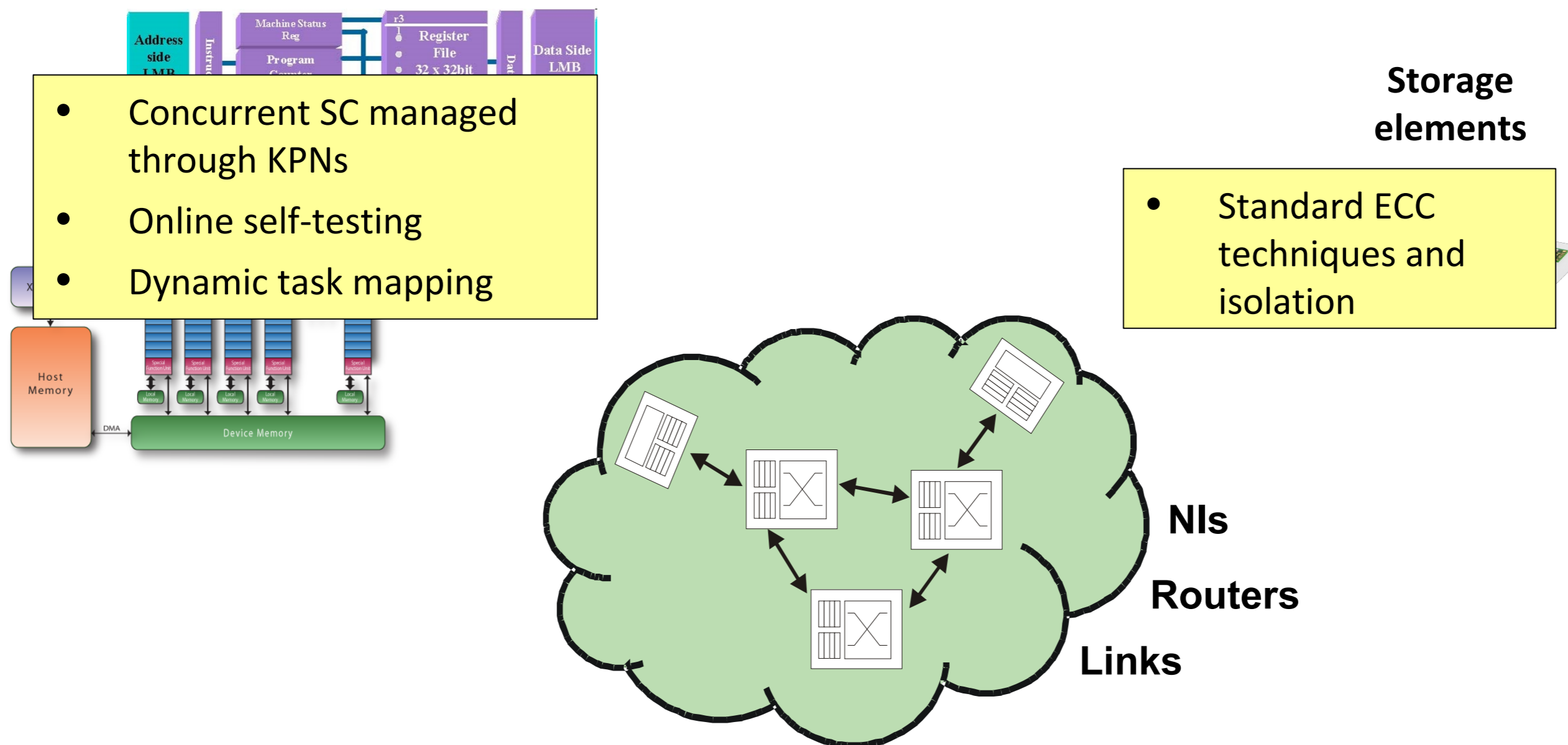
Storage elements



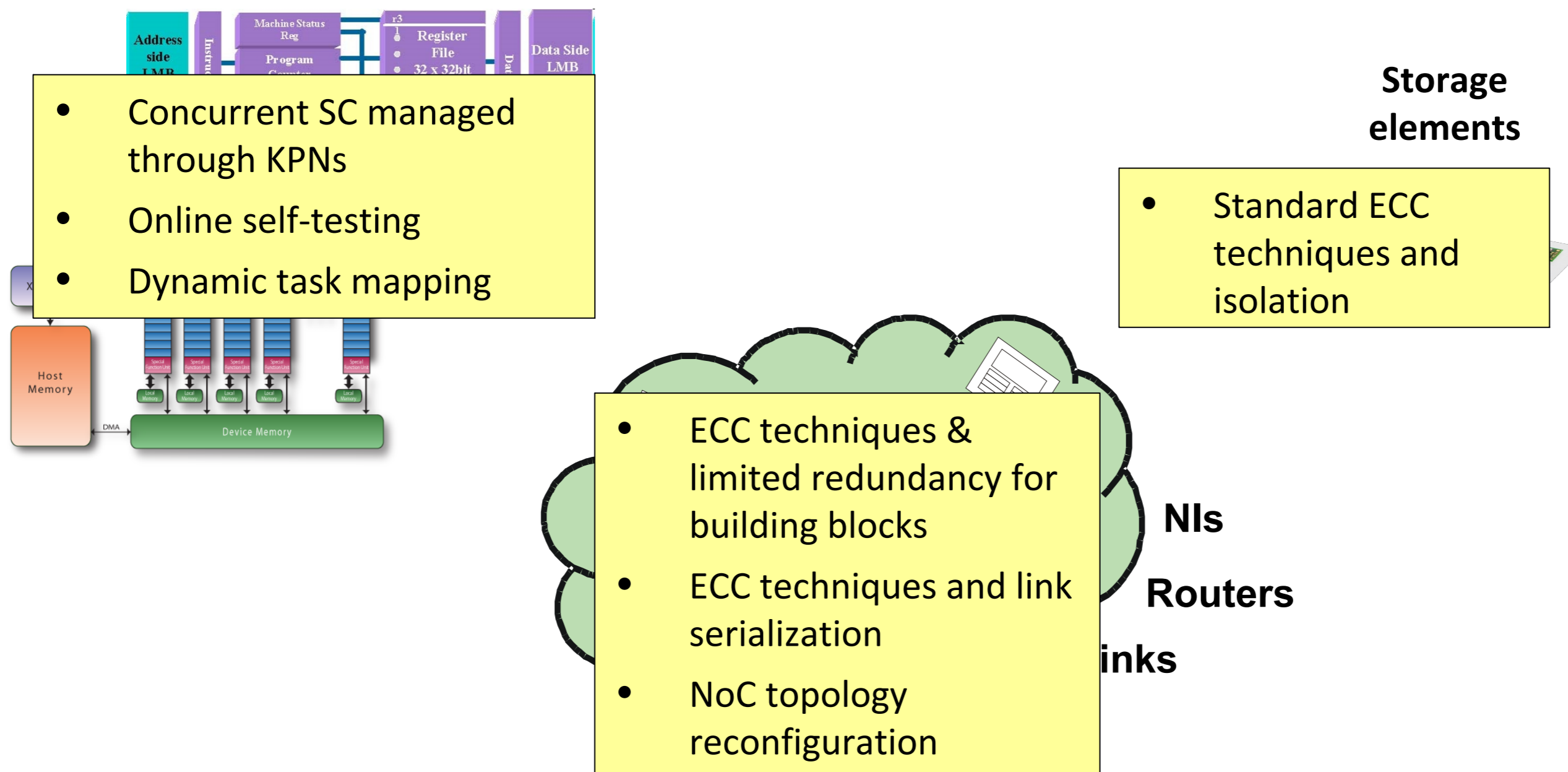
- Self-checking and fault-tolerance techniques:



- Self-checking and fault-tolerance techniques:



- Self-checking and fault-tolerance techniques:



- Correct behavior is essential in NoCs for supporting higher level reconfiguration strategies.
- The on-chip communication network plays the important role of supporting the exchanges of information between nodes.
- Continuity of service is needed to guarantee the possibility of executing correctly applications and high level system services, such as system adaptivity and reconfiguration.

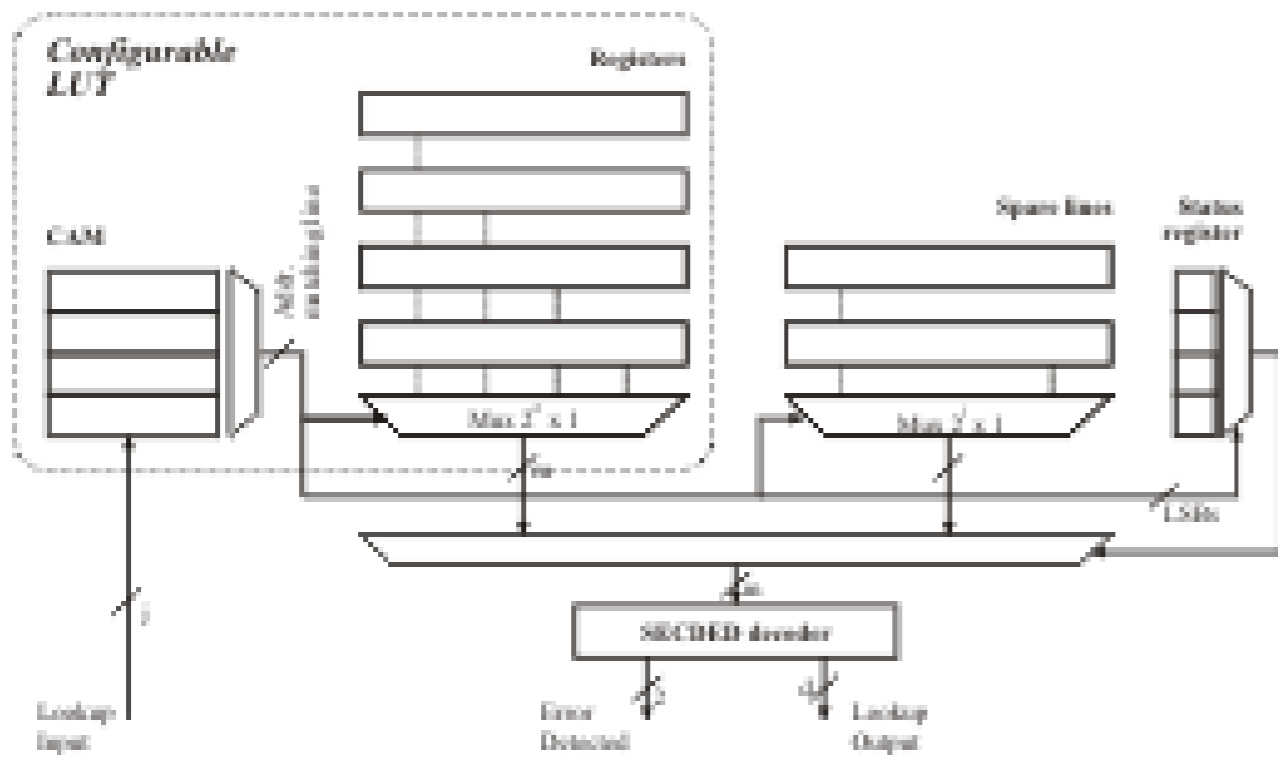
- Correct behavior is essential in NoCs for supporting higher level reconfiguration strategies
- The on-chip communication network plays the important role of supporting the exchanges of information between nodes
- Continuity of service is needed to guarantee the possibility of executing correctly applications and high level system services, such as system adaptivity and reconfiguration.

CRITICAL

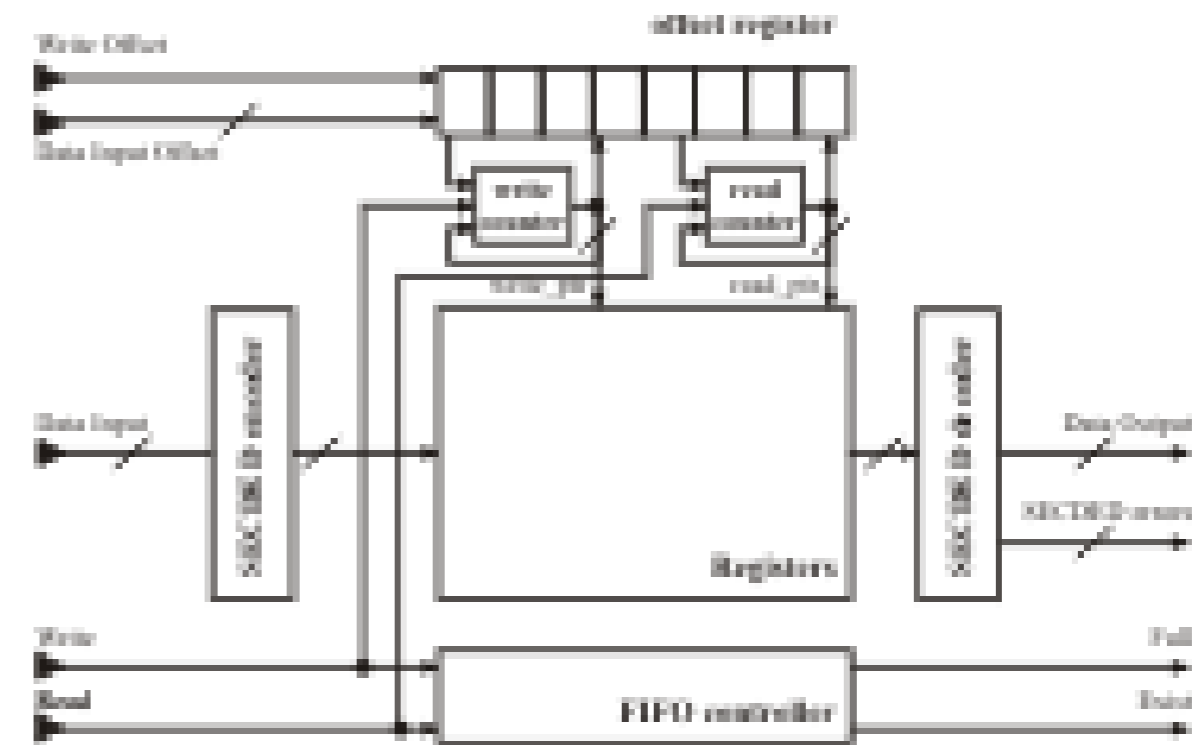
- **Concurrent self-checking:** use of error correcting and detecting codes.
- **Limited architectural redundancy:** use of intrinsic redundancy of NoCs elements, as well architectural techniques for increasing resilience at components' level.
- **Reconfiguration at different levels:** run-time reconfiguration of architectural components, as well as of routing information.

NoCs - preliminary results: the network interface

- **Self-checking and fault tolerant architecture for NIs**
 - NI is a critical point in the NoC, and faults in it can directly affect all the system;
 - We analyzed main architectural components of NI (LUT, FIFOs, FSMs), and proposed a self-checking and fault tolerance strategy as a combination of ECC (SECDED) and limited redundancy.



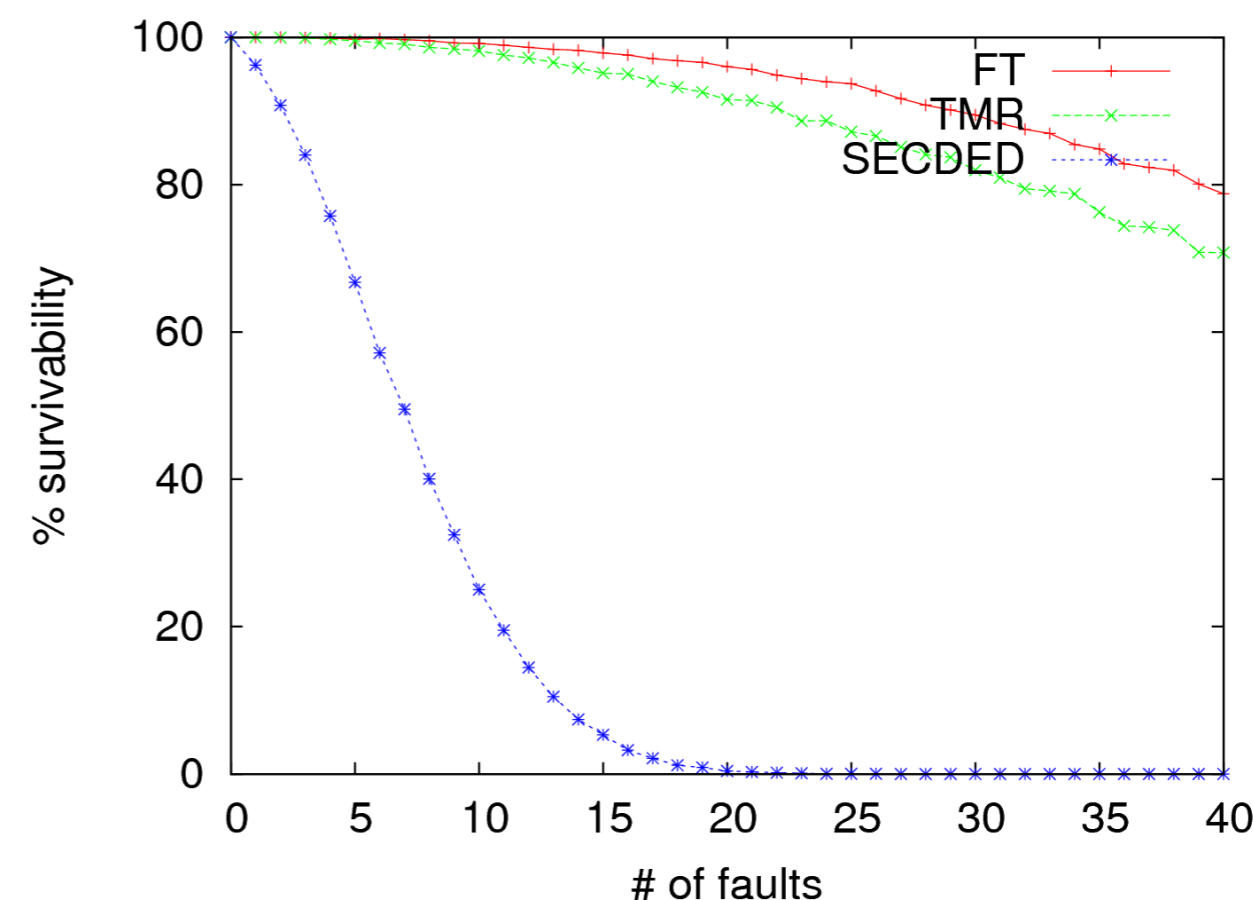
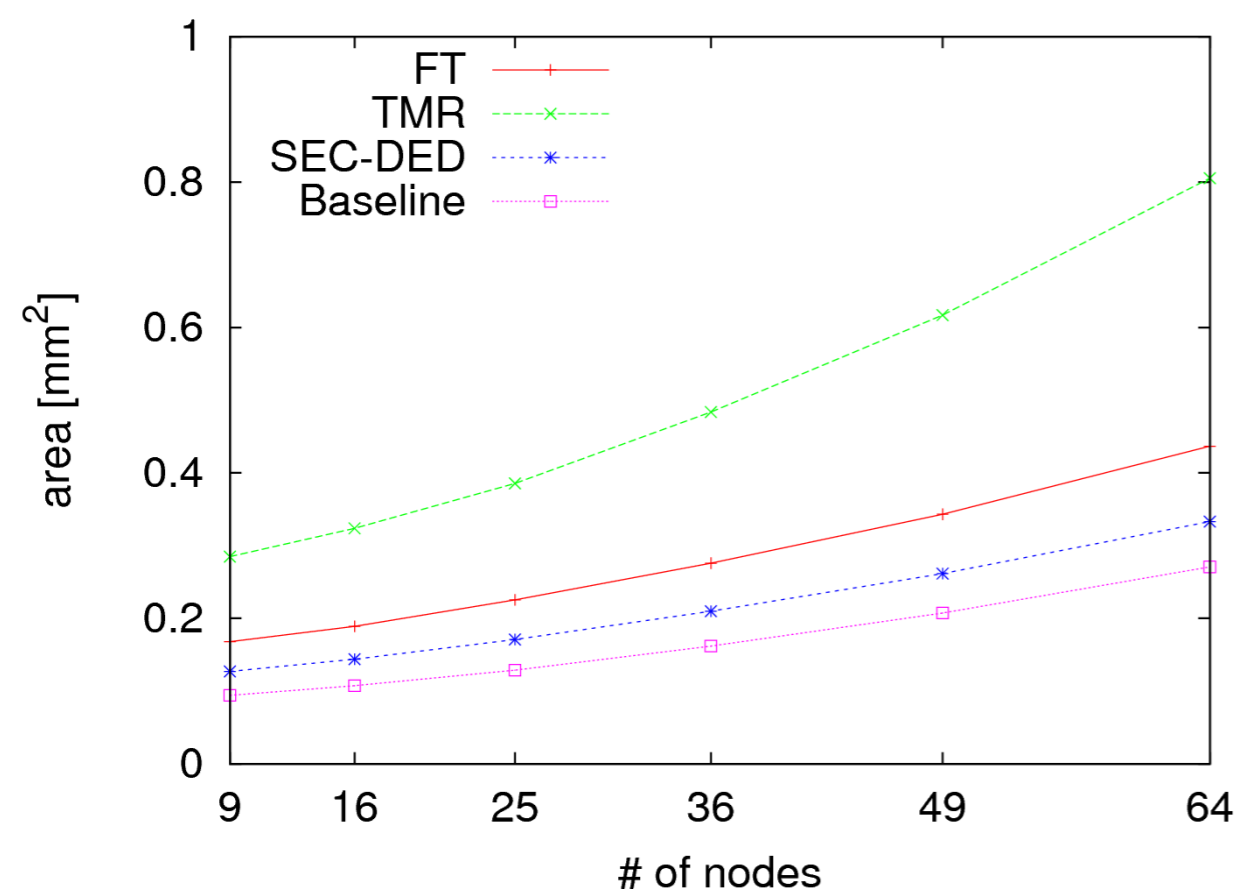
a) LUT



b) FIFO

- **Preliminary results:**

- Reduction of area and energy consumption with respect to a classical fault tolerant TMR implementation, while obtaining similar survivability to faults of the NI;
- Definition of reconfiguration policies for the NI activated at error detection.



- Programmable vs non-programmable processing elements.
- Depending on the application, different possible requirements and the possibility to allow a certain level of error propagation.

- Programmable vs non-programmable processing elements.
- Depending on the application, different possible requirements and the possibility to allow a certain level of error propagation.

CRITICAL

- **Concurrent self-checking:** use of fault tolerance patterns applied to KPNs.
- **Dynamic task remapping:** run-time reallocation of task running on faulty processing elements.

- Programmable vs non-programmable processing elements.
- Depending on the application, different possible requirements and the possibility to allow a certain level of error propagation.

CRITICAL

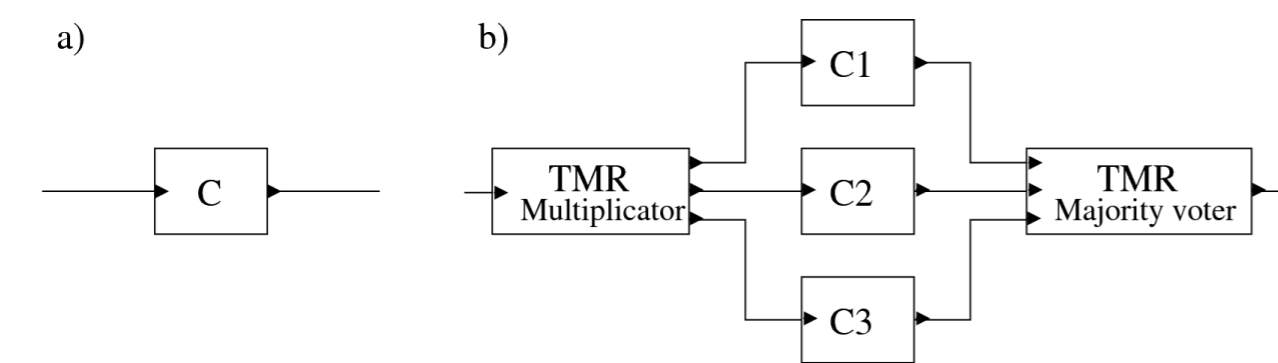
- **Concurrent self-checking:** use of fault tolerance patterns applied to KPNs.
- **Dynamic task remapping:** run-time reallocation of task running on faulty processing elements.

NON CRITICAL

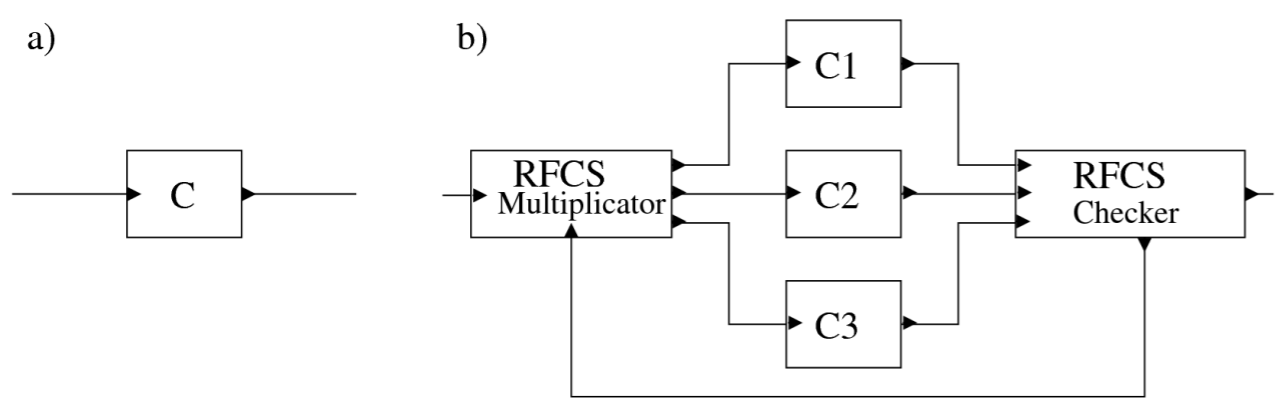
- **Online self-testing:** scheduling of self-testing software routines for detection of permanent faults.
- **Dynamic task remapping:** run-time reallocation of task running on faulty processing elements.

- **Self-checking techniques at KPN level**

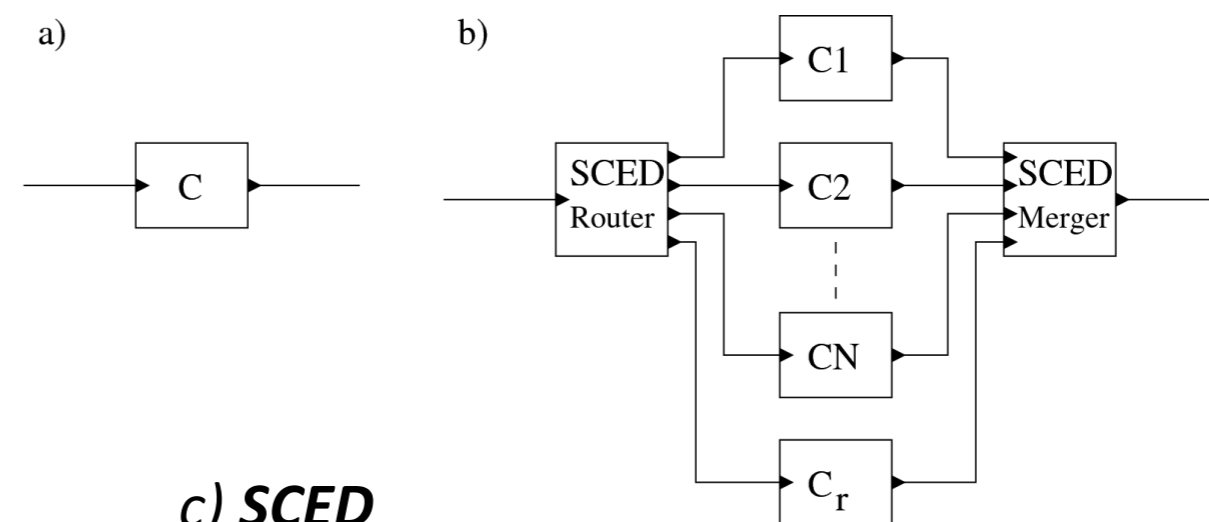
- Different fault tolerance patterns (Triple Modular Redundancy, Roll-Forward Check-Pointing, Semi-Concurrent Error Detection) can be used to detect operativeness of processing elements, as well as providing methods for detecting and correcting errors due to faulty behaviors of components.
- The idea is to transform original KPN task graph according to these patterns and obtain fault tolerant applications.



a) TMR

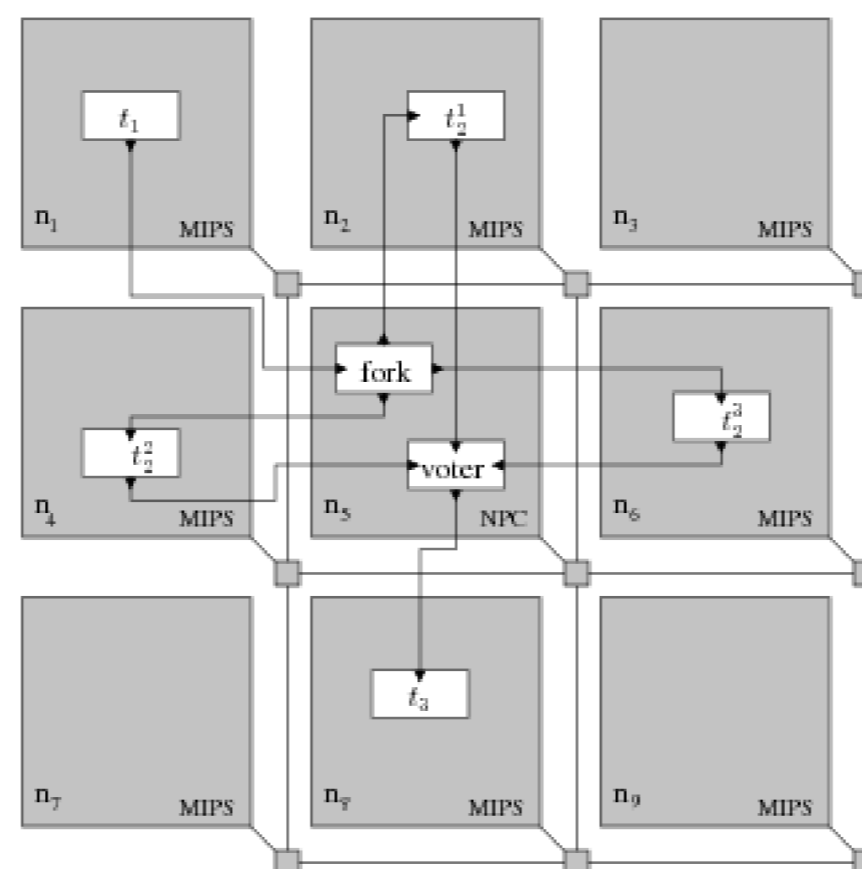


b) RFCS



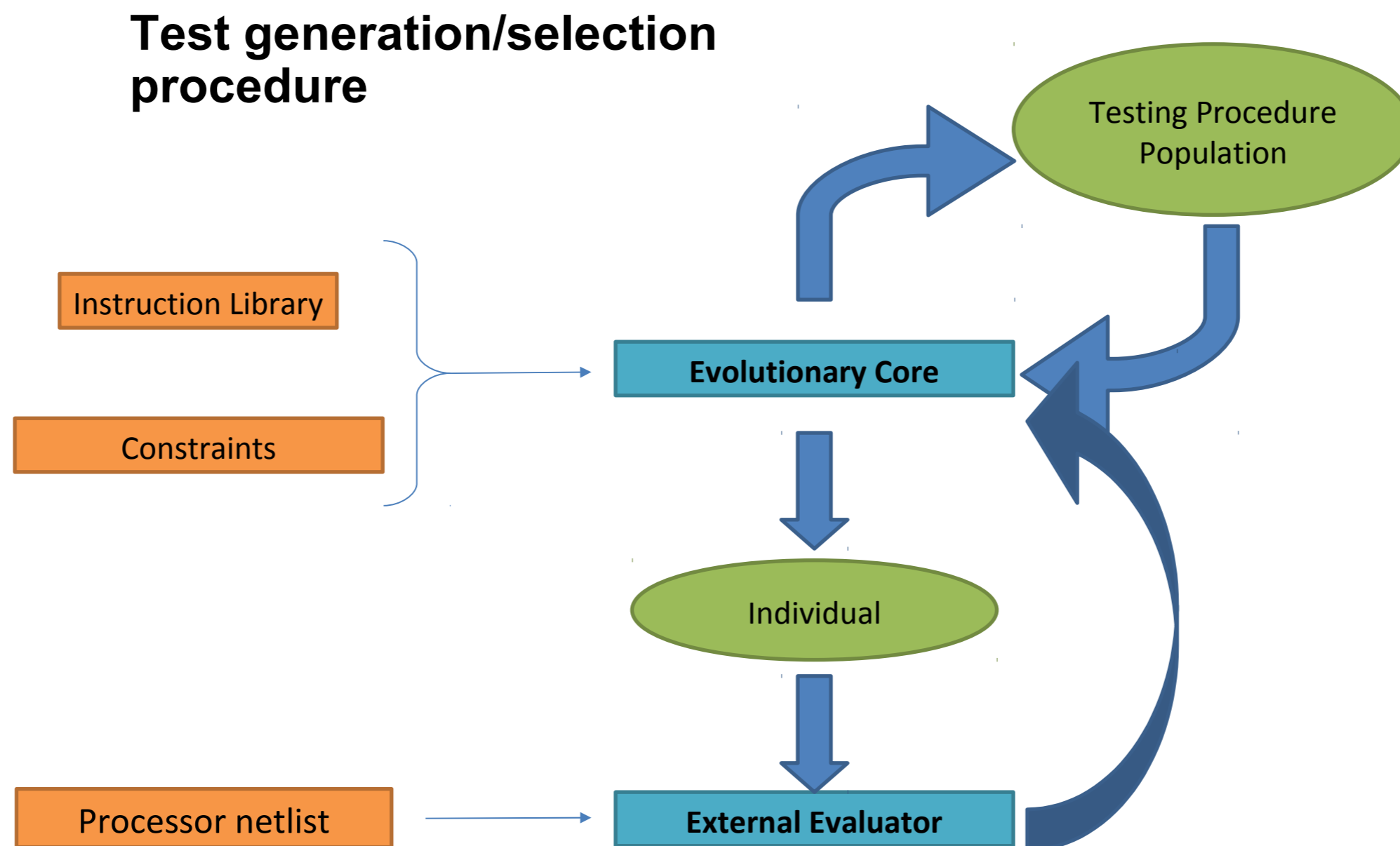
c) SCED

- Investigation of these techniques with respect to their:
 - MTTF:**
 - Create a fault tree given the application specification and mapping information;
 - Calculate MTTF using binary decision diagrams.
 - throughput and communication cost**



Possible mapping of an application with TMR pattern onto a 3x3 NoC

- Automatic generation of software test routines for generic programmable cores.
- Execution of self-checking routines supported by ad-hoc hardware modules.
- Specific software routines have been studied for exploiting detection of faults in reconfigurable cores.

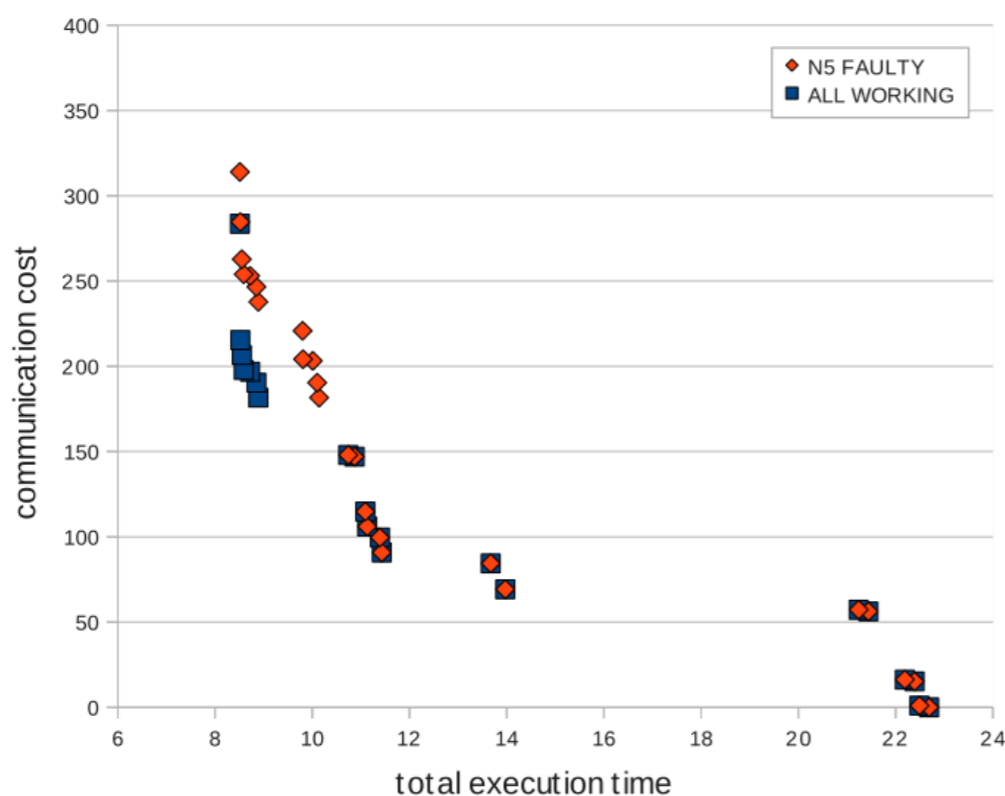


- Self-testing methodologies for programmable cores:
 - **Evolutionary core:** creates a software testing procedure starting from a prologue, an epilogue, and filling the middle part with random instruction taken from the instruction library;
 - **Inputs:** evolutionary core randomly generates individuals picking instruction up from a XML file containing the processor instruction set architecture.
 - **External evaluator:** generates the fitness function for each of the procedure within the population;
 - **Inputs:** fault coverage simulated with Tetramax (Synopsys); Processor netlist to be provided for allowing gate level simulation.
- Analysis of architecture details for defining self-testing routines, in particular for reconfigurable processors.

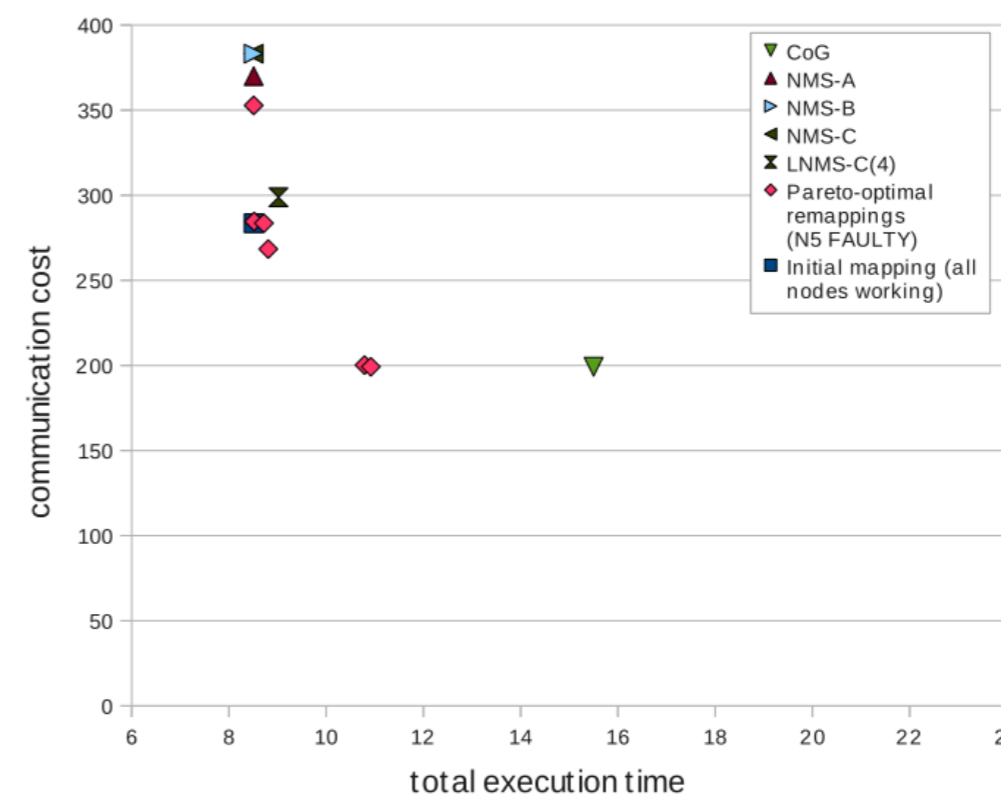
- **Online task remapping strategies**
 - Tasks reallocation represents an alternative solution to the classical use of resource redundancy, exploiting intrinsic availability of spare computation resources.
 - Up to now we addressed three main aspects:
 - Definition of a method for finding an optimal solution to mapping tasks onto heterogeneous NoC MPSoCs, based on an ILP formulation of the problem;
 - Definition of an online solution to the task remapping problem in presence of run-time faults based on heuristics;
 - Middleware support for task remapping.

- **Preliminary results:**

- Definition of several heuristics (CoG, NMS-A/B/C, LNMS-A/B/C) for the limited reconfiguration case;
- Degradation in performance measured for all single fault scenarios within 6%, in case of a MPEG2 decoder application mapped on a 3x3 mesh NoC (LNMS-C heuristic).

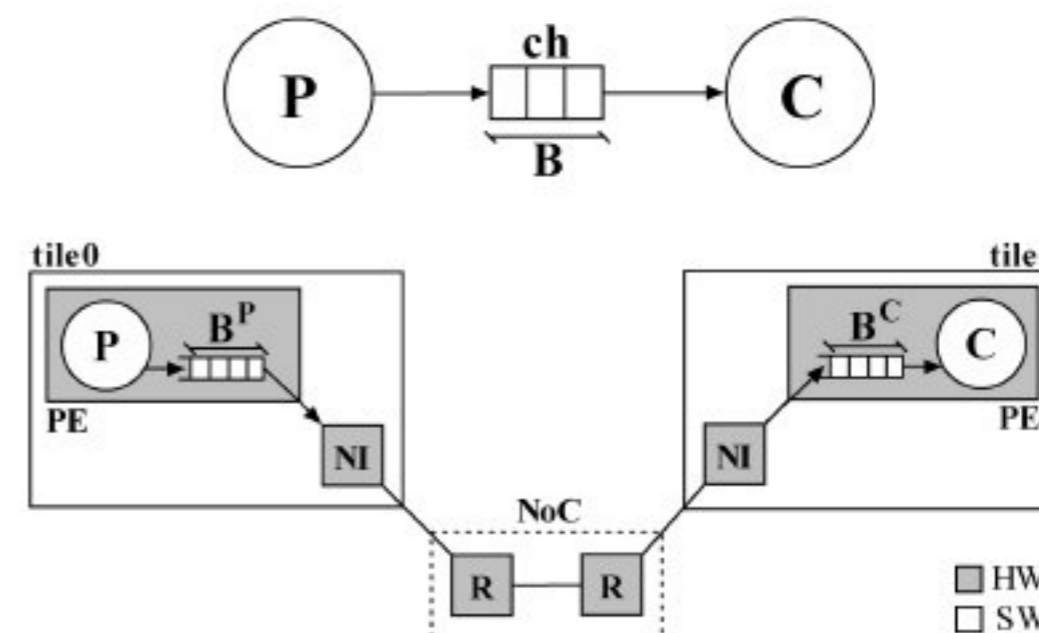
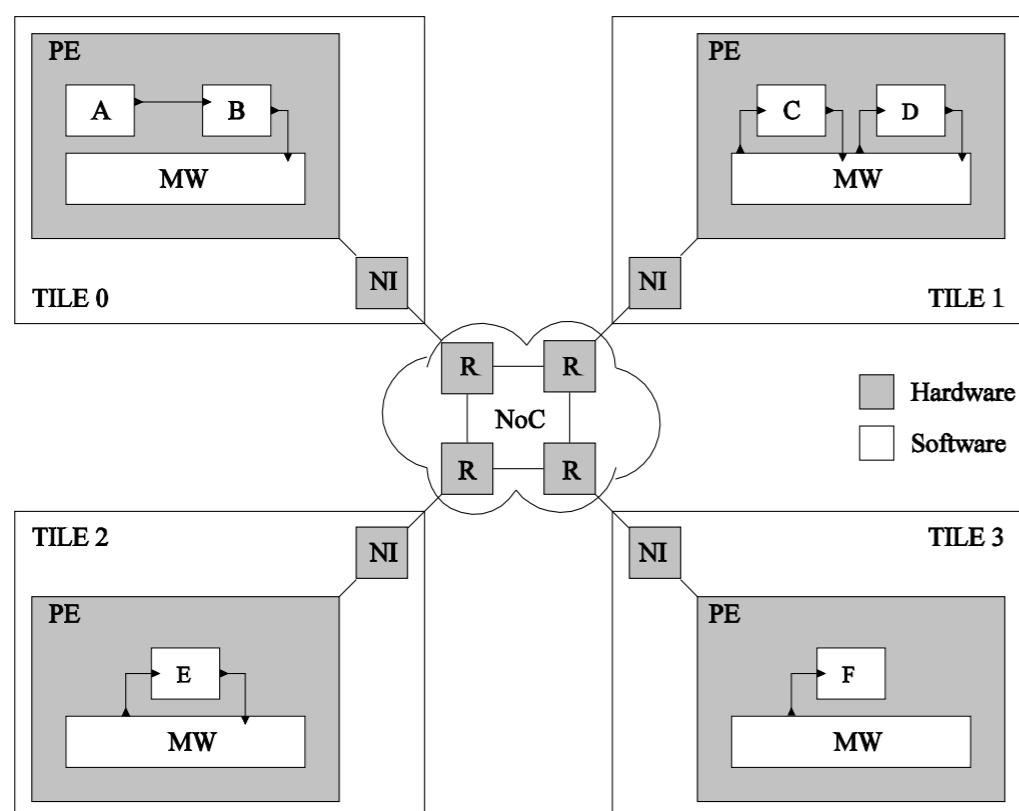


(a) Pareto curves for optimal mappings and optimal remappings (unlimited task migration)



(b) Comparison of results between heuristics, optimal remappings and the initial mapping (limited task migration)

- The middleware is a software layer that runs on the programmable cores and as a hardware wrapper for non-programmable cores.
- The main functionality of the middleware is to implement KPN semantics for the application tasks that are either software or hardware tasks.
- Remapping is performed activating and stopping threads by using the underlying operating system support and by changing the dedicated middleware tables (source and destination tiles for each channel of the KPN are updated).



- In case of faulty processing elements, task remapping cannot be supported by the faulty core...
- Task migration supported by an external module.

- **Development of module- and system-level self-checking policies**
 - Implementation of self-checking and fault-tolerant techniques for NoC routers and links.
 - Assessment and evaluation of self-checking techniques at KPN level.
 - Implementation of distributed software techniques for online self-testing of cores.

- **Development of module- and system-level self-checking policies**
 - Implementation of self-checking and fault-tolerant techniques for NoC routers and links.
 - Assessment and evaluation of self-checking techniques at KPN level.
 - Implementation of distributed software techniques for online self-testing of cores.
- **Definition of system-level reconfiguration policies**
 - Extension analysis of task remapping strategies.
 - Definition and implementation of policies for NoC topology reconfiguration.

- **Development of module- and system-level self-checking policies**
 - Implementation of self-checking and fault-tolerant techniques for NoC routers and links.
 - Assessment and evaluation of self-checking techniques at KPN level.
 - Implementation of distributed software techniques for online self-testing of cores.
- **Definition of system-level reconfiguration policies**
 - Extension analysis of task remapping strategies.
 - Definition and implementation of policies for NoC topology reconfiguration.
- **Integration of the fault tolerant support into the FPGA**
 - Integration of designed modules and methodologies within the FPGA environment.
 - Testing of self-checking and fault methodologies on the static platform.
 - Testing and validation of the reconfiguration policies on the dynamic platform.

A stylized graphic on the left side of the text. It features a blue brain with a spiral pattern, a blue vertical line, and a brown chair-like shape at the bottom.

Thank you!
Madness
Questions?

leandro.fiorin@usi.ch