Efficient predictability in Manycore systems for Real-time

francois.pecheux@lip6.fr, UPMC/LIP6/SOC
dumitru.potop_butucaru@inria.fr, INRIA

Presentation outline

- Adaptive architectures
- Predictive architectures
- FPTA

Adaptive architectures

- **Introduction**
- 2D-Mesh NoC & Shared Memory MP2SoC
- NoC Test Strategy
- DCCI & Black Hole Detection
- Experimental Results
- Conclusion

Introduction

Future Architecture (ANR ARFU, good old days)

- Massively Manycore Chips: Network-on-Chip (NoC) Based, Shared Memory
- Fault-tolerance issue: Handling permanent faults:
  - In manufacture
  - On the field (The chip has been integrated in the final equipment)

Fault-tolerance : On the field, Detection, Deactivation and Reconfiguration (ODDR)

- Detect each NoC component status
- Deactivate the faulty ones
- Reconfigure the NoC routing function

ODDR of NoC in MP2SoC

1 issue to solve:

Diagnose and locate the faulty/de-activated components.

Outline

- Introduction
- **2D-Mesh NoC & Shared Memory MP2SoC**
- NoC Test Strategy
- DCCI & Black Hole Detection
- Experimental Results
- Conclusion

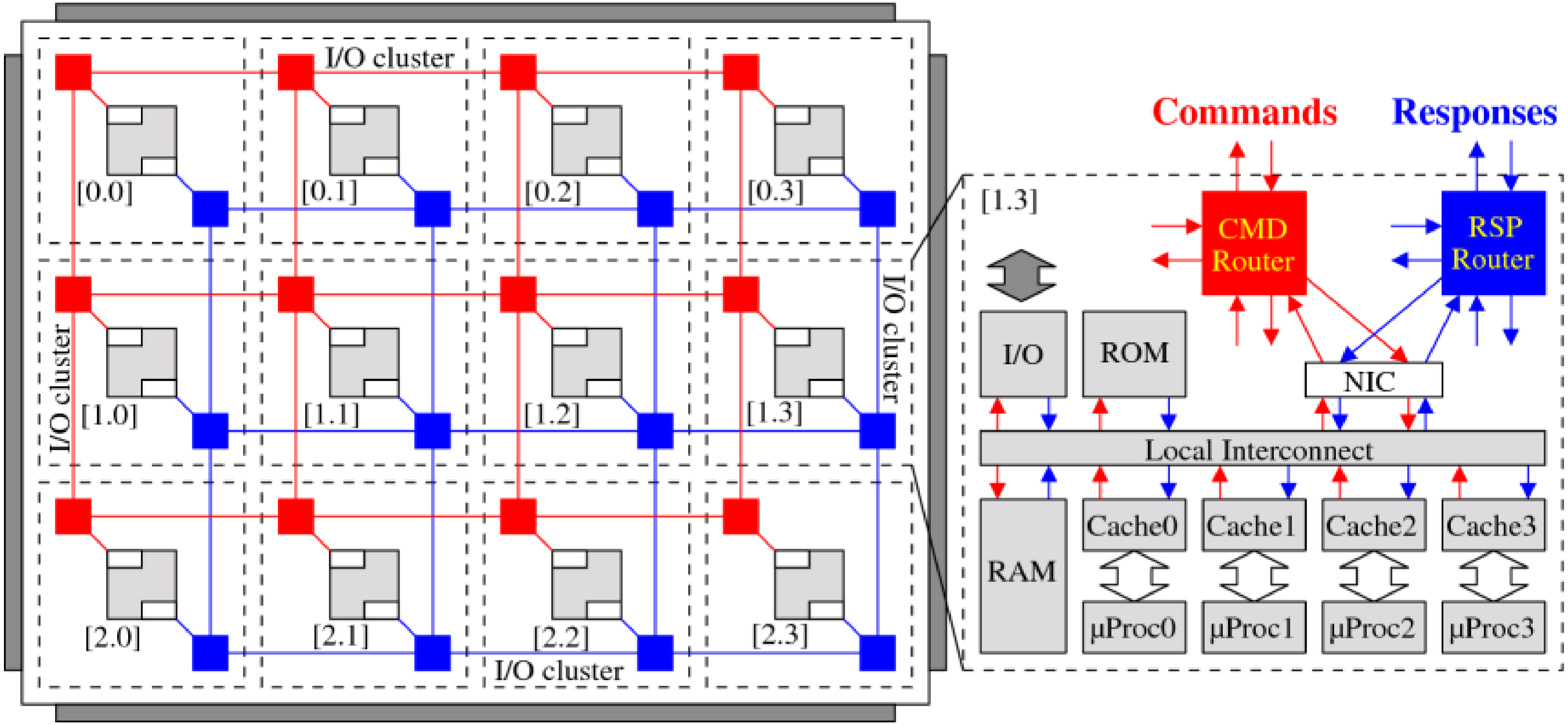
2D-Mesh NoC (DSPIN) & MP2SoC

DSPIN:

- Distributed Scalable Predictable Interconnect Network
- Designed by LIP6 laboratory and physically implemented by ST Microelectronics
- A typical 2D-Mesh NoC
- MP2SoCs architectures
- GALS (Globally Asynchronous Locally Synchronous)
  - Each subsystem is a synchronous domain
  - Susbsytem= “cluster”

MP2SoC & Cluster

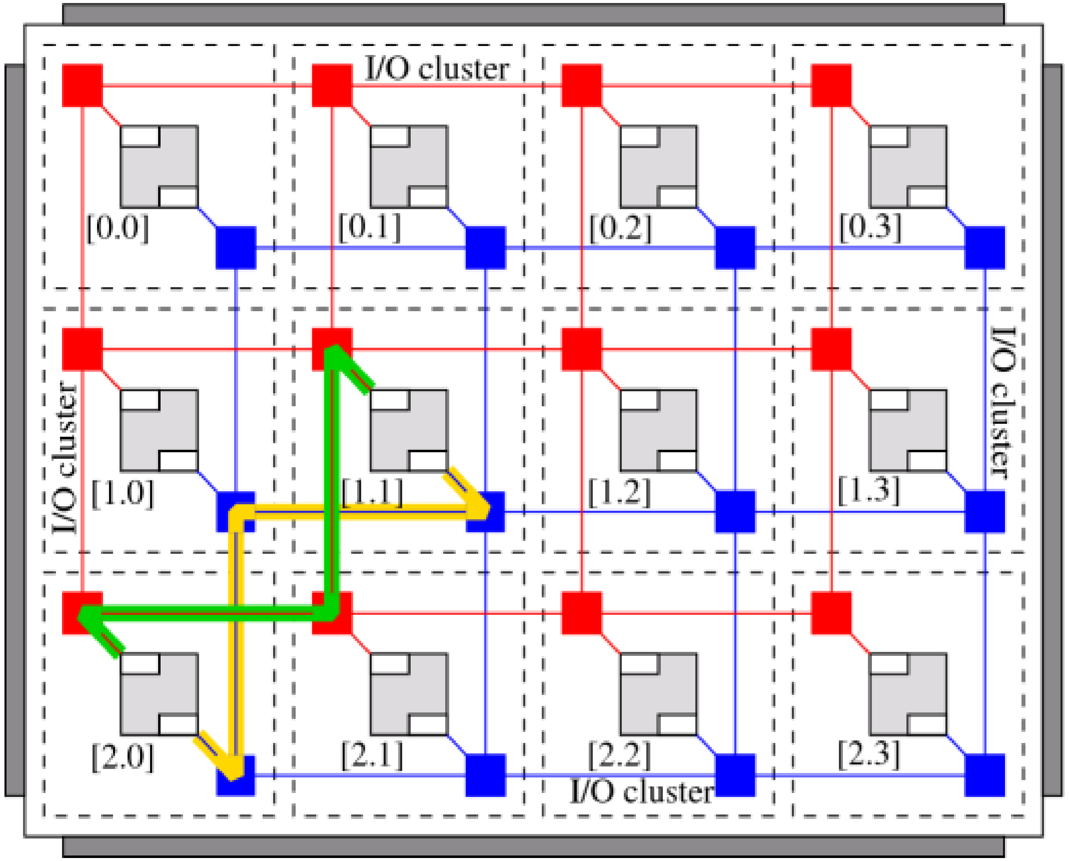
- Up to 4 Processor cores per cluster
- Network interface controller (NIC)
- Two routers (command / response)
- Embedded RAMs
- Local interconnect
- I/O Ports

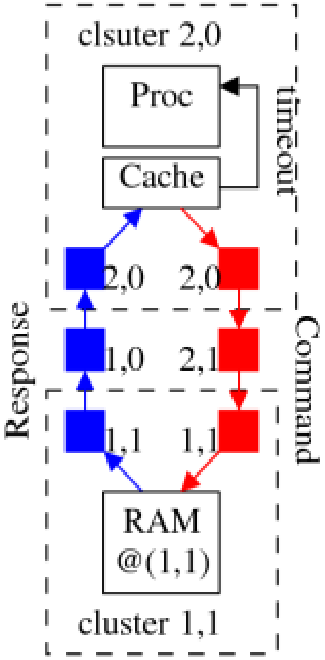


X-First Path

Routing function: X-First

- X-First path between a couple of clusters, connects a couple (processor/target)
- X-First path is round trip path: “half” for command & “half” for response.
- A timer is attached to processor to support timeout.





When a processor executes a memory load/store operation, the timer is triggered. If the transaction fails, the timeout generates an interrupt and the processor enters its exception mode.

Outline

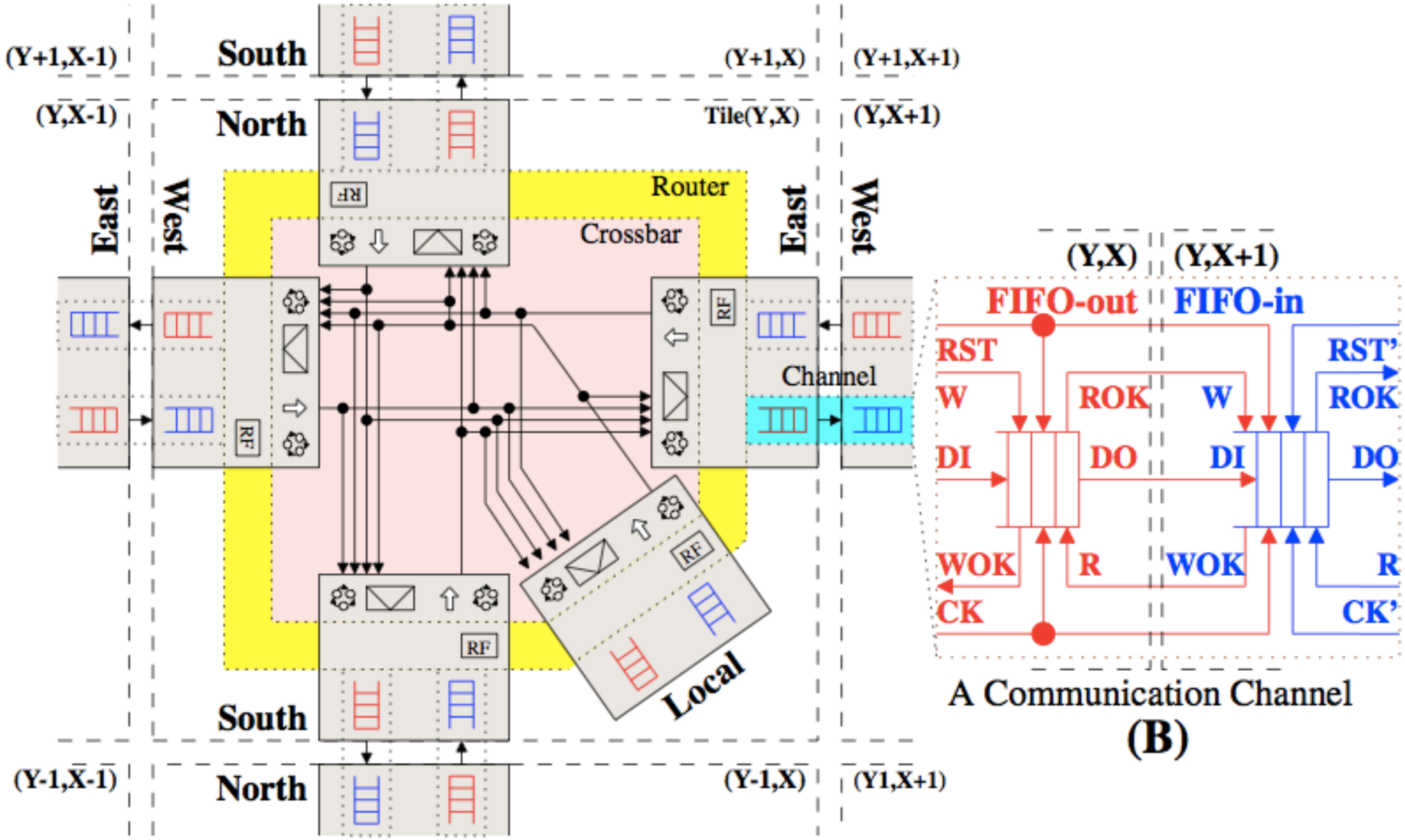
- Introduction
- 2D-Mesh NoC & Shared Memory MP2SoC
- **NoC Test Strategy**
- DCCI & Black Hole Detection
- Experimental Results
- Conclusion

NoC Test Strategy

Detection and Deactivation

- Test process executes at each system reboot or chip power-on.
- Each NoC component (router/channel) is tested in parallel & isolation.
- Faulty components are deactivated.
- Deactivated components are configured as "Black Holes"
  - Discards any incoming packet
  - Produces no outgoing packet
- Fault-free components are activated.
- X-First routing enabled on activated routers

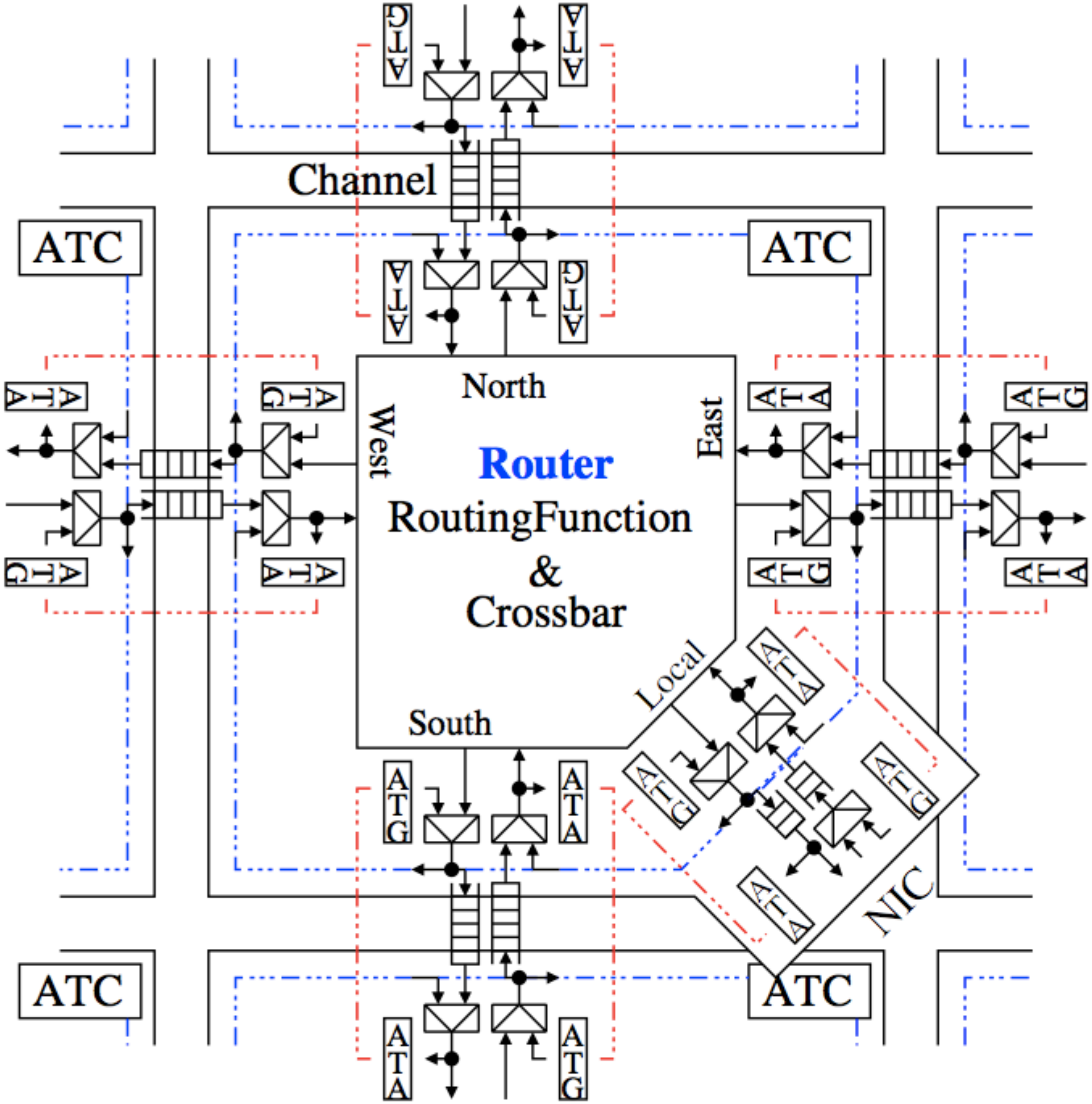
All faulty/deactivated components must be located

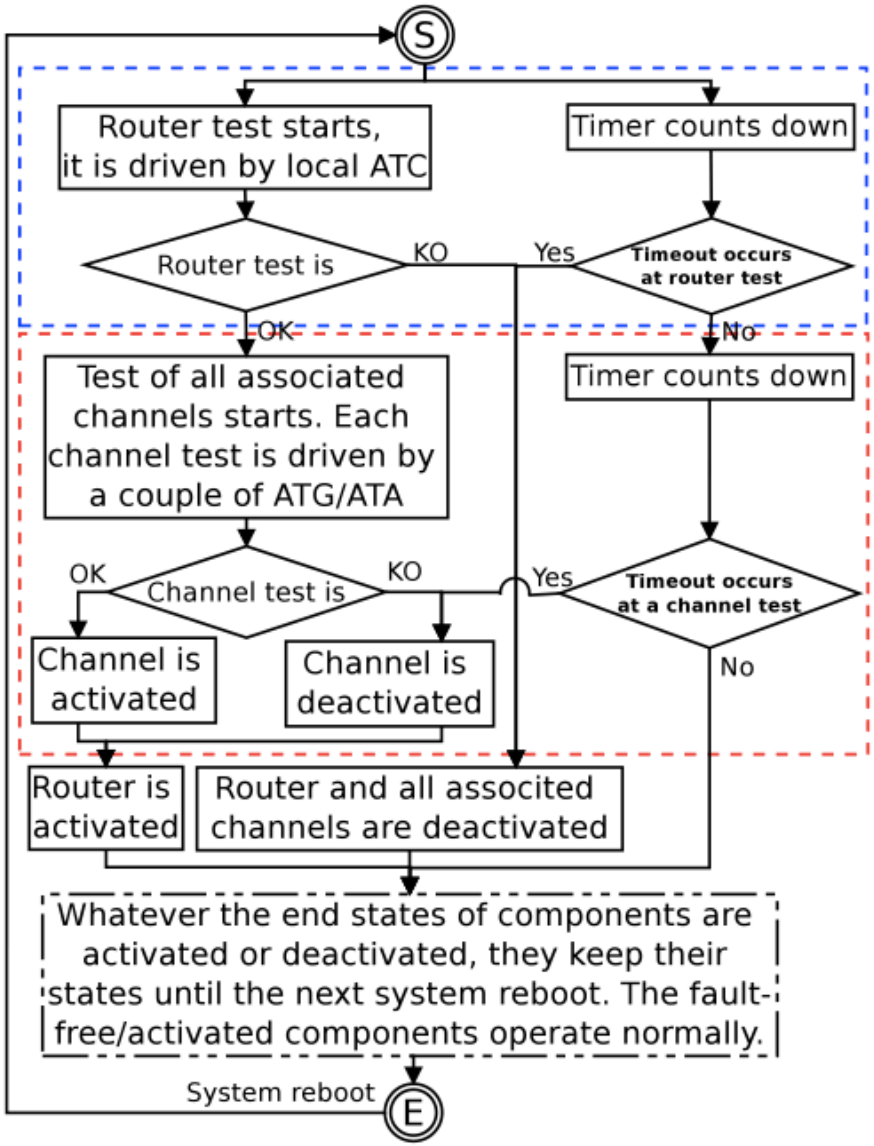


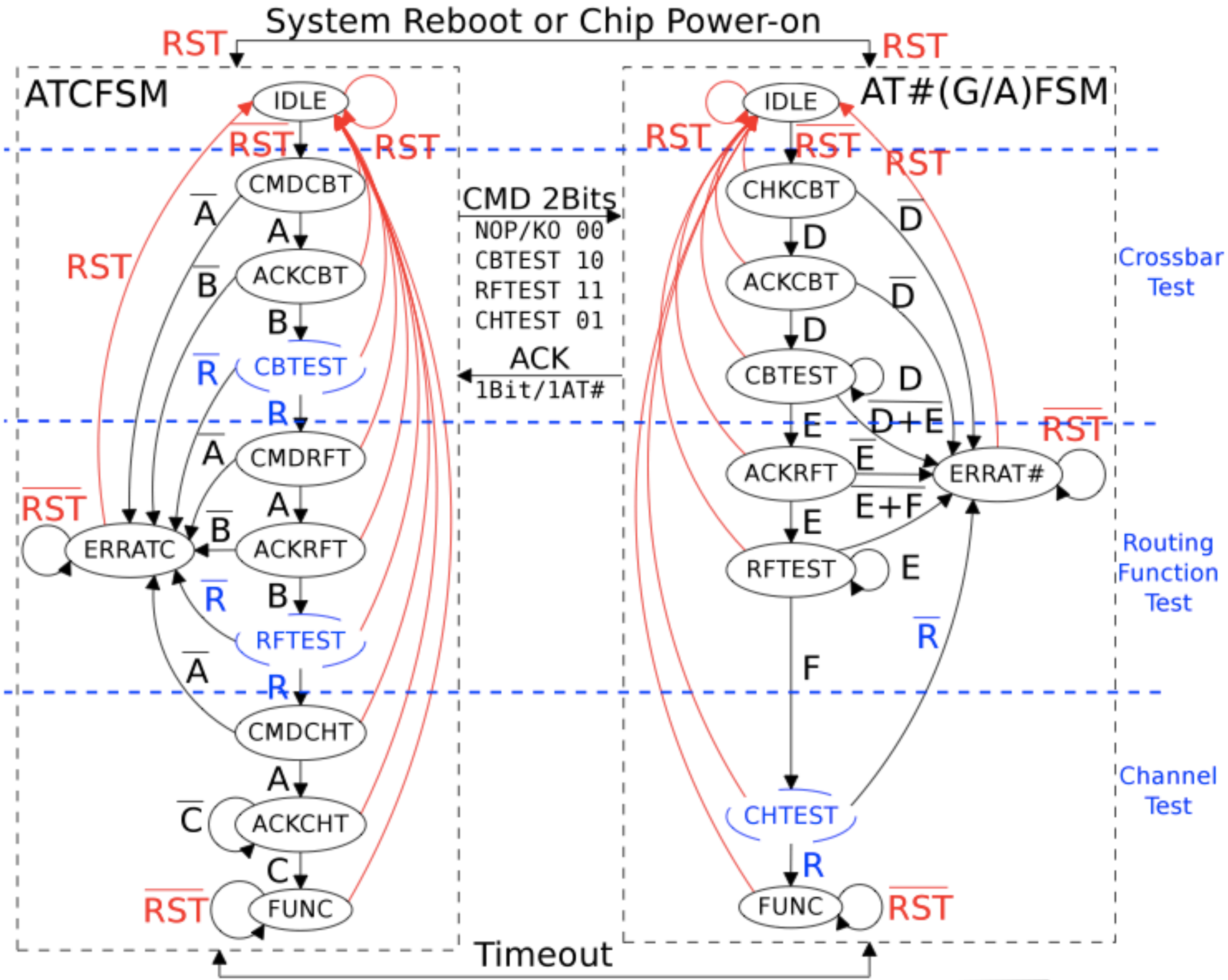
(A)

A Communication Channel

(B)


Channel
ATC
ATC
ATC
ATC
ATA
ATG
North
West
East
South
Local
NIC
Router
RoutingFunction
&
Crossbar


S
Router test starts, it is driven by local ATC
Timer counts down
Router test is
KO
Yes
Timeout occurs at router test
OK
No
Test of all associated channels starts. Each channel test is driven by a couple of ATG/ATA
Timer counts down
OK
Channel test is
KO
Yes
Timeout occurs at a channel test
Channel is activated
Channel is deactivated
No
Router is activated
Router and all associocted channels are deactivated
Whatever the end states of components are activated or deactivated, they keep their states until the next system reboot. The fault-free/activated components operate normally.
System reboot
E



A : (ACK="0...0") * $\overline{\text{Timeout}}$

B : (ACK="1...1") * $\overline{\text{Timeout}}$

C : ((ACK="1...1") + $\overline{\text{Timeout}}$)

R : TESTOK * $\overline{\text{Timeout}}$

D : (CMD="10") * $\overline{\text{Timeout}}$

E : (CMD="11") * $\overline{\text{Timeout}}$

F : (CMD="01") * $\overline{\text{Timeout}}$

R : TESTOK * $\overline{\text{Timeout}}$

Outline

- Introduction
- 2D-Mesh NoC & Shared Memory MP2SoC
- NoC Test Strategy
- **DCCI & Black Hole Detection**
- Experimental Results
- Conclusion

Configuration Infrastructure

Objectives

- Determine a global configuration master
- Create a global configuration bus
- Identify the faulty/de-activated components of NoC
- Reconfigure the NoC routing function
- DCCI (Distributed Cooperative Configuration Infrastructure) is proposed and used in our work

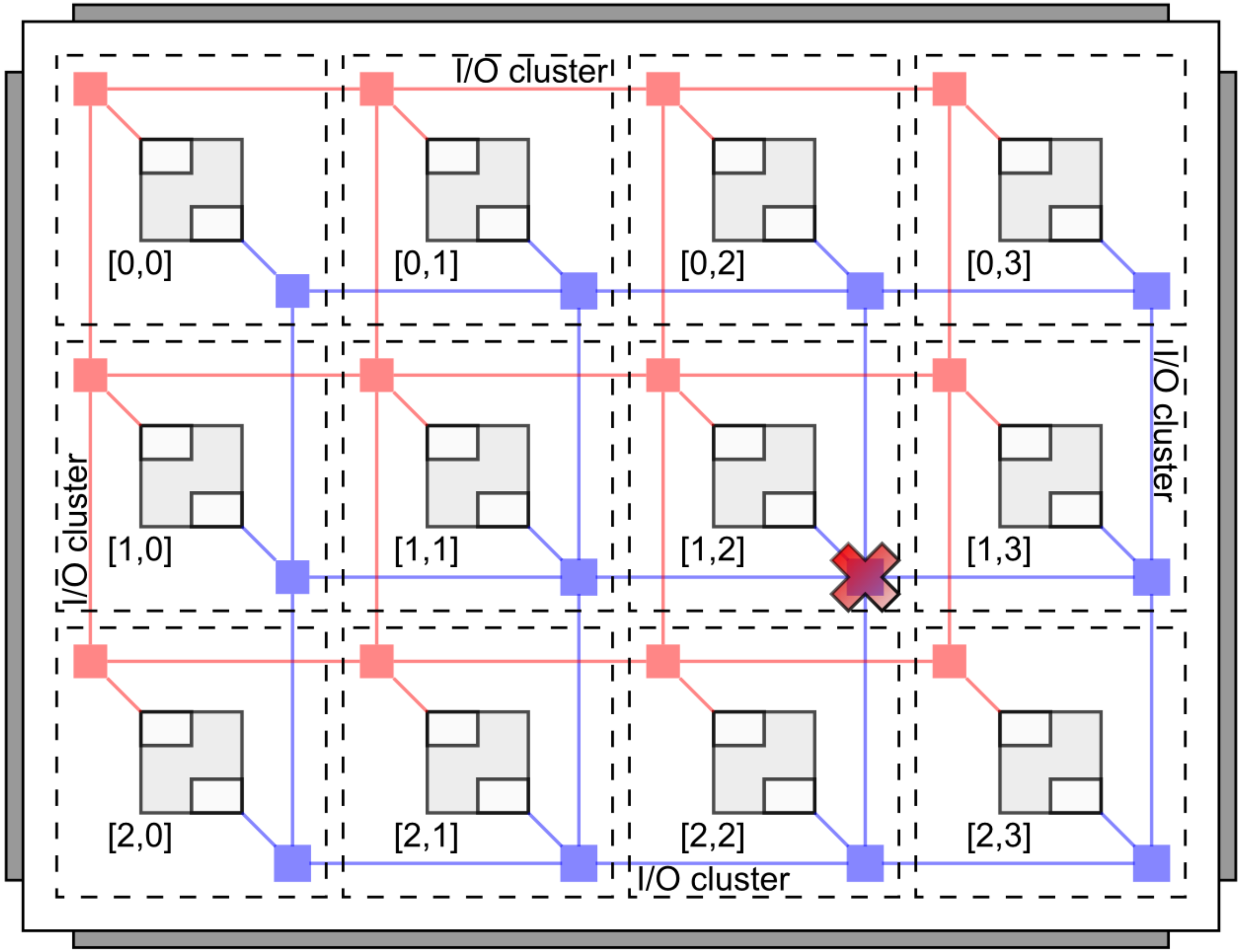
DCCI

- Every cluster has his own embedded BIOS, named CF (Configuration Firmware)
- After NoC test, each cluster runs CF code to do software-based self-test. Each faulty cluster is deactivated. Each fault-free cluster tries to communicate with its neighbor clusters
- Finally, a software-based communication tree, spanning and covering all fault-free clusters, is created
- The tree root is the configuration master, the tree itself is the configuration bus
- The tree root can load “black hole” detection software from external memory
- The tree root can send command, test, configuration orders to each node

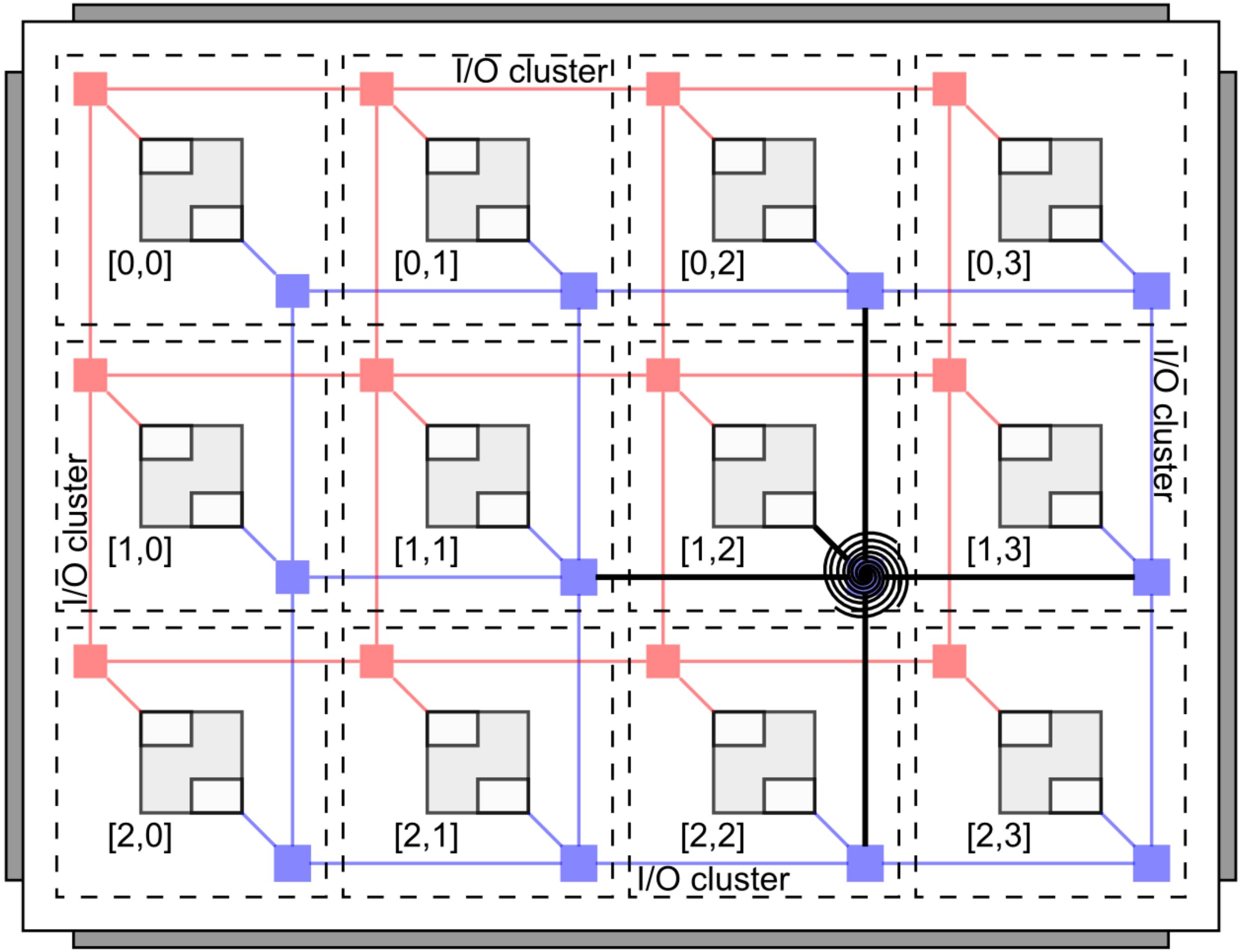
Black Hole Detection

- The Black Hole detection is a distributed software application
- DCCI Tree root loads the software from the external storage device
- Tree root distributes the software to each node
- Each node tests local X-First paths and marks fault-free NoC components
- Tree root gathers local results to achieve the global result

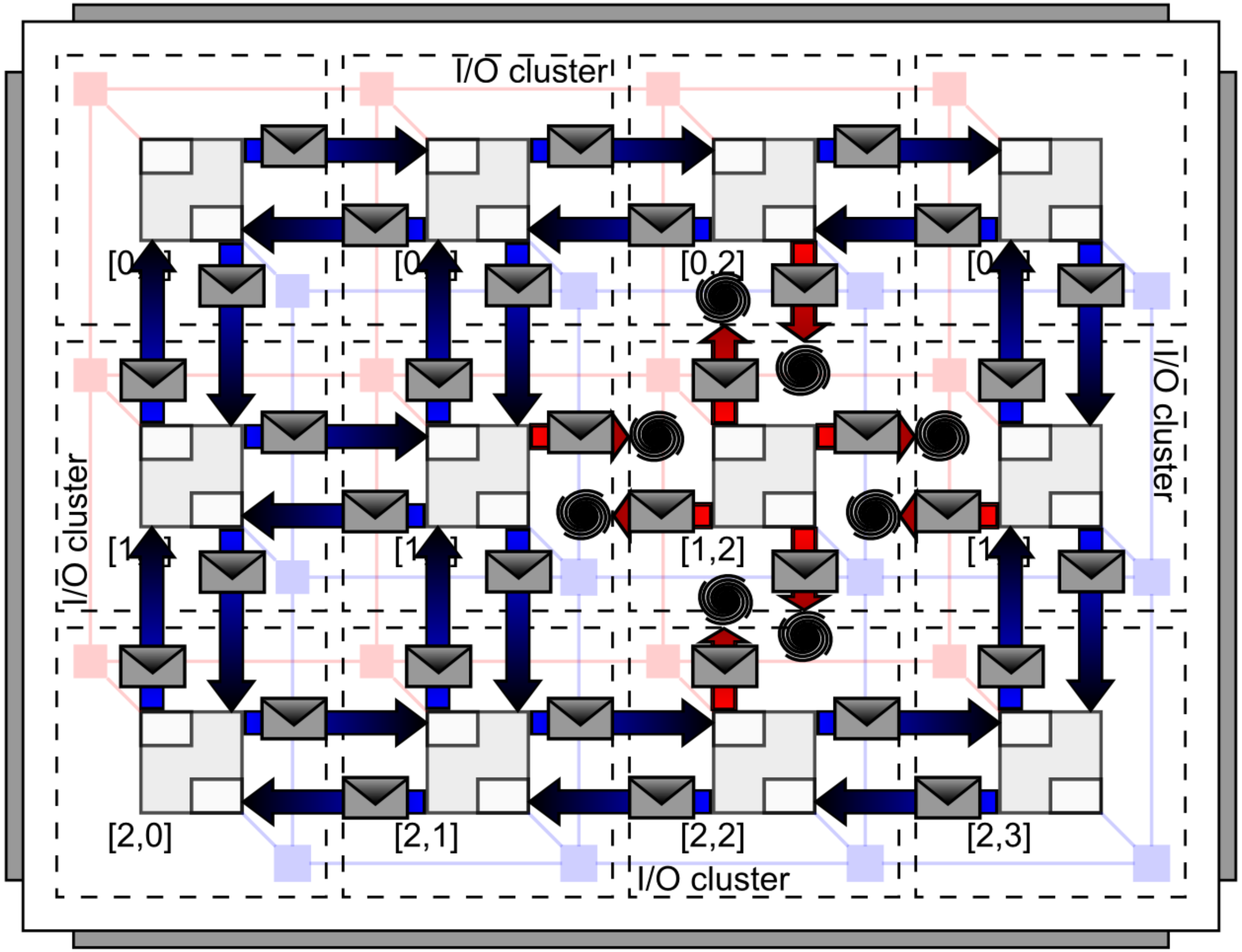
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

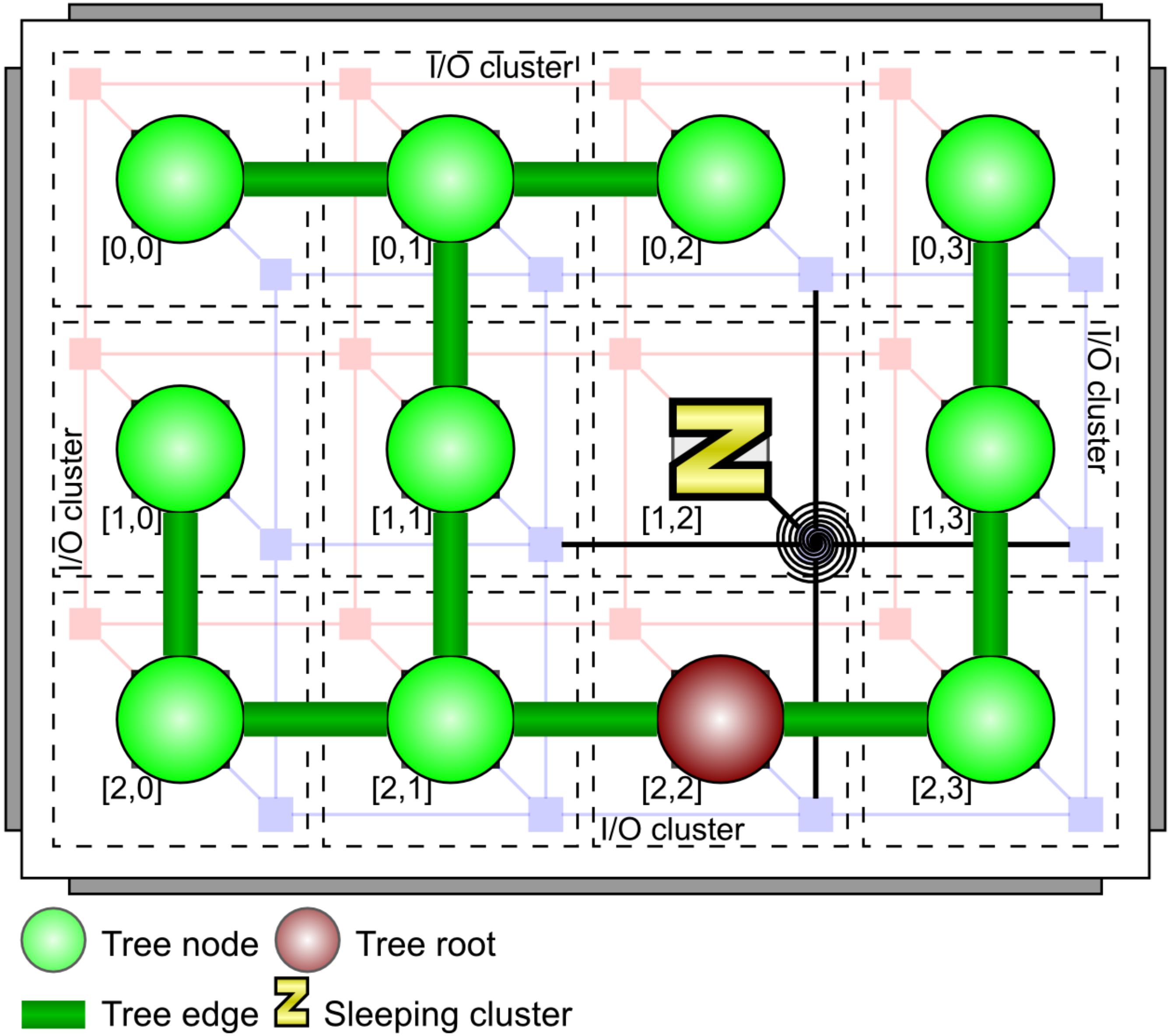
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

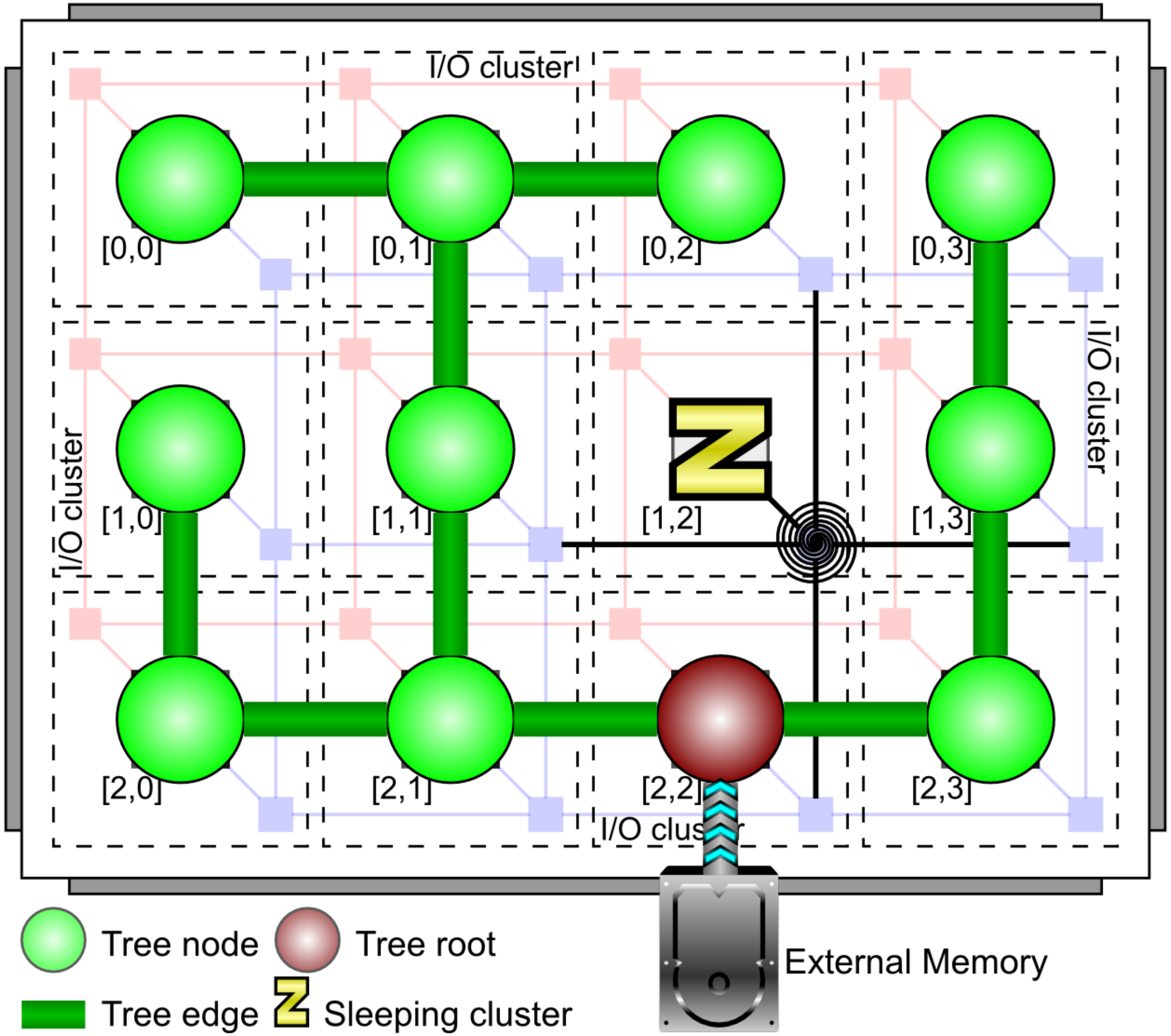
Example



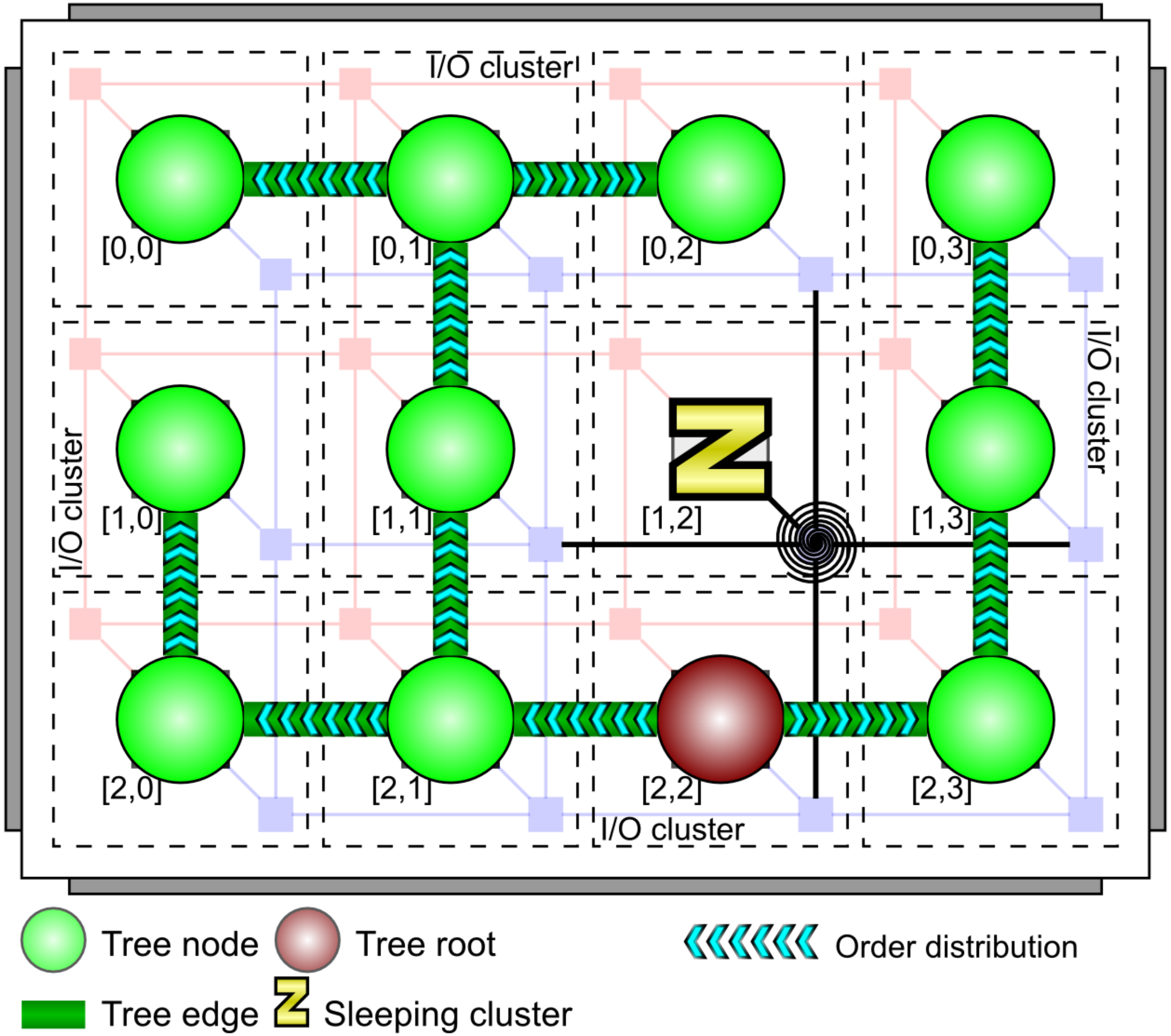
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Tree node
Tree root
Tree edge
Sleeping cluster

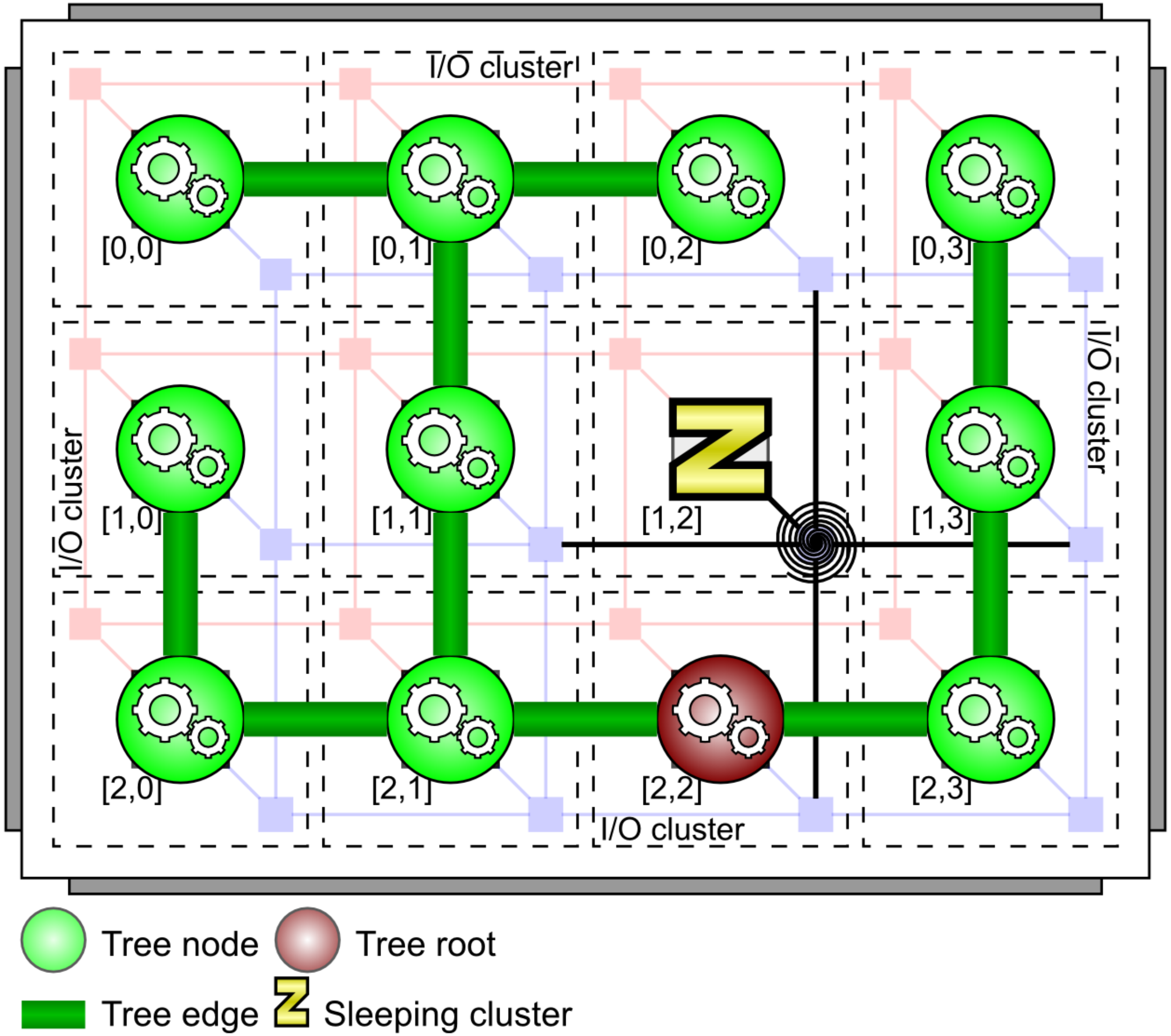
Example


I/O cluster
I/O cluster
I/O cluster
I/O clus
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
Tree node
Tree root
Tree edge
Sleeping cluster
External Memory

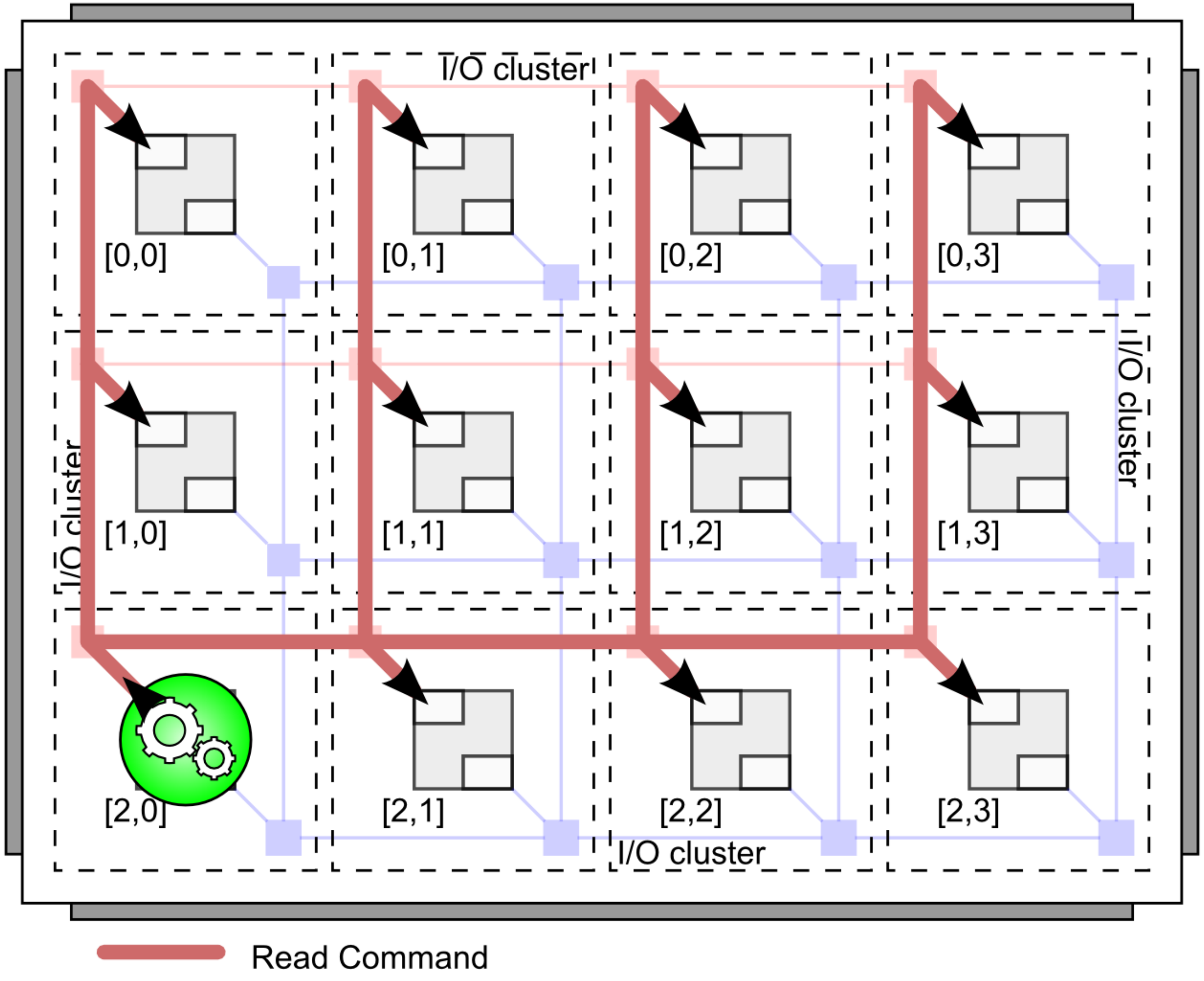
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Tree node
Tree root
Order distribution
Tree edge
Sleeping cluster

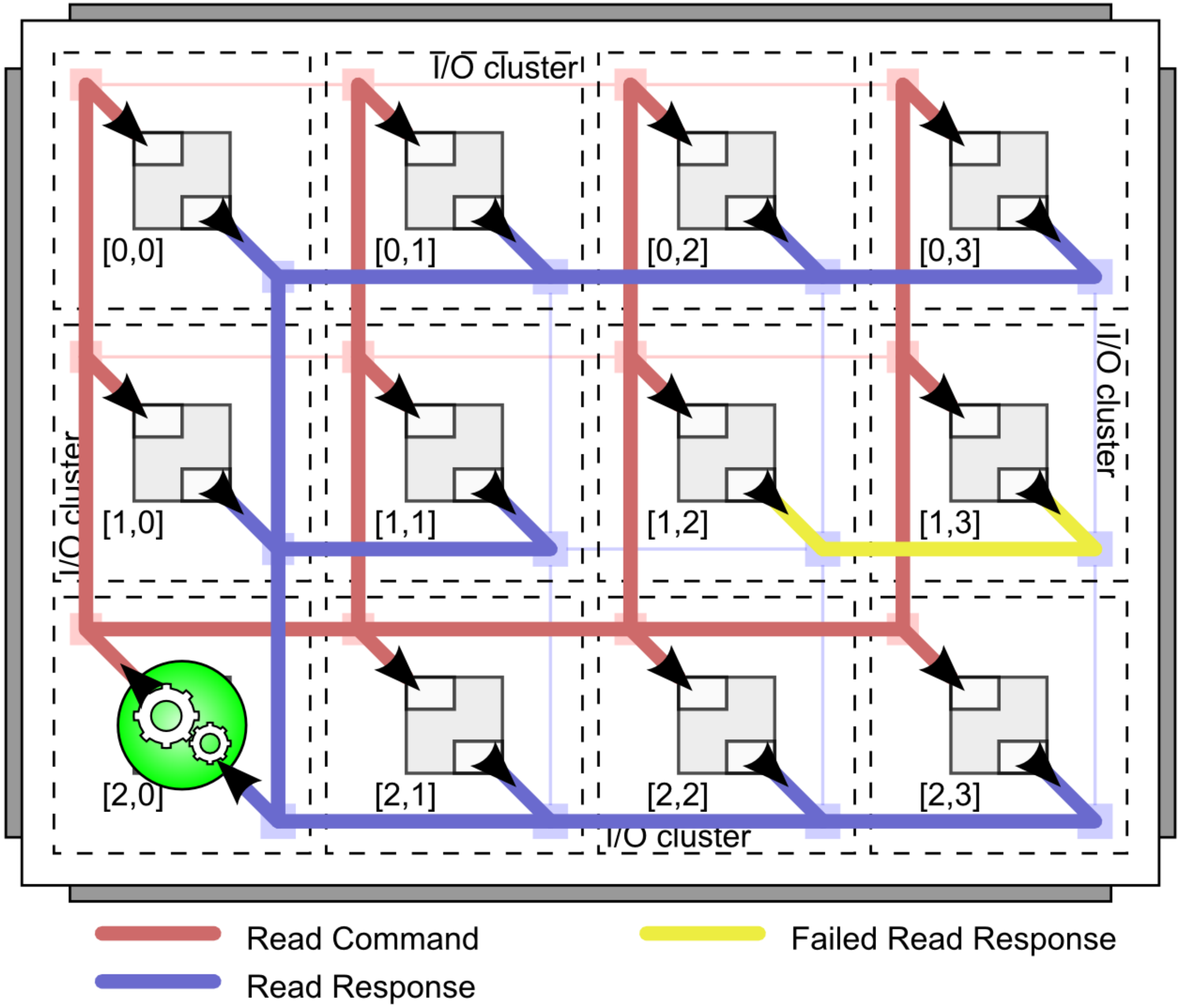
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Tree node
Tree root
Tree edge
Sleeping cluster

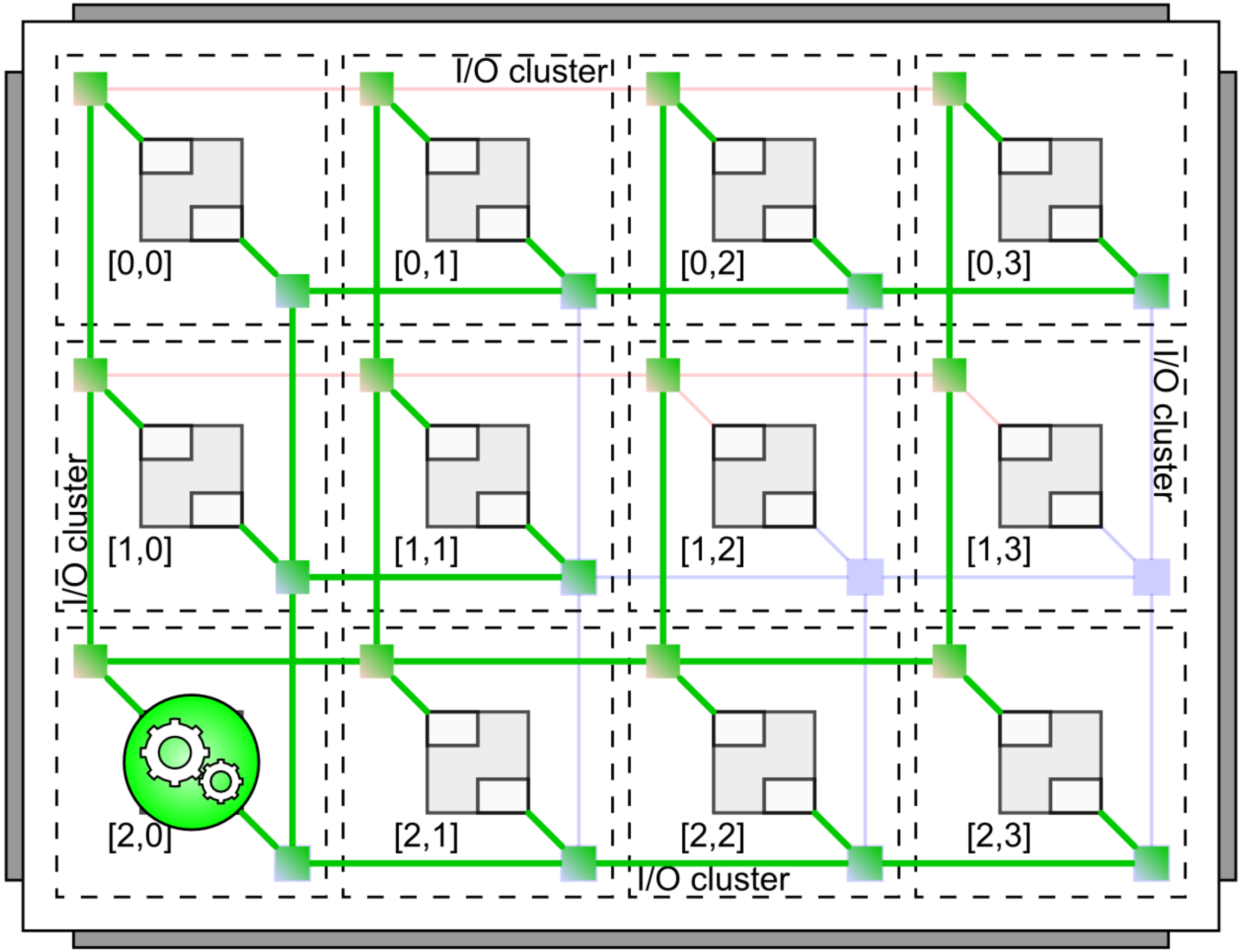
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Read Command

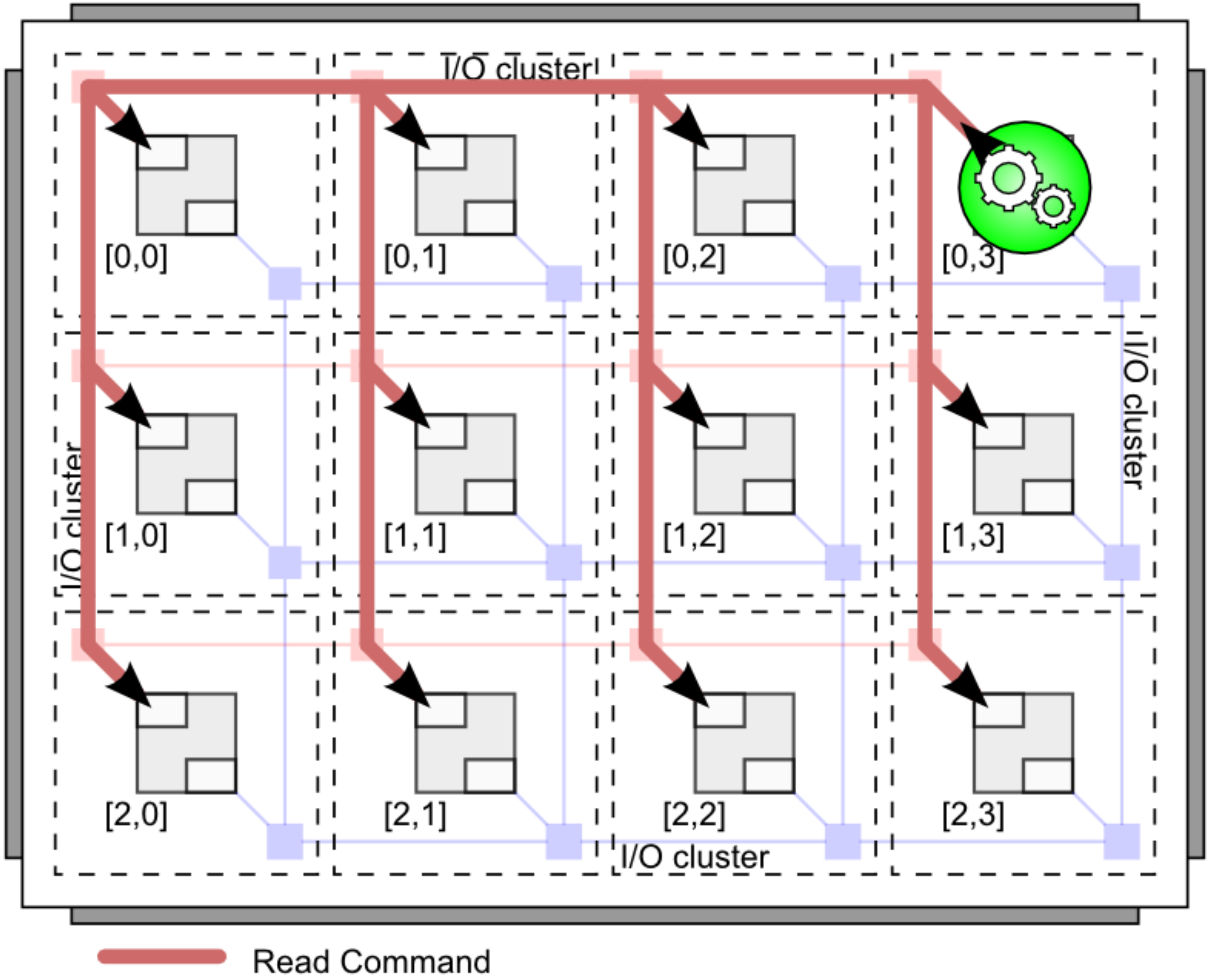
Example


I/O cluster
I/O cluster
I/O cluster
I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
Read Command
Read Response
Failed Read Response

Example

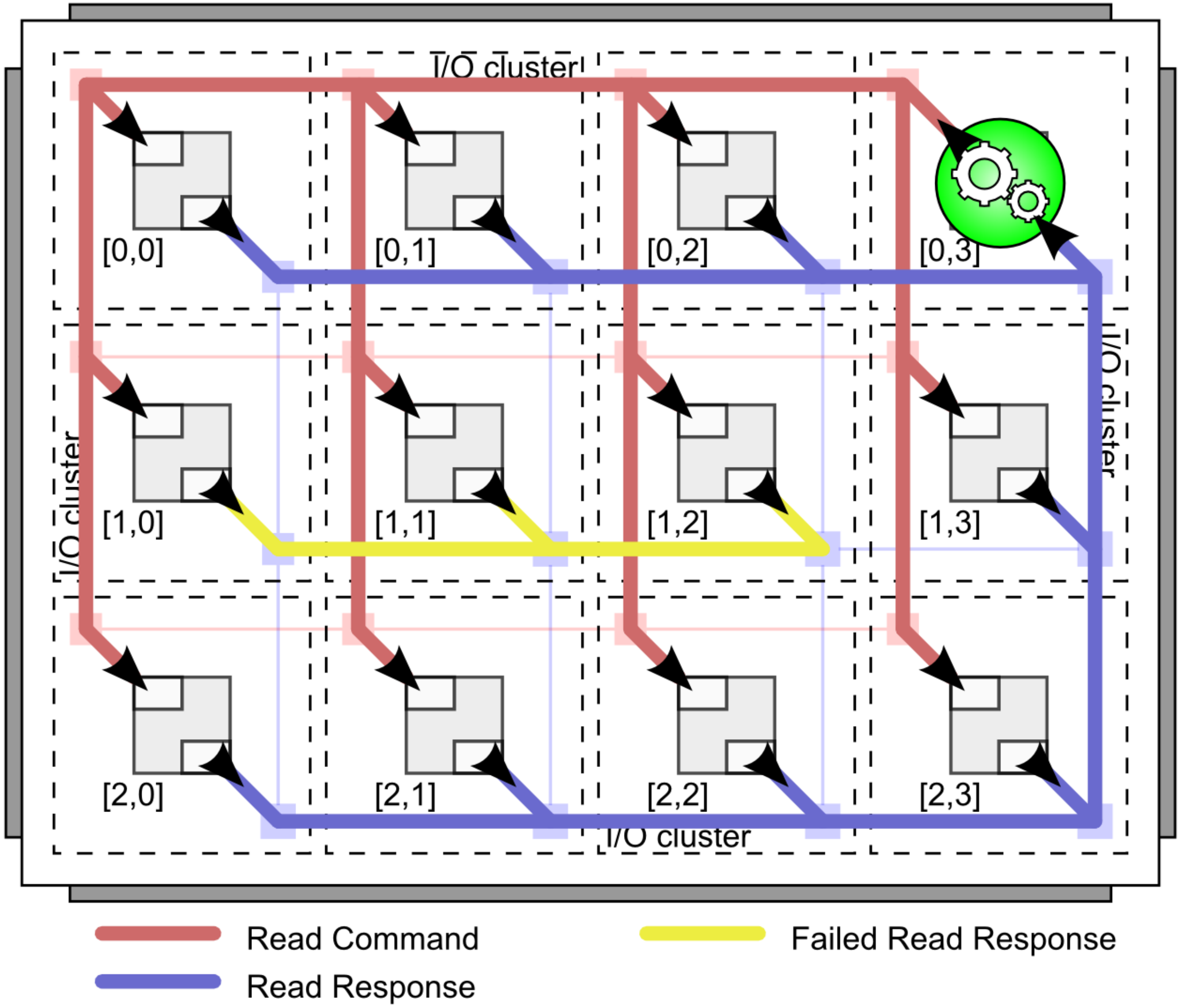

I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

Example

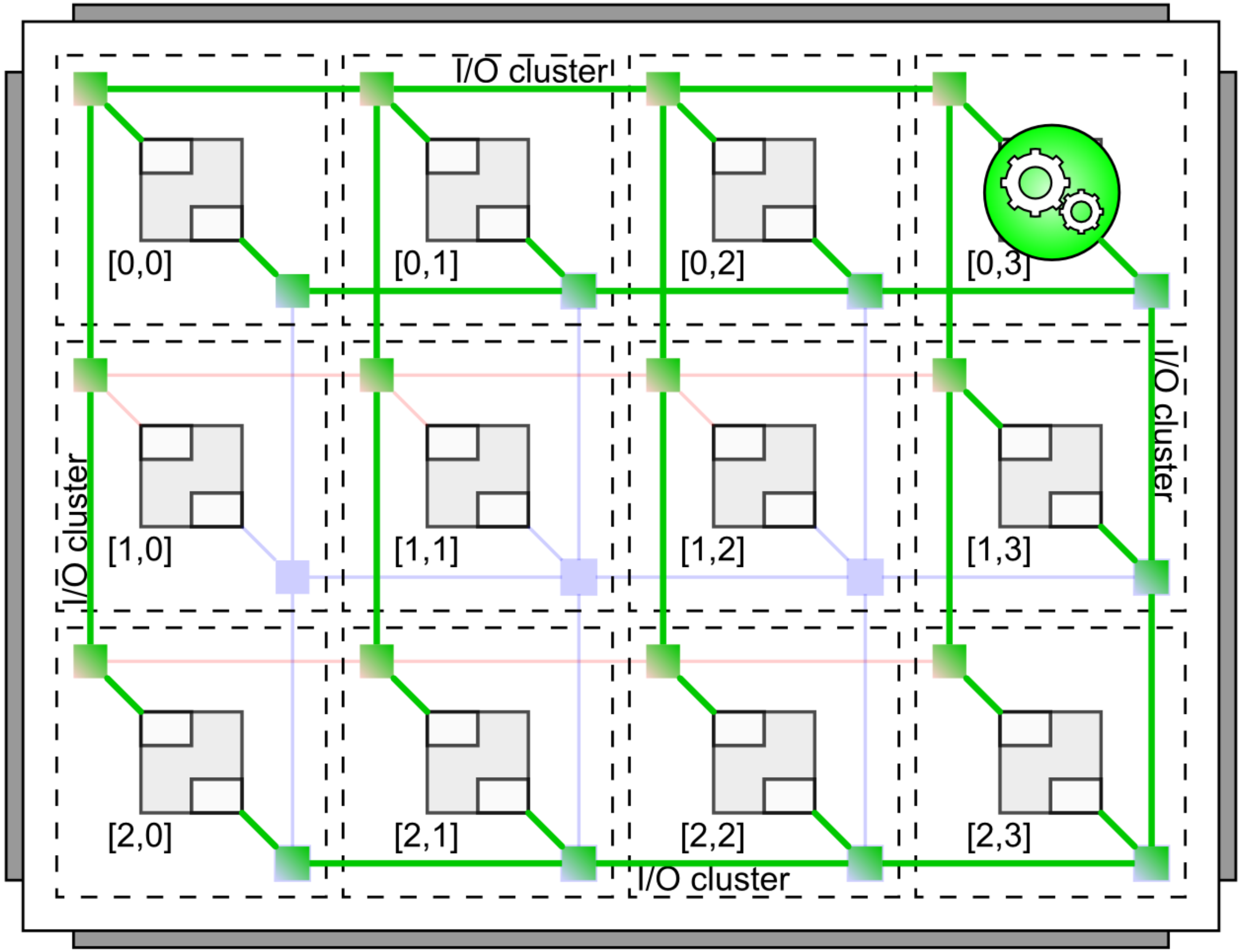


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
[1,0]
[1,1]
[1,2]
[1,3]
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
I/O cluster
I/O cluster
Read Command

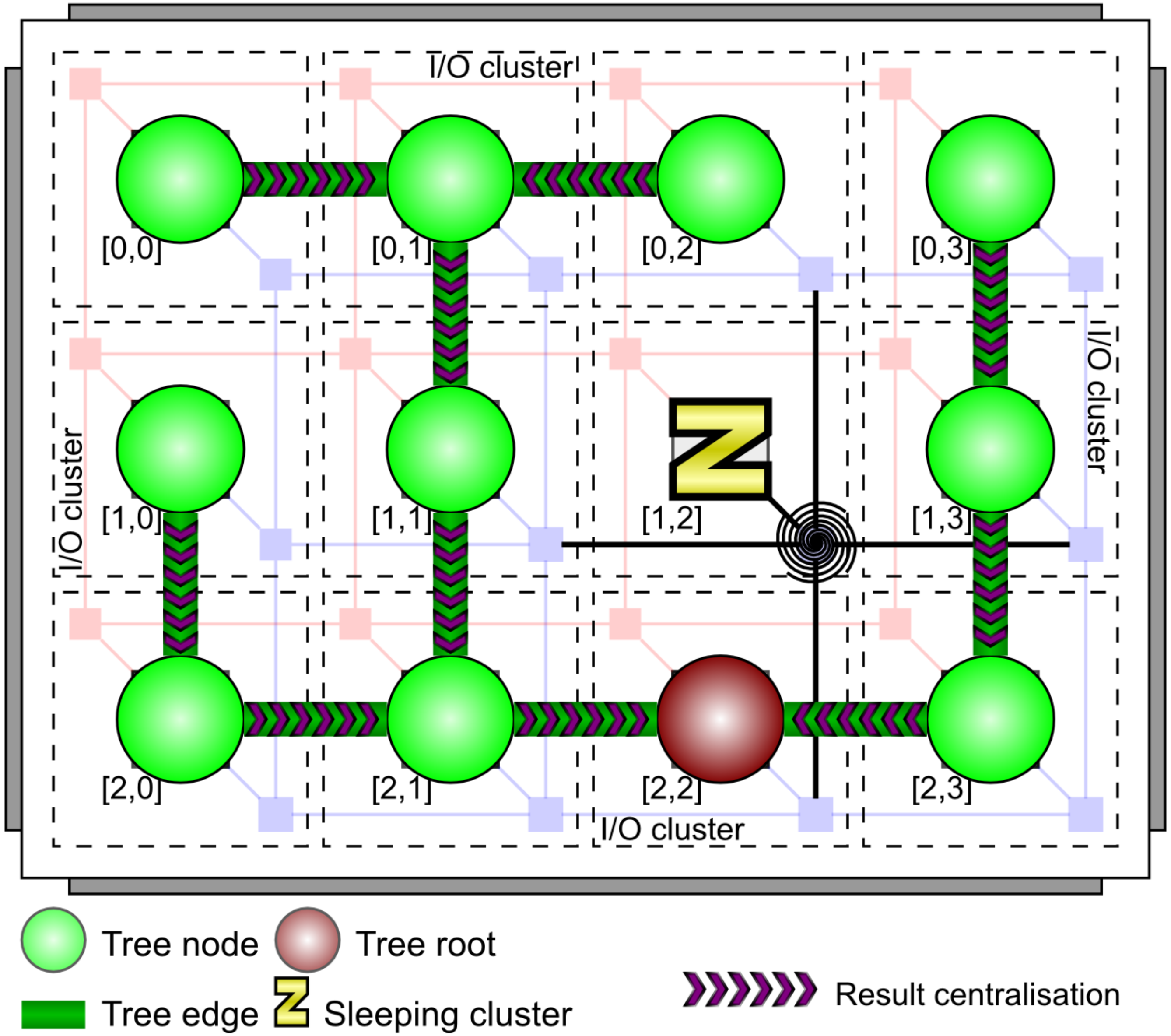
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Read Command
Read Response
Failed Read Response

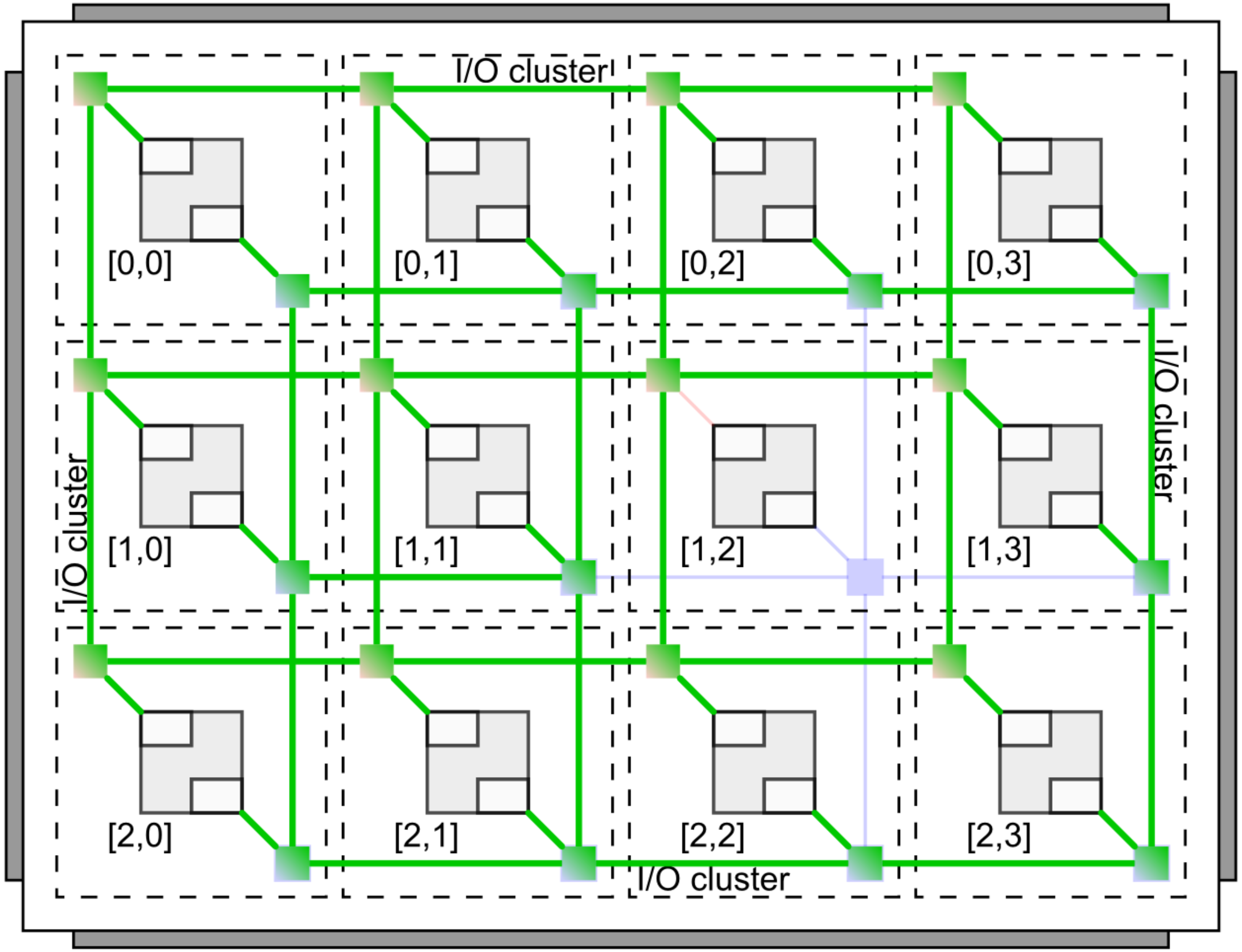
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

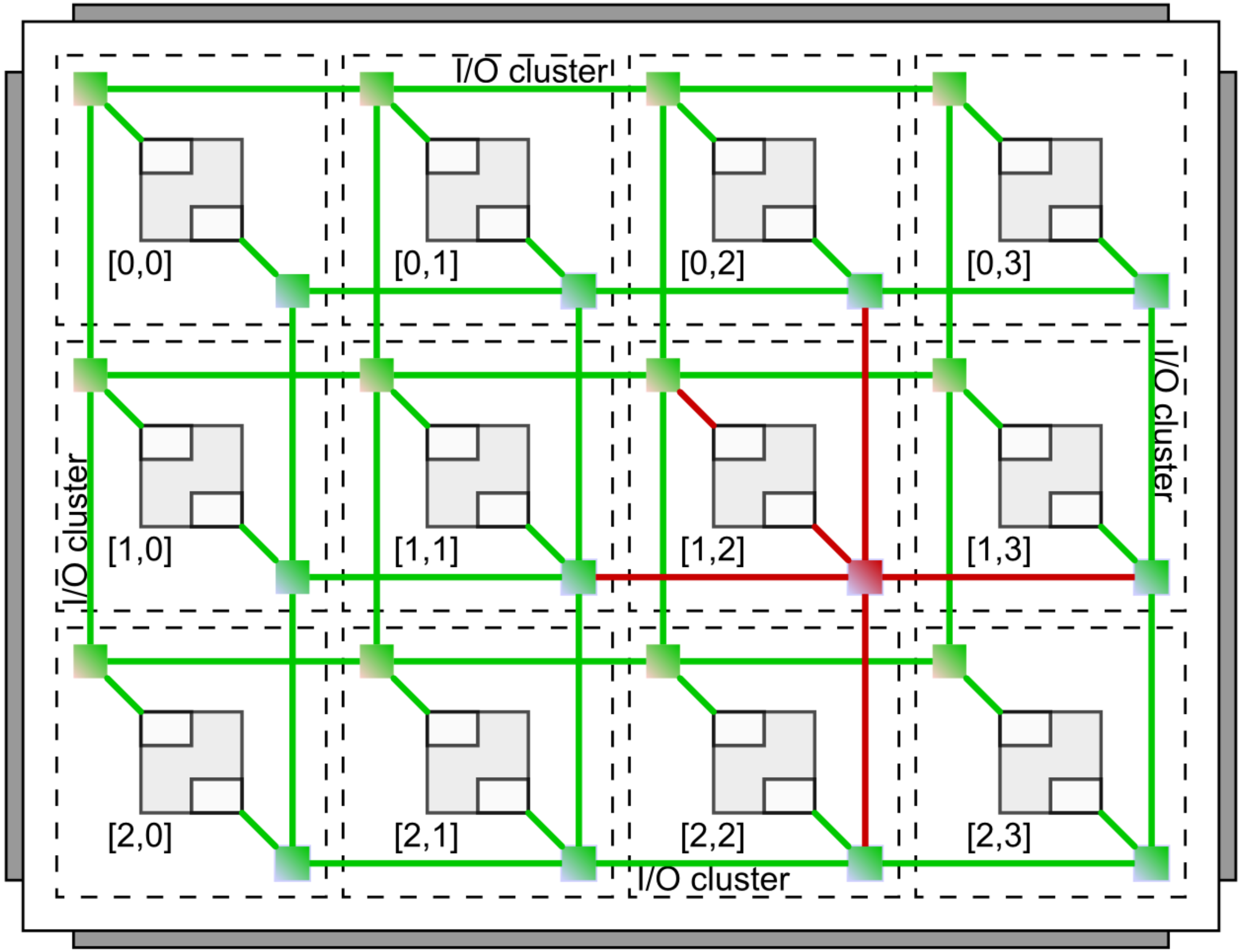
Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster
Tree node
Tree root
Tree edge
Sleeping cluster
Result centralisation

Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

Example


I/O cluster
[0,0]
[0,1]
[0,2]
[0,3]
I/O cluster
[1,0]
[1,1]
[1,2]
[1,3]
I/O cluster
[2,0]
[2,1]
[2,2]
[2,3]
I/O cluster

Outline

- Introduction
- 2D-Mesh NoC & Shared Memory MP2SoC
- NoC Test Strategy
- DCCI & Black Hole Detection
- **Experimental Results**
- Conclusion

Coverage of Black Hole Detection

- C program simulation
- MP2SoC with 4×4 clusters
- Simulated experiments
  - One single fault injection
    - one faulty channel
    - one faulty router
  - Multi faults injection
    - 1 faulty channel + 1 faulty routers
    - 2 faulty channels
    - 2 faulty routers
    - 1 faulty channel + 2 faulty routers
    - 2 faulty channel + 1 faulty routers
    - 2 faulty channel + 2 faulty routers

Black Hole detection coverage is 100%

Execution Time

- 4×4 MP2SoC architecture containing 16 processors, modeled with the cycle-accurate **[SoCLib]** virtual prototyping platform
- One single fault
- The total time is 7.1×10^6 cycles (without hardware test process):
  - Time for (DCCI) tree construction: 1:9×10^6 cycles
  - Time for for test task distribution: 1:2×10^6 cycles
  - Time for test execution: 3.5×10^6 cycles
  - Time for test result centralization: 0.5×10^6 cycles

0.014 second at 500Mhz

Application Code Size

Application Code Size (for a MIPS32 processor):

- DCCI : 5 Kbytes
- Black Hole Detection : 2.5 Kbytes

Outline

- Introduction
- 2D-Mesh NoC & Shared Memory MP2SoC
- NoC Test Strategy
- DCCI & Black Hole Detection
- Experimental Results
- **Conclusion**

Conclusion

- DCCI dynamically builds a software based communication tree, covering all the nodes that have successfully passed the local BIST.
- DCCI communication infrastructure is a distributed software mechanism. The tree root is the configuration master.
- Relying on the DCCI tree, the configuration master can locate 100% of the faulty components (a point-to-point communication channel, or a complete router), converted into black holes.
- The same DCCI communication tree can be used to distribute the resulting modified routing functions to the fault-free routers.
- The method proposed can be used in any shared memory multi-core architecture with a 2D-Mesh NoC.

And don’t forget the interconnect…

The case for programmable on-chip interconnect

François Pêcheux – UPMC/Lip6

Dumitru Potop-Butucaru – INRIA

Conclusion

- The future of computing is parallel
  - Both embedded and high-performance
  - Computing elements (CPUs) and interconnect are equally important
  - But:
    - CPUs can be programmed (in C)
    - Interconnect only provide limited configurability (many approaches)

- Interconnect should allow better and more standard « programmability »
  - Especially in Systems-on-Chips

- Application mapping (compilers/OS) should take into acount both CPUs and interconnect (global optimization)

Outline

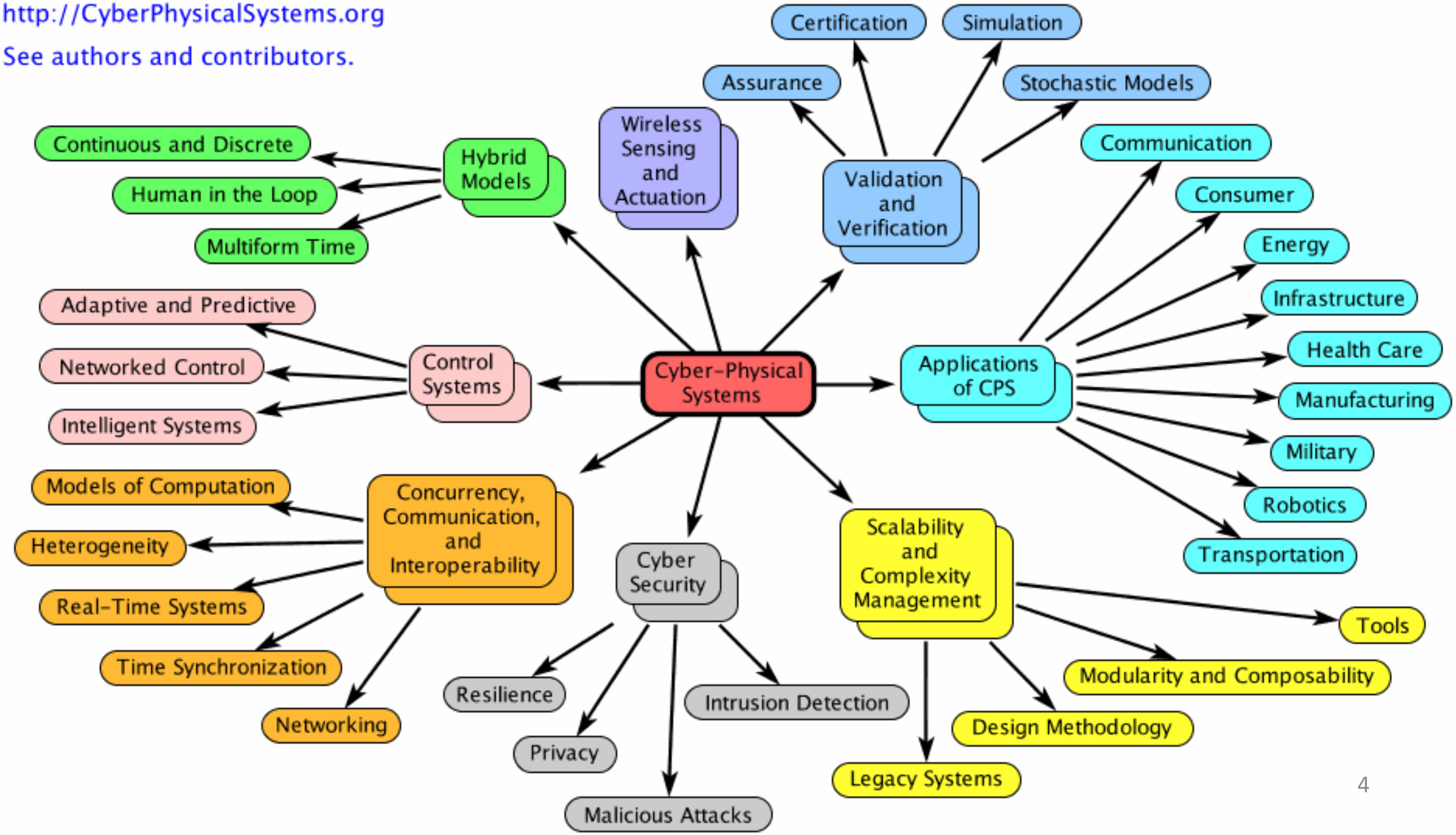
- Multiprocessor embedded systems
  - The mapping problem
- Playing with applications, architectures, and mapping
- Conclusion: Field-Programmable Tile Arrays

Embedded systems

Cyber-Physical Systems – a Concept Map

http://CyberPhysicalSystems.org

See authors and contributors.



Cyber-Physical Systems

Hybrid Models
- Continuous and Discrete
- Human in the Loop
- Multiform Time

Wireless Sensing and Actuation

Validation and Verification
- Assurance
- Certification
- Simulation
- Stochastic Models

Applications of CPS
- Communication
- Consumer
- Energy
- Infrastructure
- Health Care
- Manufacturing
- Military
- Robotics
- Transportation

Control Systems
- Adaptive and Predictive
- Networked Control
- Intelligent Systems

Concurrency, Communication, and Interoperability
- Models of Computation
- Heterogeneity
- Real-Time Systems
- Time Synchronization
- Networking

Cyber Security
- Resilience
- Privacy
- Malicious Attacks
- Intrusion Detection

Scalability and Complexity Management
- Tools
- Modularity and Composability
- Design Methodology
- Legacy Systems

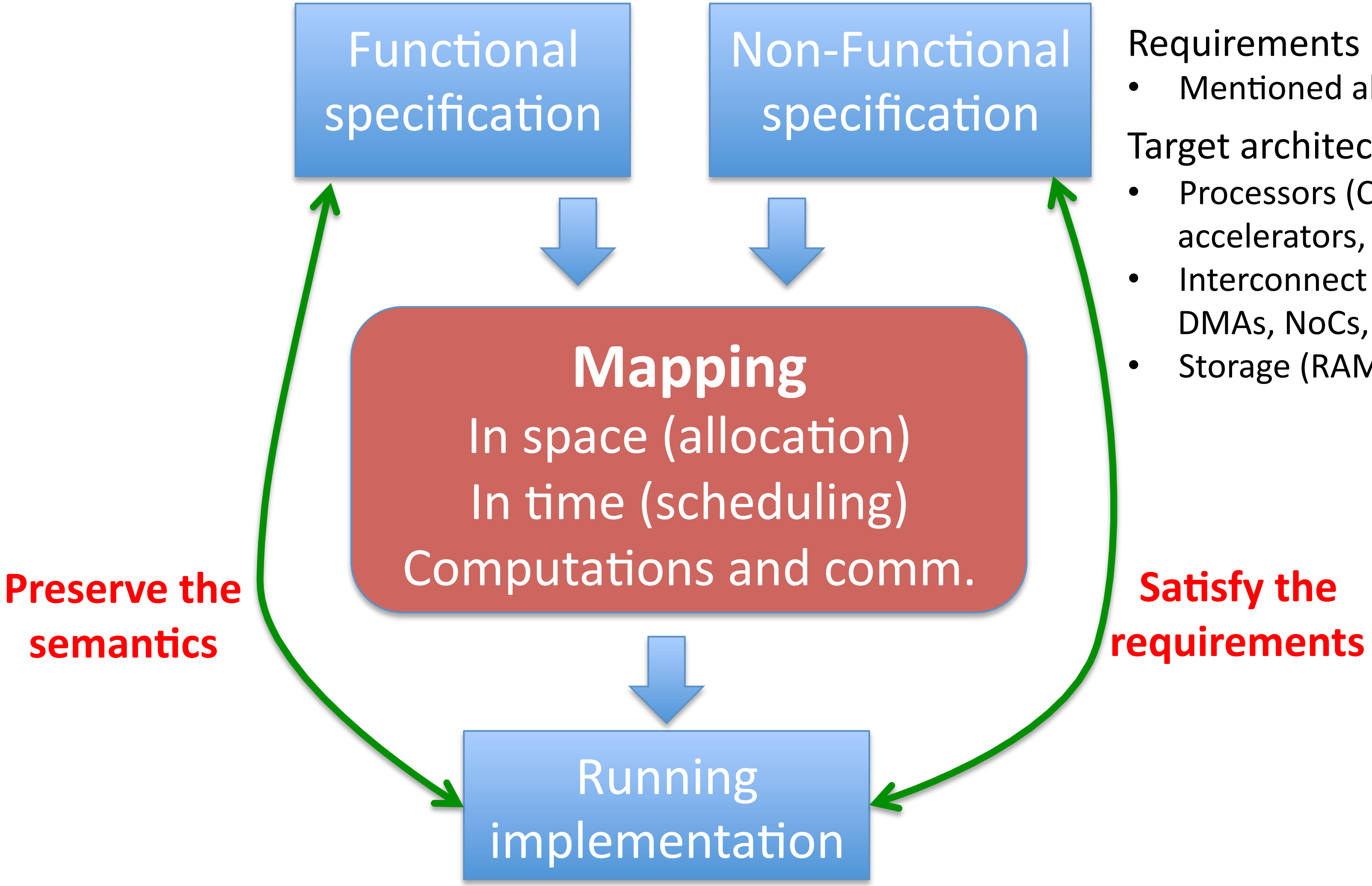
Embedded systems

- Common features:
  - Reactive systems: Execution is *a priori* infinite
  - Non-functional requirements.
  - Specification and implementation are complex by both engineering and theoretical criteria
    - Specification: Multiple languages/formalisms, both general-purpose (C, Ada,UML) or domain-specific languages (DSLs like Simulink, SCADE, AUTOSAR, AADL, SysML, etc.). Heavy use of program analysis techniques (verification, simulation, platform exploration, etc.).
    - Implementation: Custom hardware (micro-controllers with small speeds/ RAMs, FPGAs, specific buses, etc.), non-functional requirements.
  - Safety-critical, errors are expensive (in either lives or money)
    - Functional determinism is often desired
- Consequences: Common needs in the development process

Embedded systems

- Complex system-level non- functional requirements:
  - Real-time
    - Efficiency
    - Predictability
  - Low-power
    - Green computing
  - Safety
    - Fault tolerance
  - Security
    - Application isolation
  - Cost (money/time/…)
    - platform/development/exploitation
  - Flexibility
    - System evolution
  - Size
  - Thermal
  - …

Embedded systems implementation



Requirements

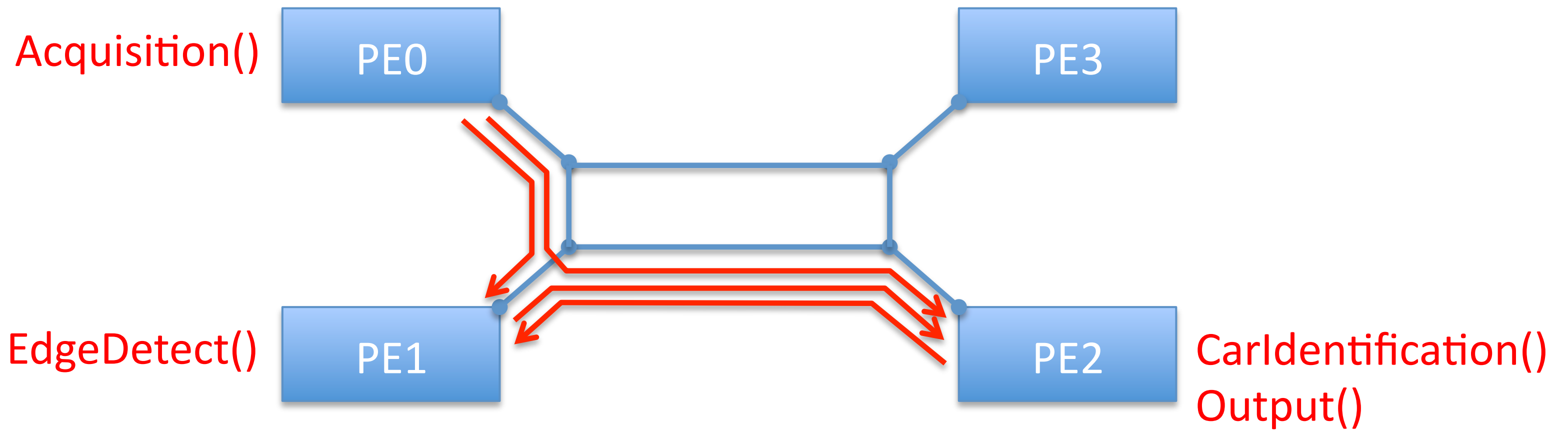
- Mentioned above

Target architecture (

- Processors (CPUs, accelerators, etc.)
- Interconnect (buses, DMAs, NoCs, etc.)
- Storage (RAM)

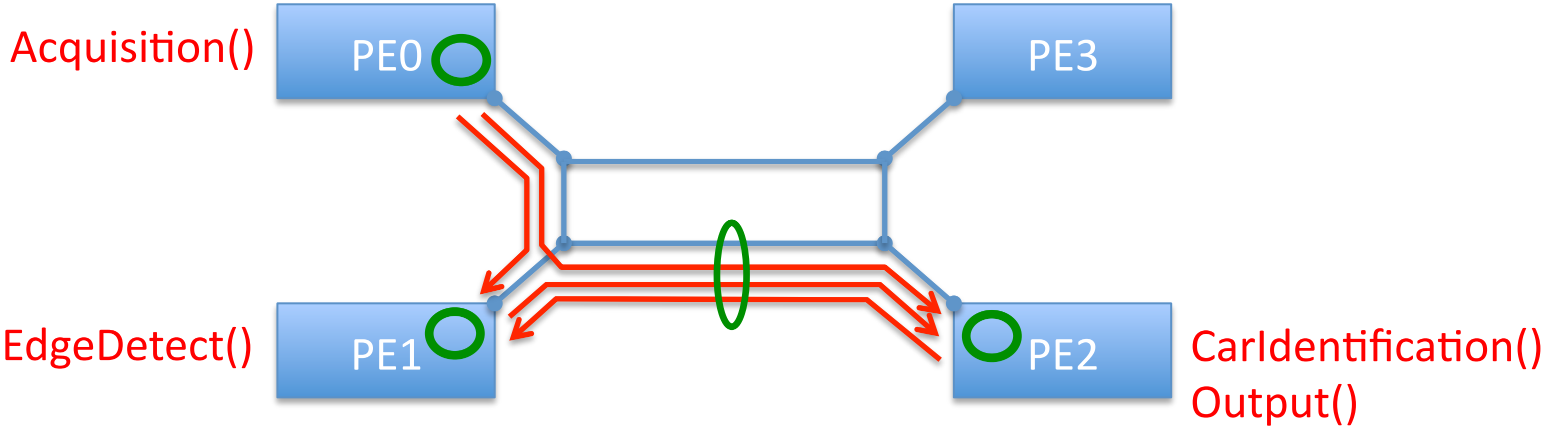
Embedded systems development

- Mapping in space
  - Where (by whom) is the operation performed
  - Vocabulary:
    - CPU/RAM: **allocation, distribution**
    - Interconnect: **routing**



Embedded systems development

- Mapping in time
  - When is the operation performed (timing and/or order)
    - Resource allocation in concurrent systems
      - Implementation-level concurrency, HW or SW
  - Vocabulary:
    - CPU: **scheduling, sequencing**
    - Interconnect: **arbitration, scheduling, sequencing**



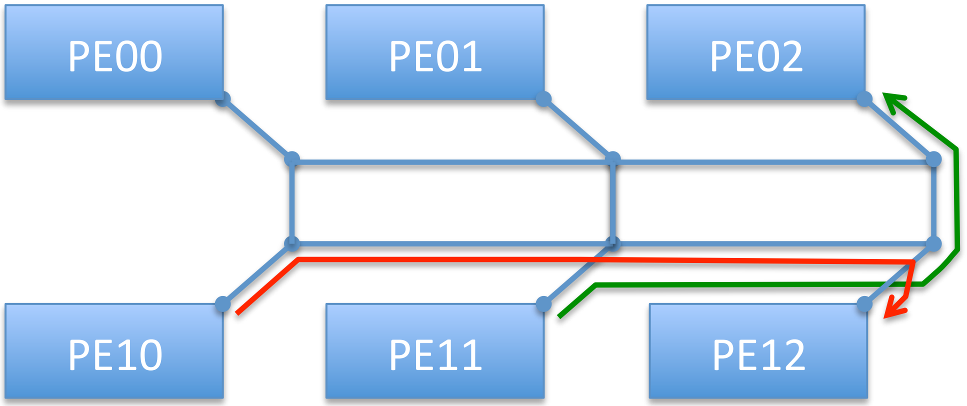
Embedded systems development

- Mapping complexity
  - Optimal: NP-hard at best, untractable in practice
  - Heuristics (experience-based techniques)
    - Among them: classical scheduling policies (RM, EDF, etc.)
    - May be optimal or formally characterized under restrictive hypotheses

- Classification of mapping techniques. Criterion 1:
  - Offline/Static
    - Mapping decisions are made before execution
      - By extension, decisions (conditional execution) of the functional specification are often allowed.
    - No timing/order imprecision (in some referential)
  - Online/Dynamic
    - Timing/order/etc. imprecision remains
    - Mapping decisions depend on system aspects that are unspecified or not analyzed off-line (too complex):
      - Input event arrival dates
      - Execution time variations
      - Unknown/unobservable OS/HW internals, etc…

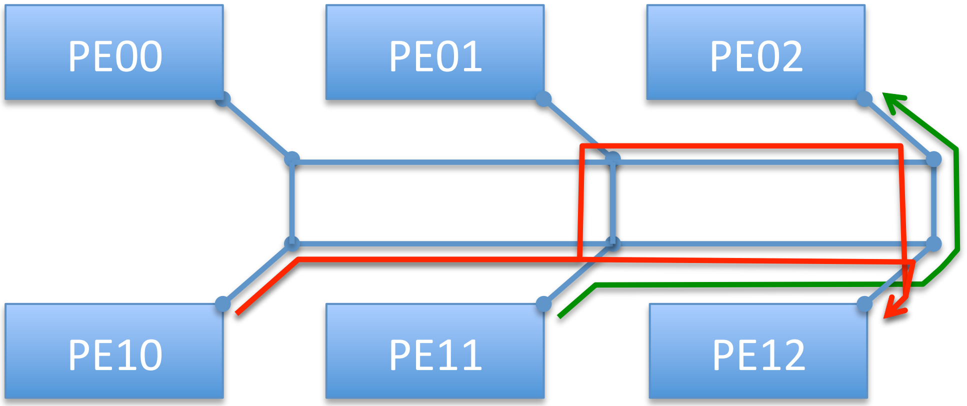
Embedded systems development

- Example on a piece of interconnect:
  - Problem: transmit 2 pieces of data
  - Static routing (X-first):



Embedded systems development

- Example on a piece of interconnect:
  - Problem: transmit 2 pieces of data
  - Dynamic routing (adaptive):



Embedded systems development

- Example on a CPU:
  - Problem: cyclically execute functions f(), g()
  - Dynamic scheduling:

Process1:

```
for(;;){
    f();
}
```

Process2:

```
for(;;){
    g();
}
```

Launch the processes under Linux

  - Static scheduling:

```
for(;;){
    f();
    g();
}
```

Constraints:

- Periods will be equal
- Scheduling must satisfy dependencies

Scheduling/arbitration (classification)

- Basic single-processor scheduling algorithms (policies):
  - Simplest: Fixed order, FIFO
  - Fair policies
    - (weighted) round robin
    - (weighted) fair queuing, etc.
  - Priority-based
    - Static priorities: FP, RM, DM
    - Dynamic priorities: EDF, LLF
    - …

  } Applicable online (low complexity)

  - Off-line heuristics (or exact algorithms)
    - Make off-line scheduling decisions that can be applied on-line
- Other choices:
  - Event-driven vs. Time-triggered (what triggers decisions?)
  - Preemptive vs. Non-preemptive (can we interrupt an operation?)
  - Partitioned vs. Global scheduling (mono- and multi-processor)
  - Single criticality vs. Mixed criticality
  - Fault-tolerant or not, etc.

Scheduling on multiprocessor systems

- **System mapping =**
  CPU mappings + interconnect mappings
  - Performance bottleneck can be in either CPU or interconnect, or in both
    - Depends on HW, on the functionality, and on the mapping itself (computation-intensive vs. communication-intensive).
  - Different algorithms are needed in different contexts on both CPUs and interconnect
    - FIFO scheduling is simple/low-cost
    - Fair algorithms are useful in soft real-time systems (e.g. signal processing)
    - Priority-based algorithms are useful when response time for some tasks is more important
    - Static scheduling is useful for regular processing (loop nests) and for safety-critical systems, …

Scheduling on multiprocessor systems

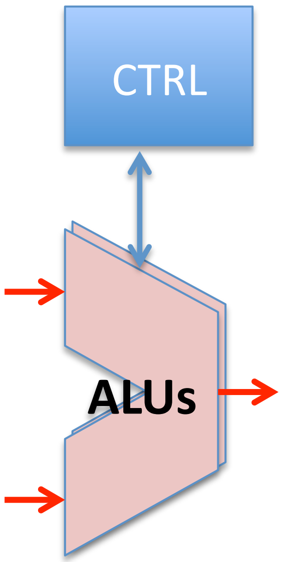
- There is however a major difference:
  - CPUs are programmable
    - CPUs can use any scheduling/allocation policy
    - Much work on synchronizing CPU schedules/allocations
  - Interconnect is (at best) configurable
    - Interconnect scheduling = CPU control + configuration
    - Configuration (choose one):
      - Scheduling/routing tables, Priorities, Assigned throughputs (e.g. config for weighted RoundRobin), etc.
    - Little work on synchronizing CPUs and interconnect schedules
      - More things on worst-case response time analysis (WCRT)
- Example:
  - Embedded networks
    - Fair arbitration (Ethernet), Priority-driven (CAN), static (TTA)
  - On-chip networks are similar, but changing the interconnect means changing the chip (expensive)

Scheduling on multiprocessor systems

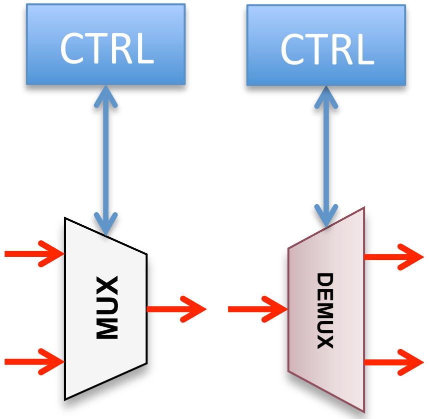
- Our thesis:

CPUs and interconnect should provide a similar level of control

CPU=



interconnect=



Scheduling on multiprocessor systems

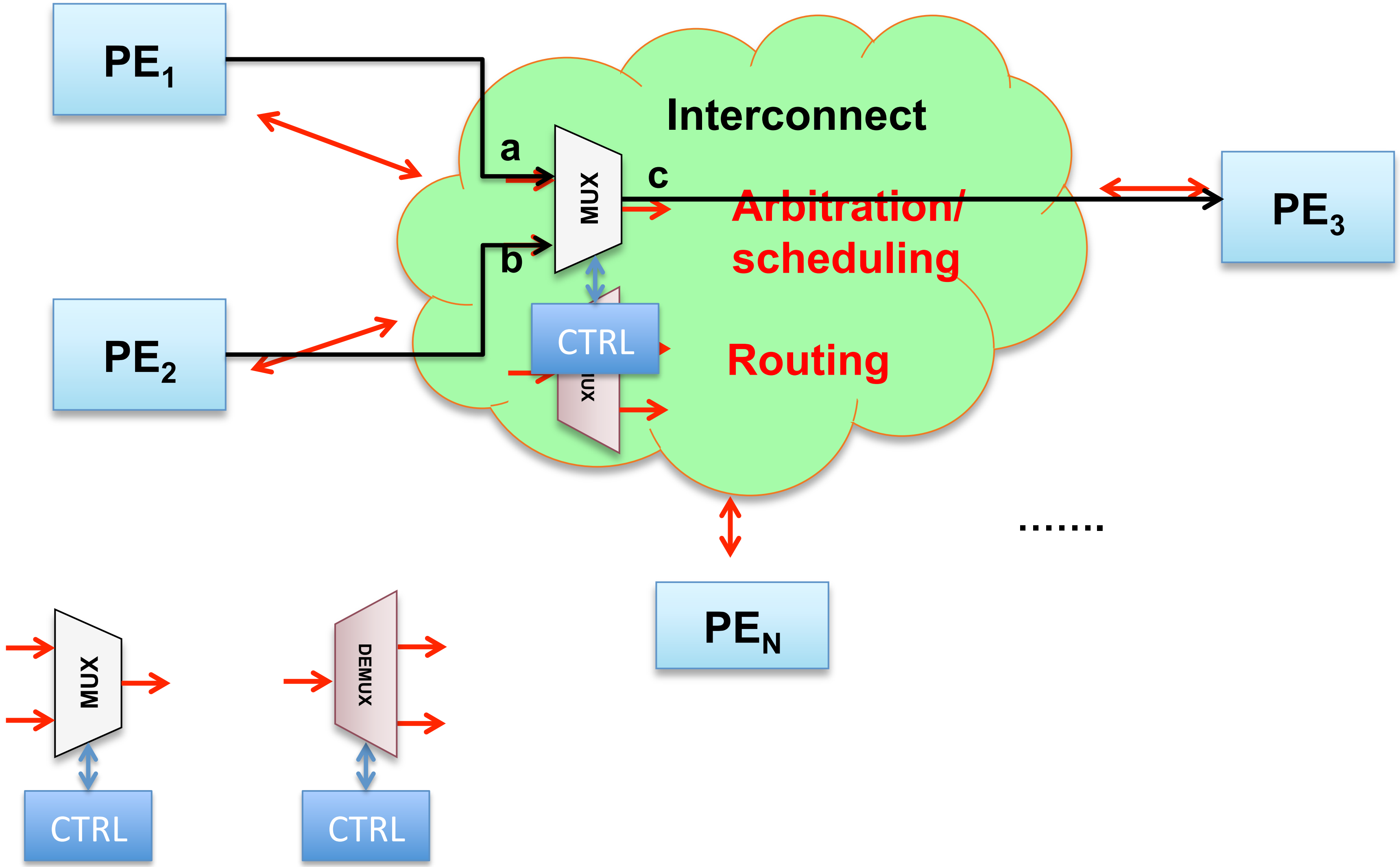
- Our thesis:

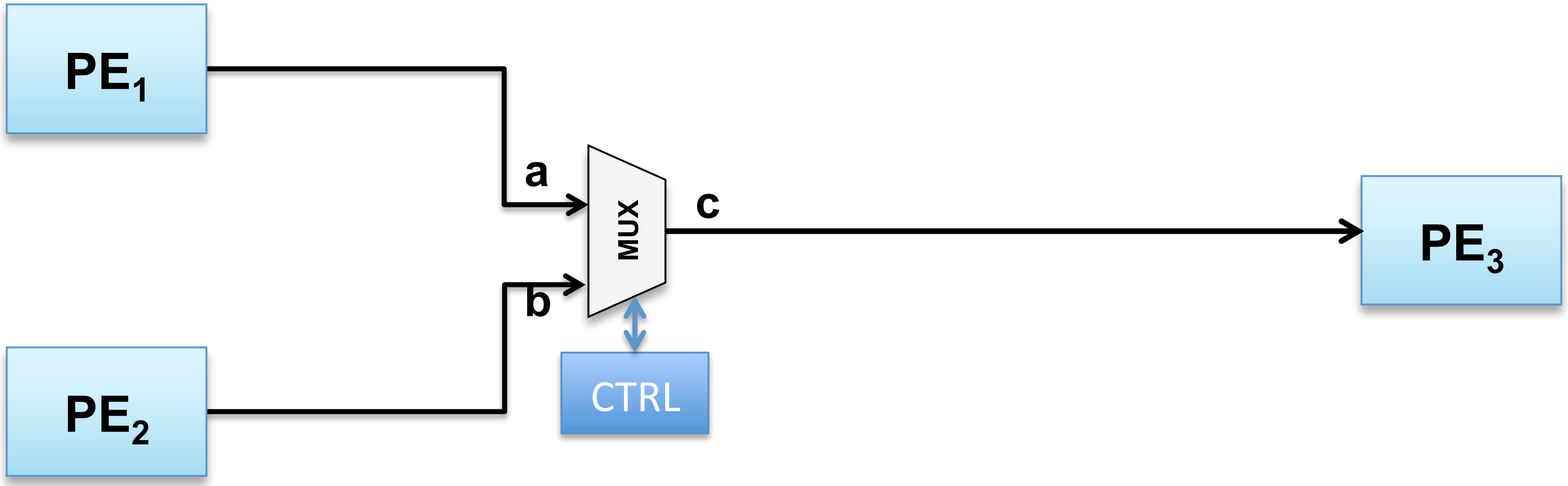
  CPUs and interconnect should provide a similar level of control

  - Previous attempts:
    - Scalar Operand Networks (MIT RAW, Waingold et al.)
      - Programmed interprocessor communication
    - Efficient Embedded Computing (Stanford ELM, Dally et al.)
      - Programmed prefetching for energy efficiency
    - Network Code (Fischmeister et al.)
      - Programmable network interfaces for time-triggered scheduling
    - Aethereal/CompSoC (Goossens et al.)
      - Programming-based configuration of NoCs for fairness

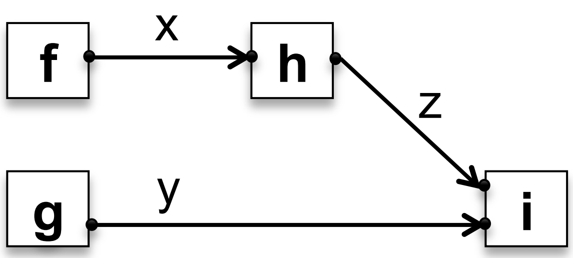
Scheduling on multiprocessor systems

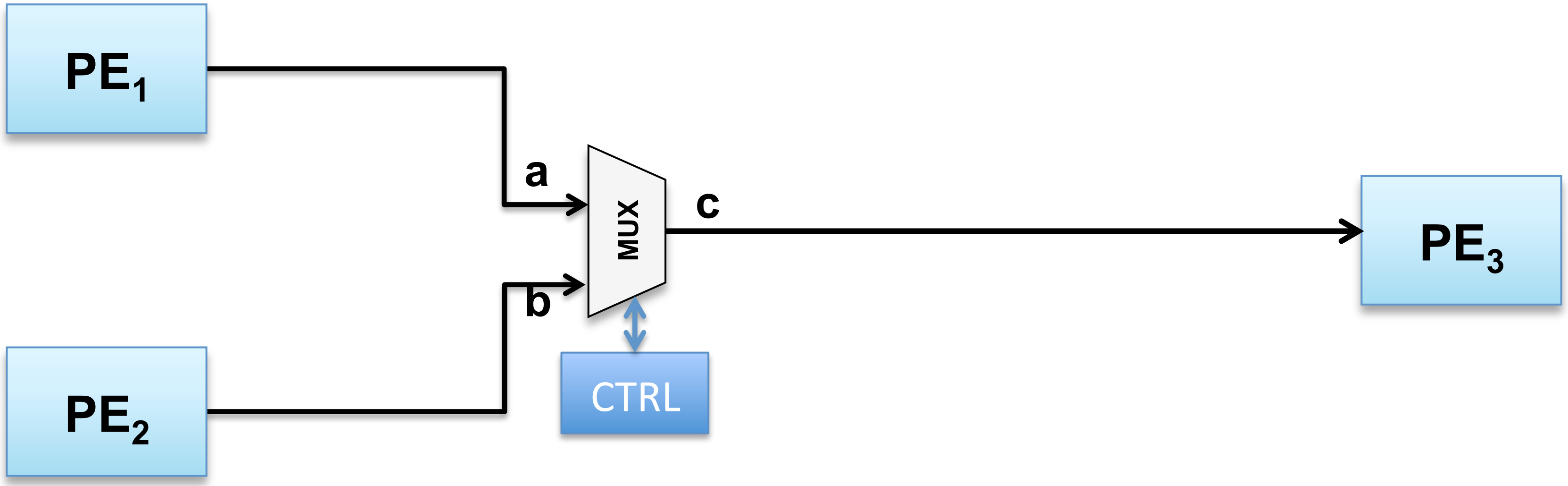
- Question:
  - What is the good level of interconnect control?
    - Trade-off between:
      - Programmability (control)
      - Complexity of program synthesis (global: interconnect & CPU)
      - Area
      - Speed
      - Low-power, thermal, etc.
    - Lots of work on this:
      - Tilera (5 networks), Kalray (Harrand and Durand’s patent), Fault tolerance in NoCs, Precision Timed Architectures, etc.
- Let’s take some examples !


PE$_1$
PE$_2$
PE$_3$
PE$_N$
Interconnect
a
b
c
MUX
CTRL
Arbitration/
scheduling
Routing
....
MUX
CTRL
DEMUX
CTRL

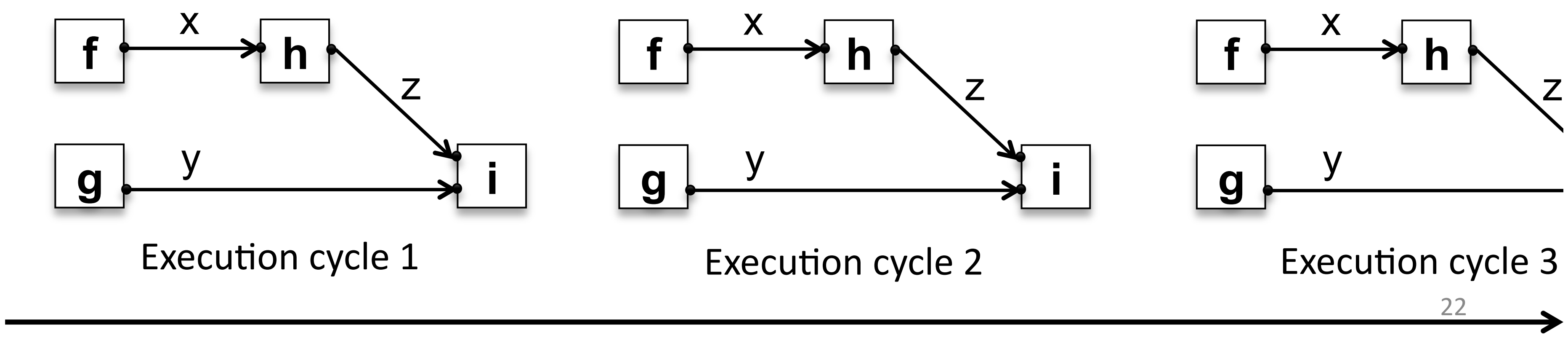


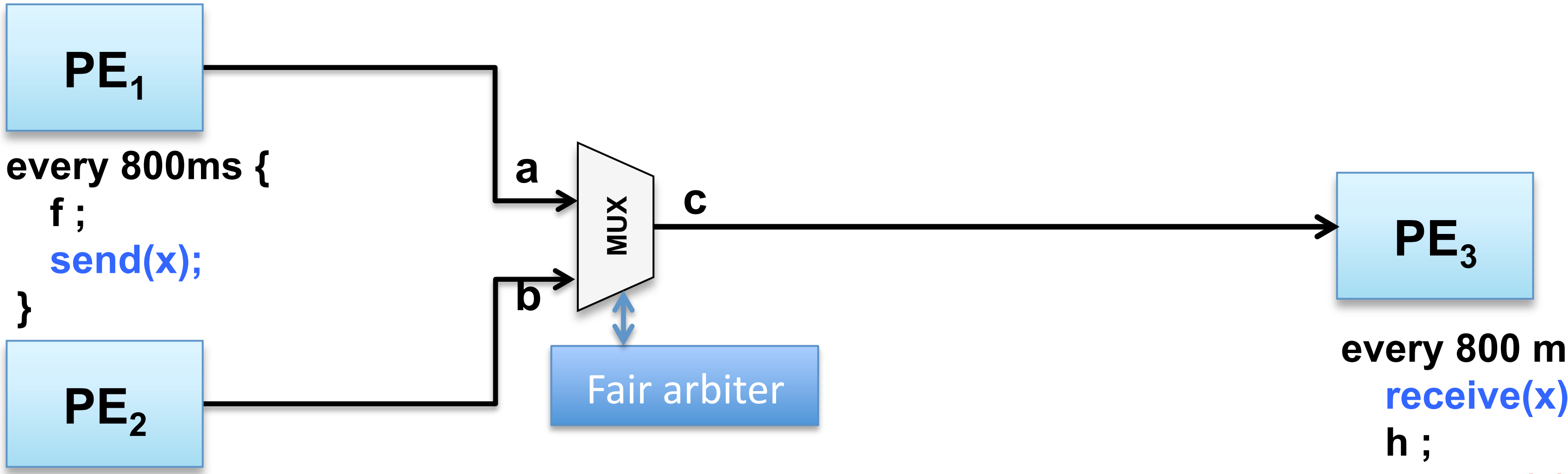
Data-flow functional specification





Data-flow functional specification
(Infinite) cyclic execution



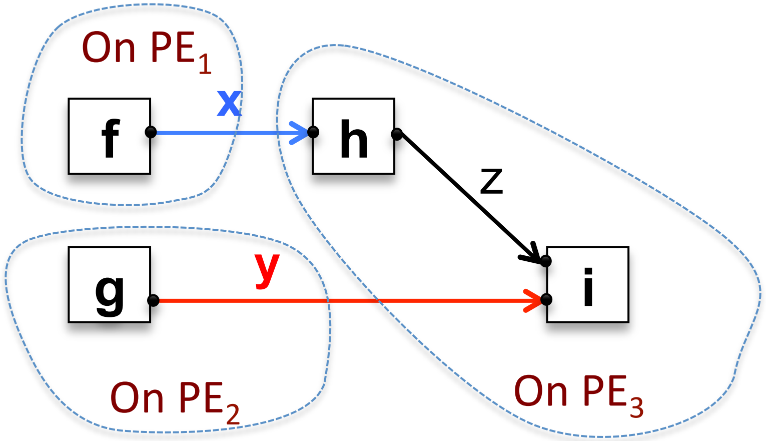


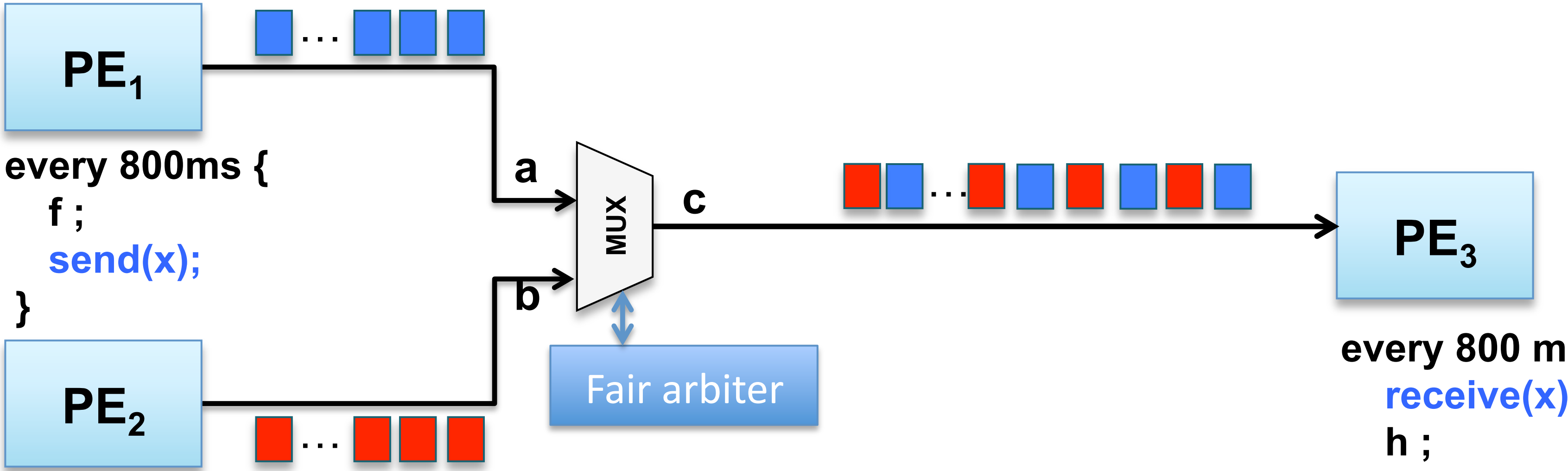
```
every 800ms {
    f ;
    send(x);
 }
```

```
every 800ms {
    g ;
    send (y);
}
```

```
every 800 ms {
    receive(x);
    h ;
    receive(y);
    i ;
 }
```

Data-flow functional specification, fully allocated, scheduled on the PEs

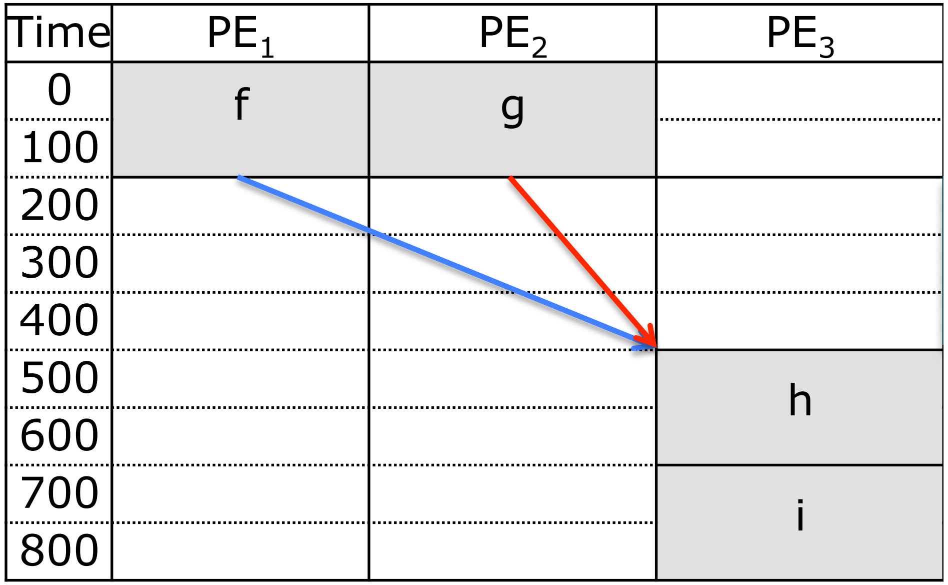




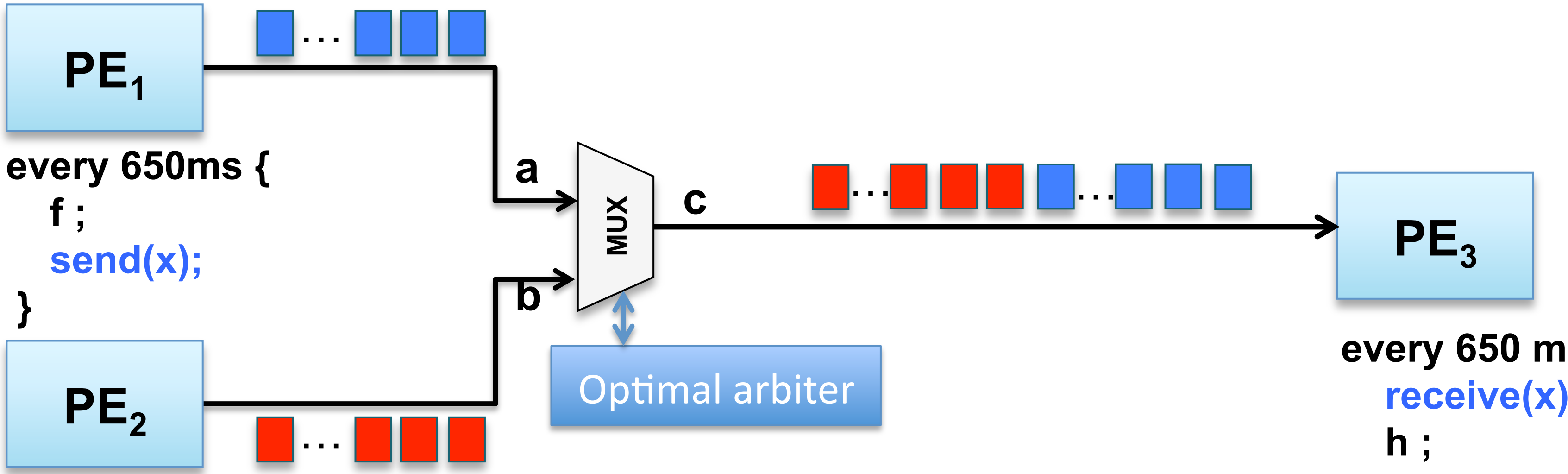
```
every 800ms {
    f ;
    send(x);
 }
```

```
every 800ms {
    g ;
    send (y);
}
```

```
every 800 ms {
    receive(x);
    h ;
    receive(y);
    i ;
}
```



Time	PE$_1$	PE$_2$	PE$_3$
0	f	g	
100			
200			
300			
400			
500			h
600			
700			i
800			

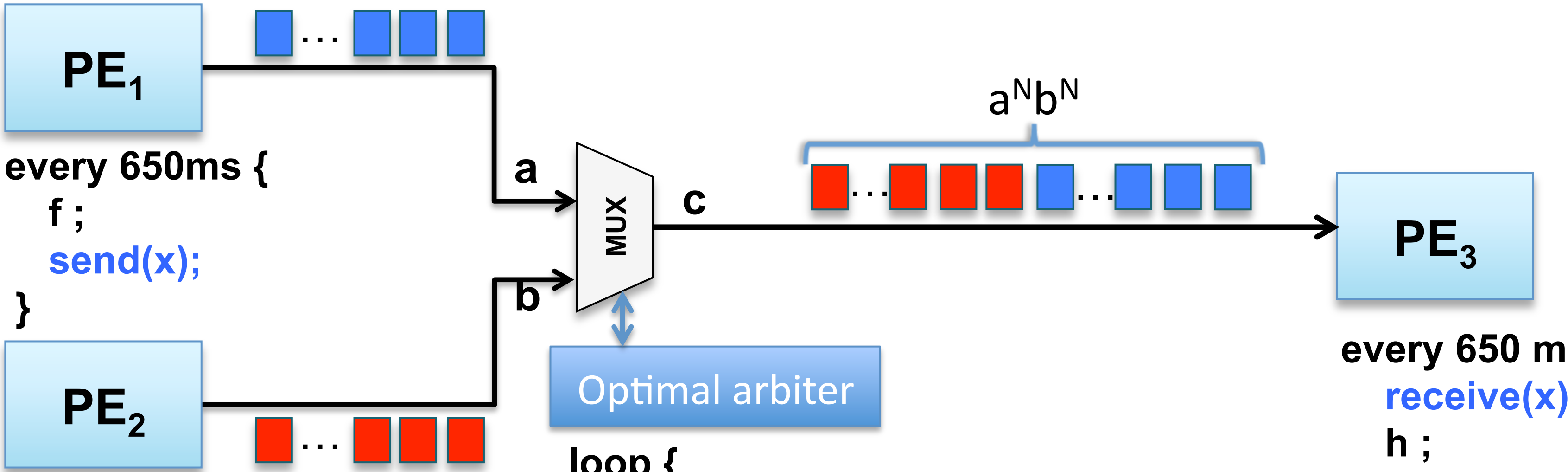


```
every 650ms {
    f ;
    send(x);
 }
```

```
every 650ms {
    g ;
    send (y);
}
```

```
every 650 ms {
    receive(x);
    h ;
    receive(y);
    i ;
}
```

Time	PE$_1$	PE$_2$	PE$_3$	MUX
0	f	g		
100				
200				🟦
300			h	🟥
400				
500			i	
600				
700				
800				

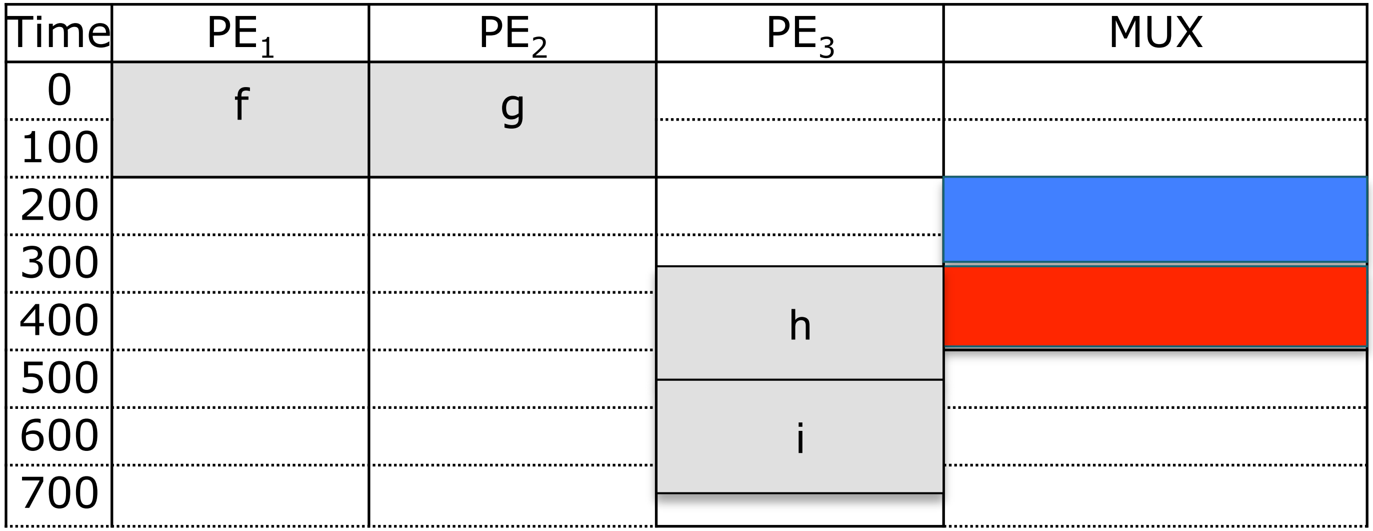


```
every 650ms {
    f ;
    send(x);
}
```

```
every 650ms {
    g ;
    send (y);
}
```

```
loop {
    do N times grant(a) ;
    do N times grant(b) ;
}
```

```
every 650 ms {
    receive(x);
    h ;
    receive(y);
    i ;
}
```



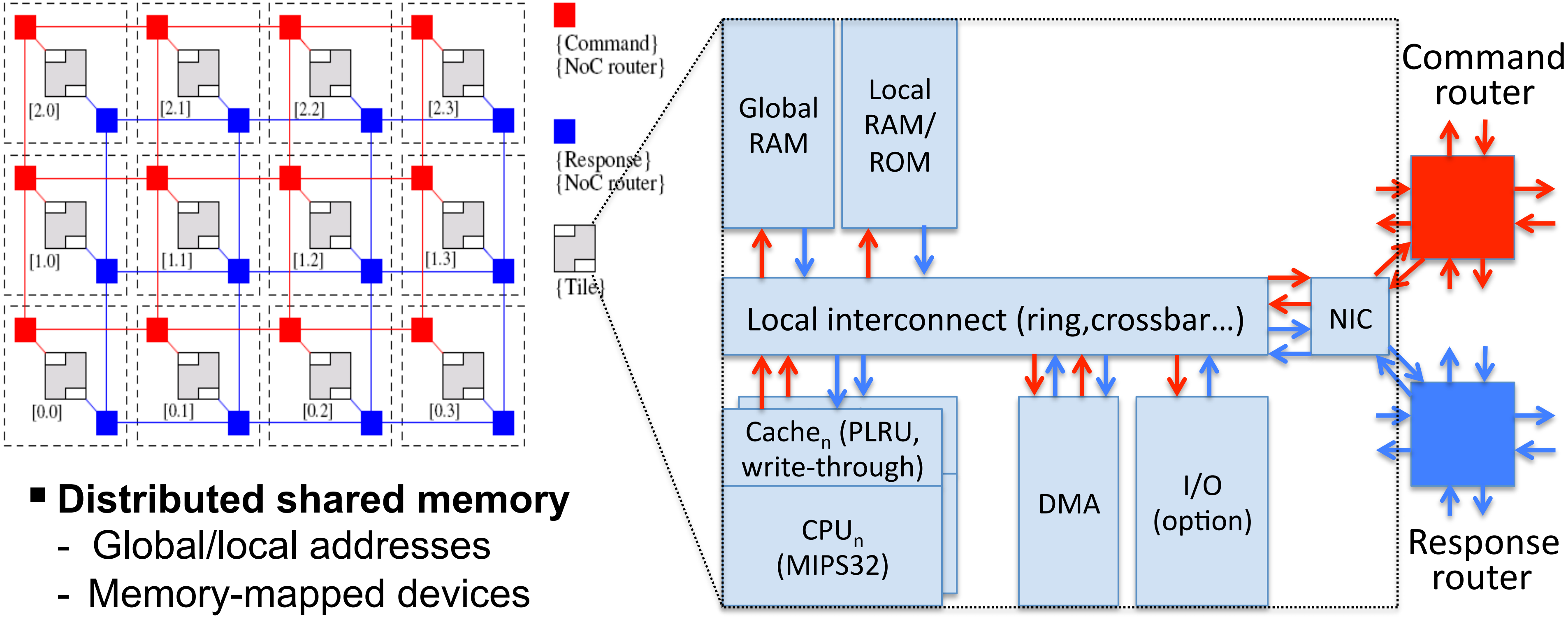
In the general case, priority-based arbitration is not optimal, either.

- ASAP scheduling is not optimal

Can we/Should we program a NoC in practice ?

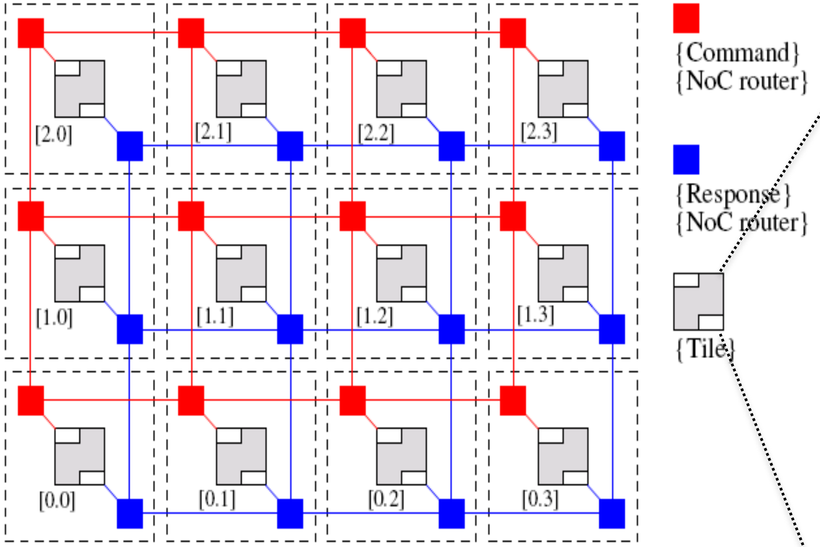
- Does programming help?
  - Efficiency issues
- Is the cost of programmability reasonable ?
  - Circuit area
  - Programming model changes
  - Synthesis of the network programs
- Workbench: DSPIN 2D mesh NoC

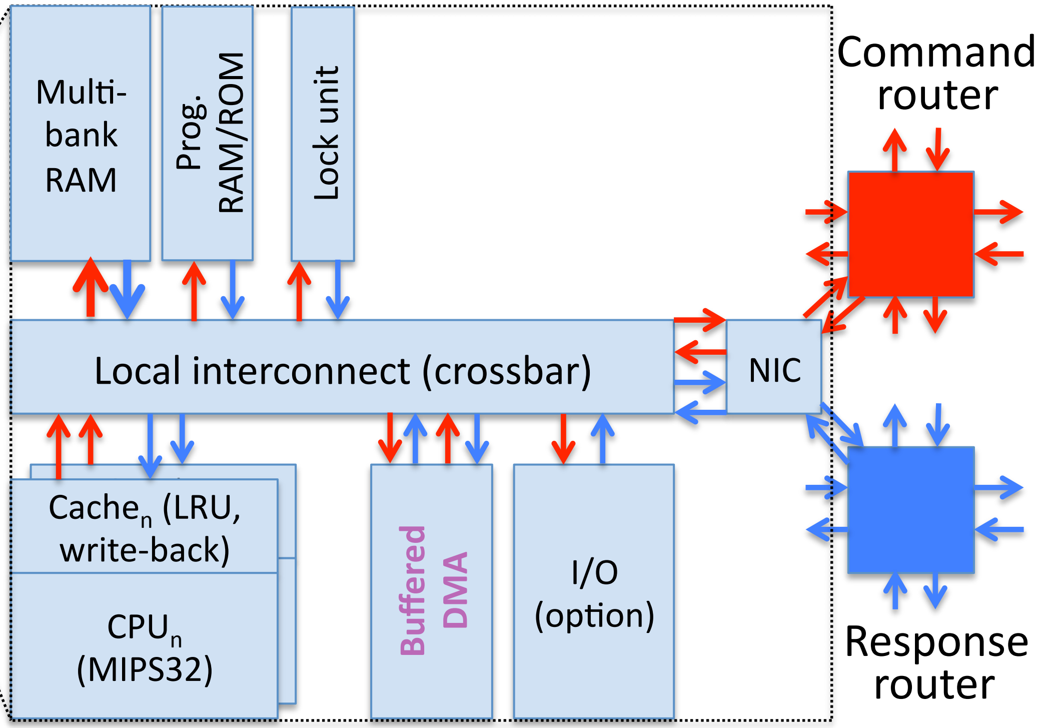
Tiled MPSoC architectures in SoCLib



- **Distributed shared memory**
  - Global/local addresses
  - Memory-mapped devices
  - Simple programming model & tools (gcc cross-compiler)
- **VCI/OCP protocol (command and response networks)**
- **2D mesh Network-on-Chip**
  - Wormhole packet switching
  - Fair arbitration
  - X-first/Y-first wormhole routing for commands/responses
- **Simulation support:** SystemC-compatible CABA models (http://www.soclib.fr)

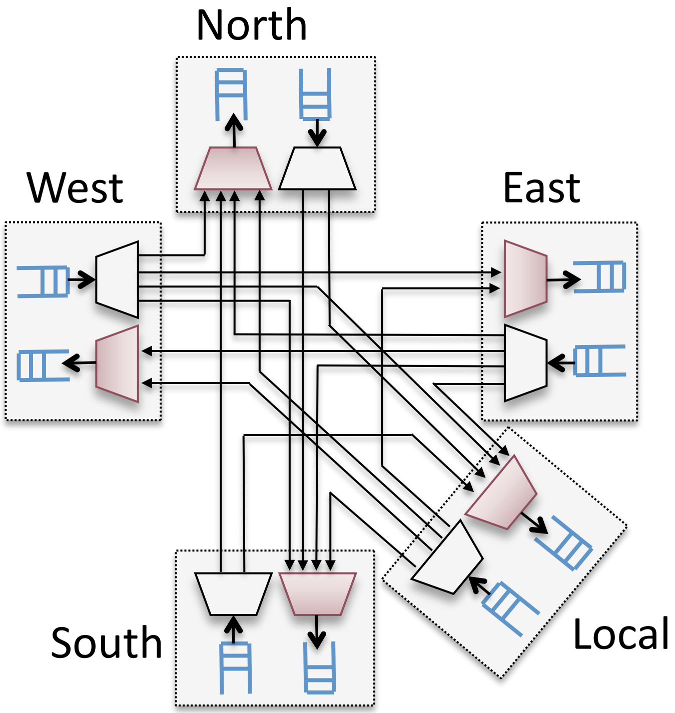
Tiled MPSoC architectures in SoCLib

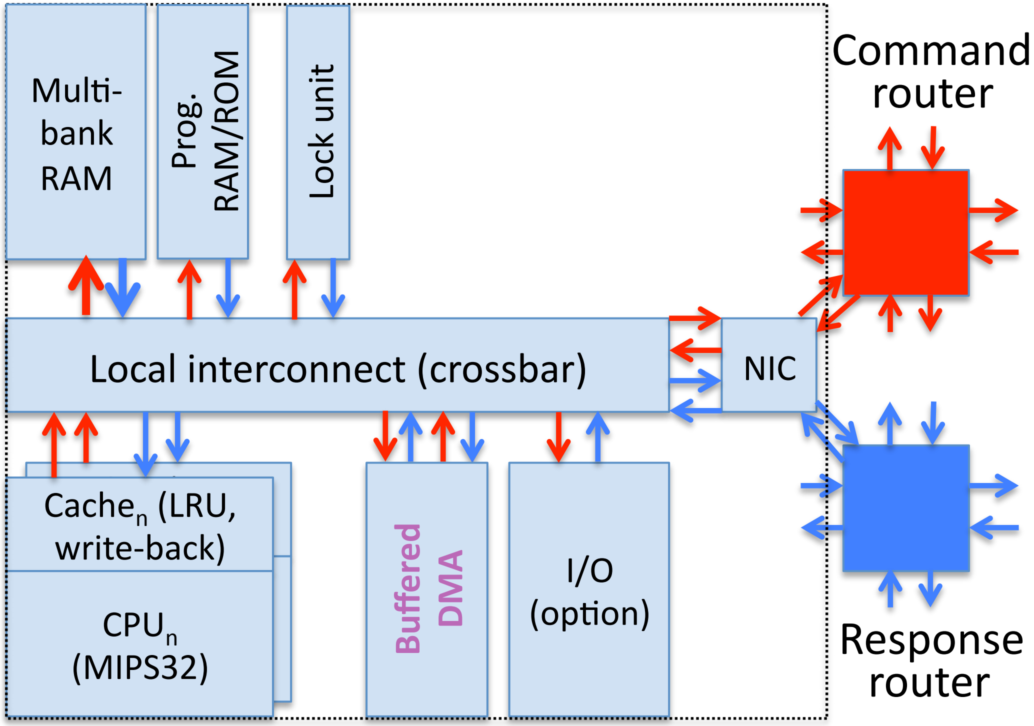




- **Tile architectural choices for better speed**
  - Multi-bank RAM
  - Separate program RAM/ROM
  - Low-overhead hardware locks (instead of interrupt-based synchronization)
  - Crossbar/logarithmic interconnect (reduced contentions)
  - Write-back caches (and we also change them to LRU for predictability)
  - DMAs with command buffers
  - Increased number of CPUs/tile (16)

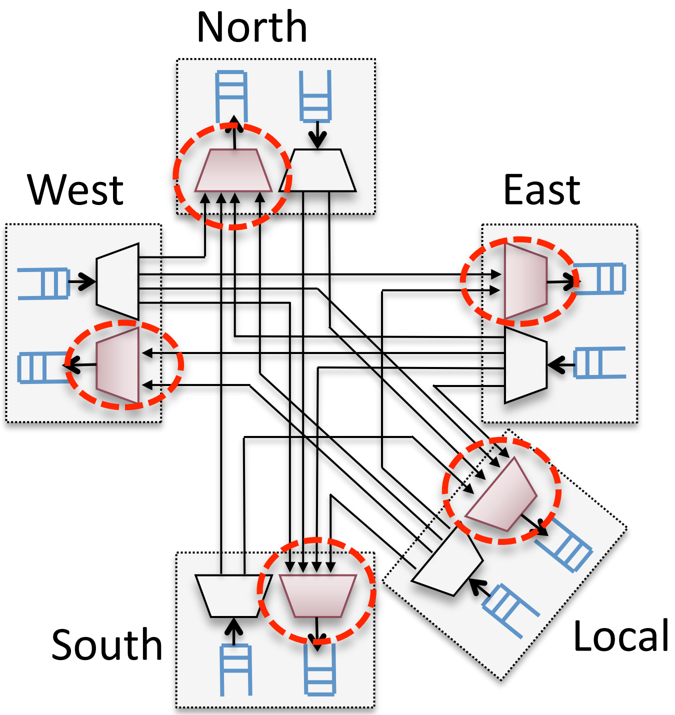
What's in a DSPIN NoC router?

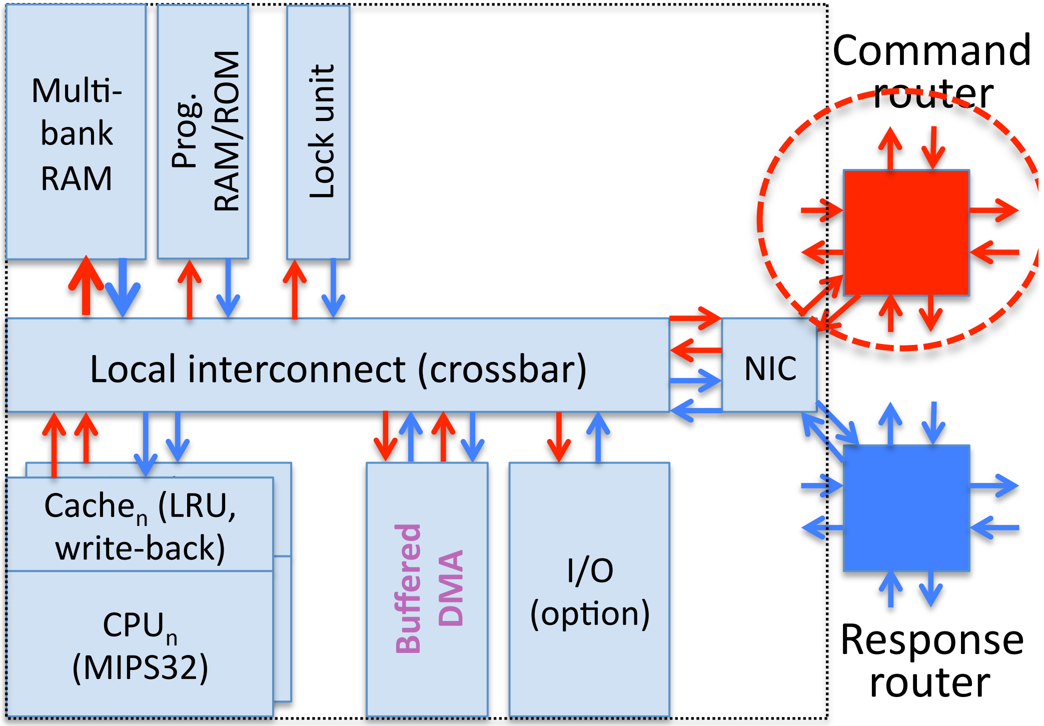




- **1 NoC router = 5 modules**
- **1 module = 1 MUX + 1DEMUX + control logic**
  - Static (X-first) routing
  - Fair arbitration
  - No configuration possible

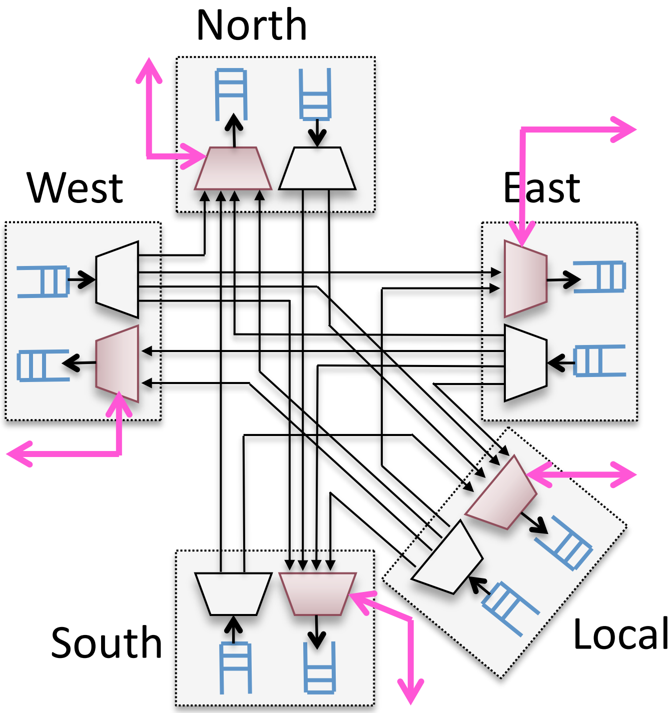
Adding programming to the NoC

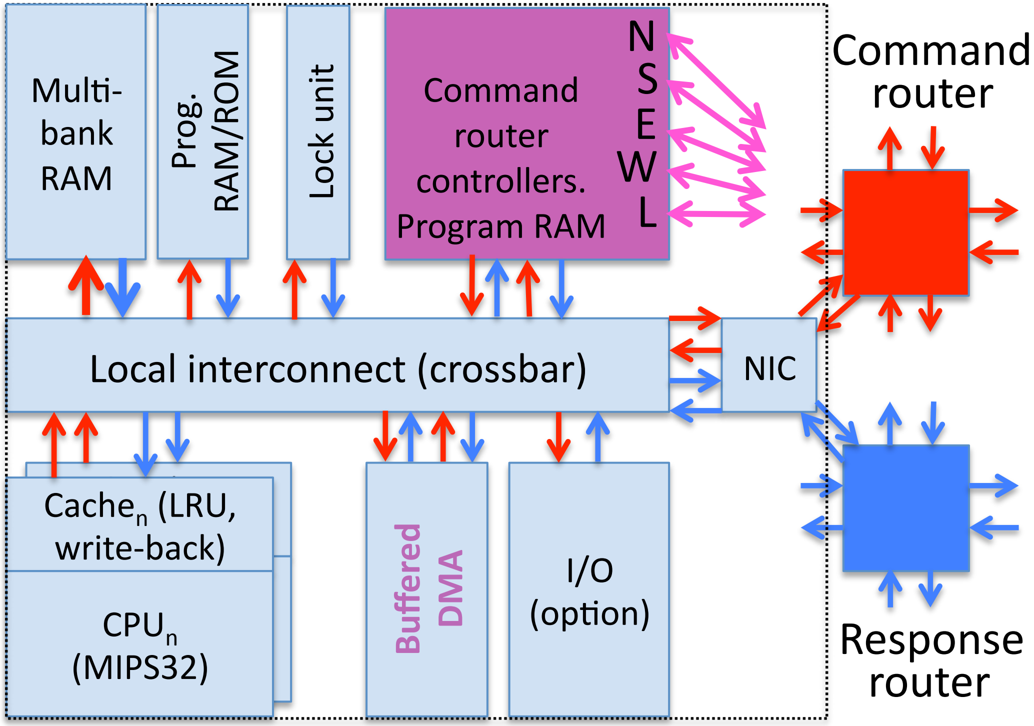




- **Programmability is expensive (mostly in program memory)**
  - **Network programs should be seen as equivalent to CPU programs**
- **Only the command router**
  - Transfers of data between tiles are performed with write operations
  - Response network only transfers 2-flits acknowledge packets (negligible contentions)
- **Only arbitration (not routing)**
  - Future work

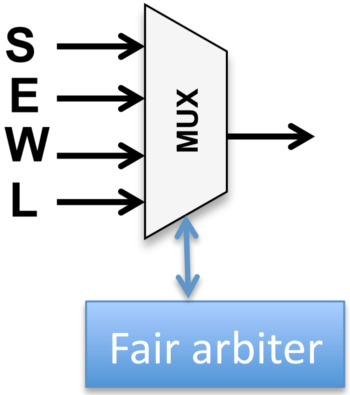
Adding programming to the NoC





- **Programmability is expensive (mostly in program memory)**
  - **Network programs should be seen as equivalent to CPU programs**
- **Only the command router**
  - Transfers of data between tiles are performed with write operations
  - Response network only transfers 2-flits acknowledge packets (negligible contentions)
- **Only arbitration (not routing)**
  - Future work

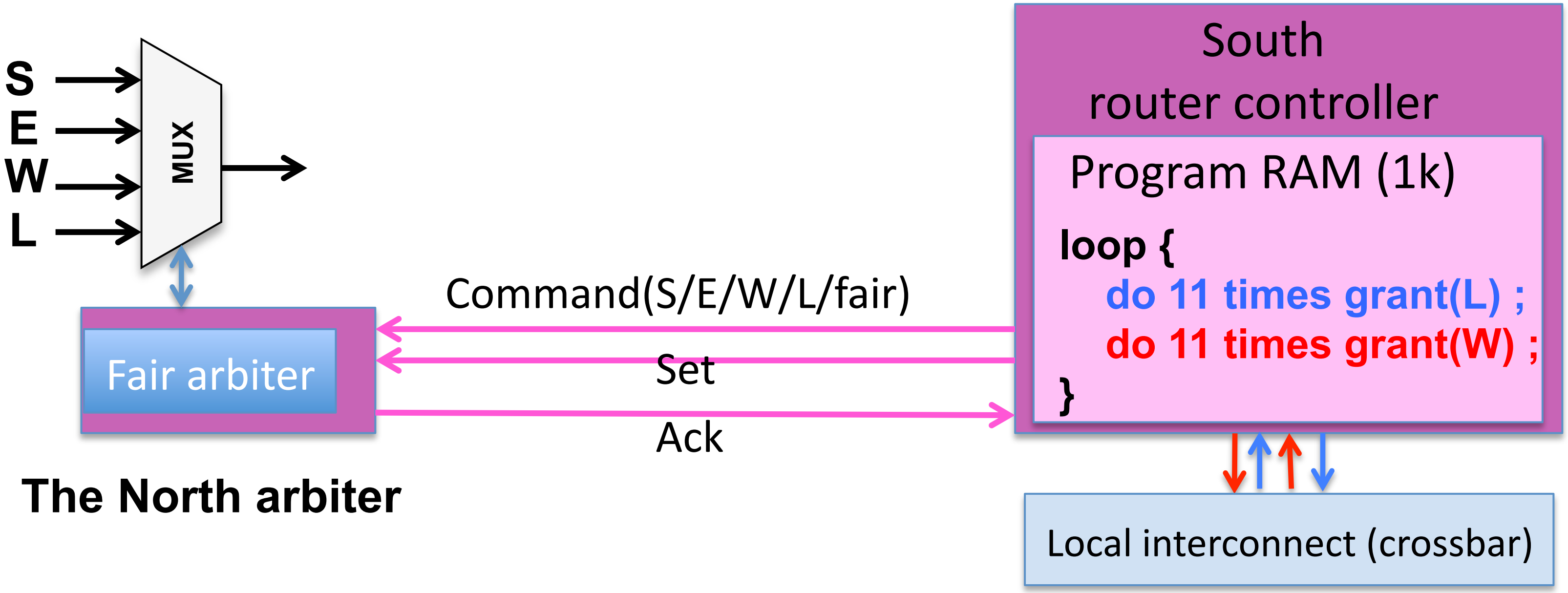
Adding programming to the NoC



The North arbiter

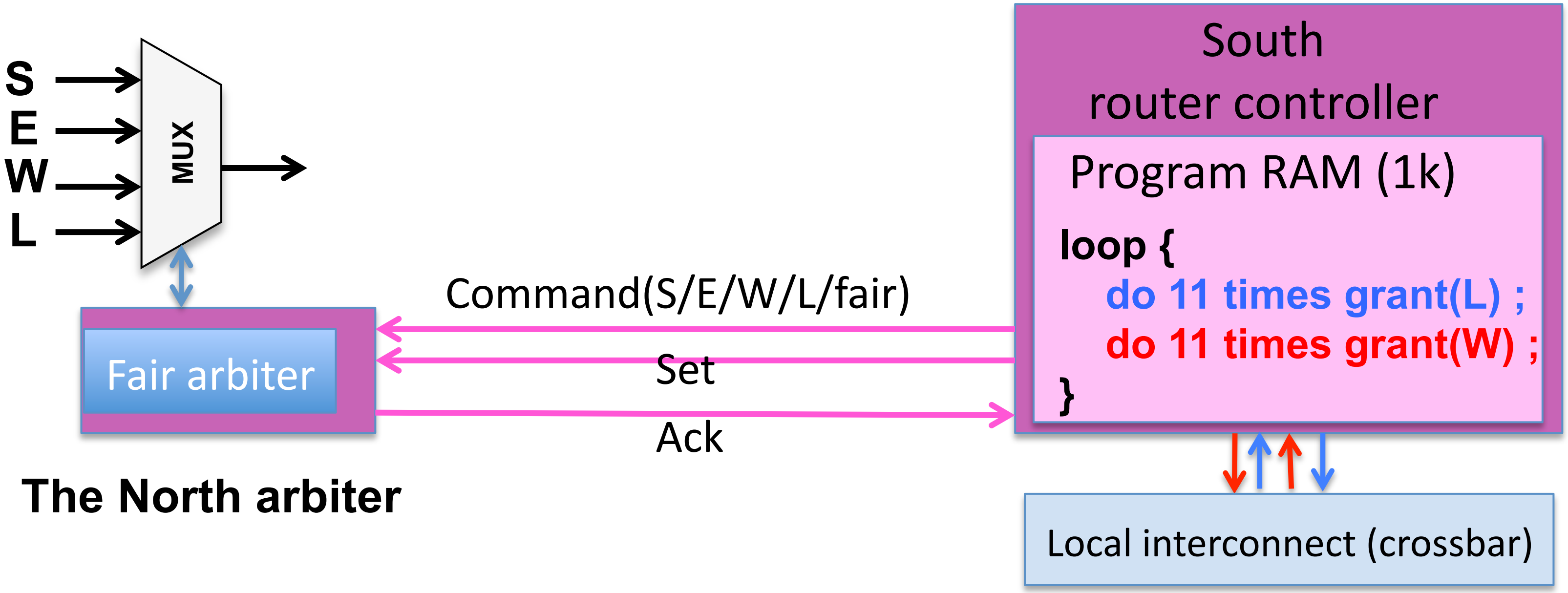
- **Programmability is expensive (mostly in program memory)**
  - **Network programs should be seen as equivalent to CPU programs**
- **Only the command router**
  - Transfers of data between tiles are performed with write operations
  - Response network only transfers 2-flits acknowledge packets (negligible contentions)
- **Only arbitration (not routing)**
  - Future work

Adding programming to the NoC



- **Programmability is expensive (mostly in program memory)**
  - **Network programs should be seen as equivalent to CPU programs**
- **Only the command router**
  - Transfers of data between tiles are performed with write operations
  - Response network only transfers 2-flits acknowledge packets (negligible contentions)
- **Only arbitration (not routing)**
  - Future work

Adding programming to the NoC



S
E
W
L
MUX

Fair arbiter

The North arbiter

Command(S/E/W/L/fair)

Set

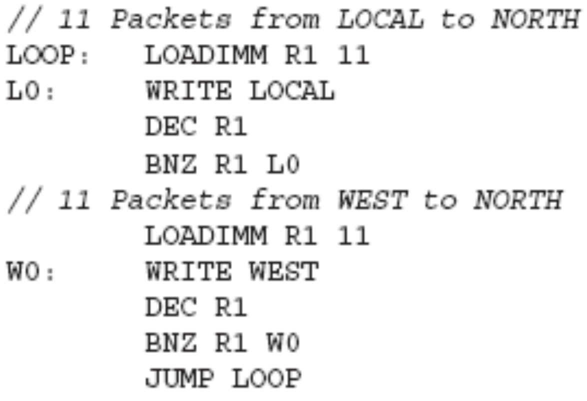
Ack

South
router controller

Program RAM (1k)

```
loop {
   do 11 times grant(L) ;
   do 11 times grant(W) ;
}
```

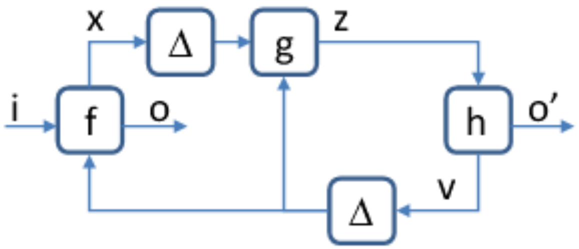
Local interconnect (crossbar)

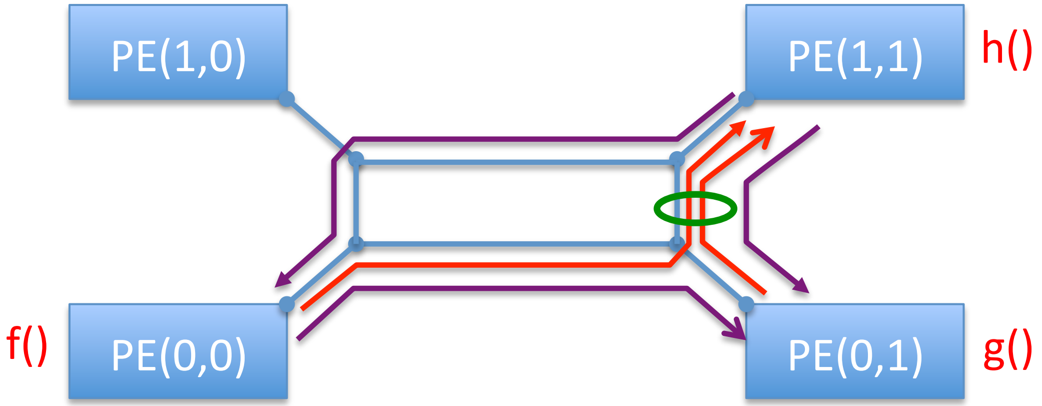


```
// 11 Packets from LOCAL to NORTH
LOOP:    LOADIMM R1 11
L0:      WRITE LOCAL
         DEC R1
         BNZ R1 L0
// 11 Packets from WEST to NORTH
         LOADIMM R1 11
W0:      WRITE WEST
         DEC R1
         BNZ R1 W0
         JUMP LOOP
```

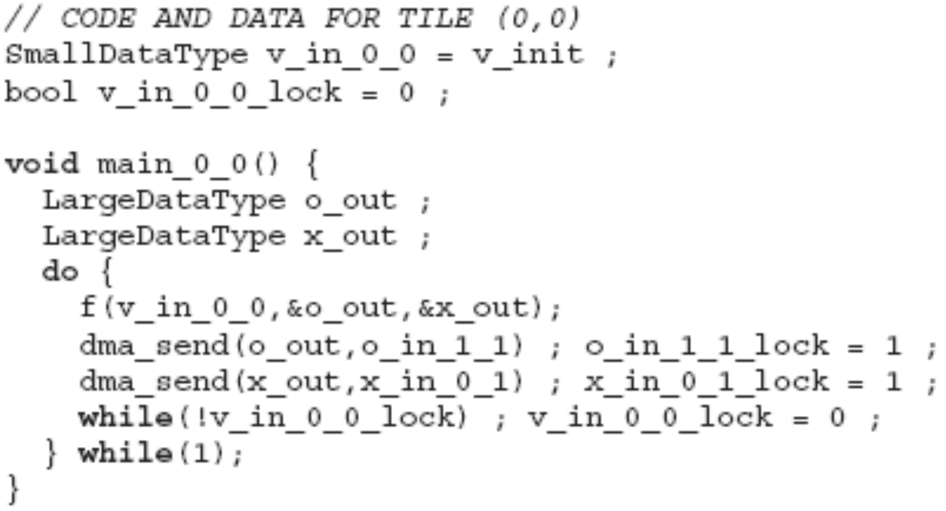
- **Simple instruction set (5 opcodes)**
  - Ease of implementation
  - Compact code
  - Can be optimized
- **No impact on speed**
  - Load next “write” while the current is executed
- **Simple extensions allows data-dependent control**
  - Inspection of packet header

Global view of the applications





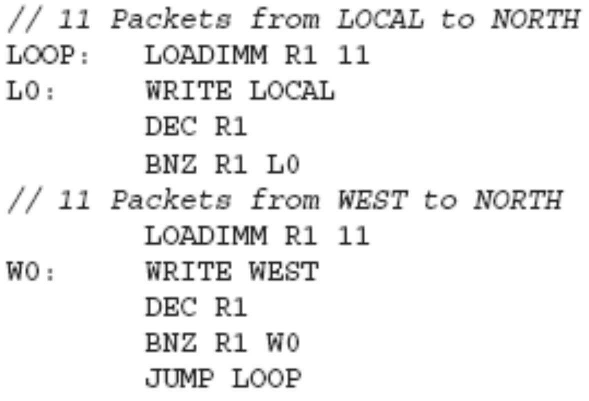
Application =
CPU programs + communication programs



```
// CODE AND DATA FOR TILE (0,0)
SmallDataType v_in_0_0 = v_init ;
bool v_in_0_0_lock = 0 ;

void main_0_0() {
  LargeDataType o_out ;
  LargeDataType x_out ;
  do {
    f(v_in_0_0,&o_out,&x_out);
    dma_send(o_out,o_in_1_1) ; o_in_1_1_lock = 1 ;
    dma_send(x_out,x_in_0_1) ; x_in_0_1_lock = 1 ;
    while(!v_in_0_0_lock) ; v_in_0_0_lock = 0 ;
  } while(1);
}
```

C code for tile (0,0)



```
// 11 Packets from LOCAL to NORTH
LOOP:   LOADIMM R1 11
L0:     WRITE LOCAL
        DEC R1
        BNZ R1 L0
// 11 Packets from WEST to NORTH
        LOADIMM R1 11
W0:     WRITE WEST
        DEC R1
        BNZ R1 W0
        JUMP LOOP
```

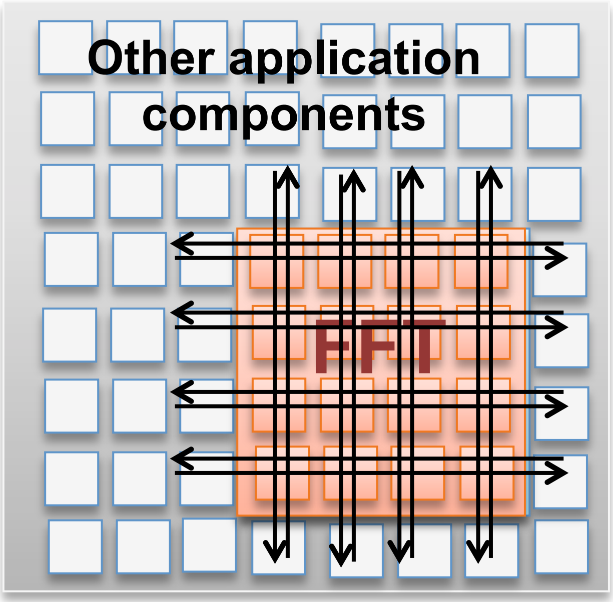
Assembly code for the North MUX of cluster (0,1)

Area cost of NoC programmability

- Simple NoC router controllers
- Area cost due mainly to program memory
  - 1kbytes of program memory x 5 = 5kbytes
  - 256kbytes RAM/tile
  - Result: <2% area overhead
- Moreover:
  - NoC program RAM contributes to the efficiency of the application just like the regular program RAM.

Case study: the FFT

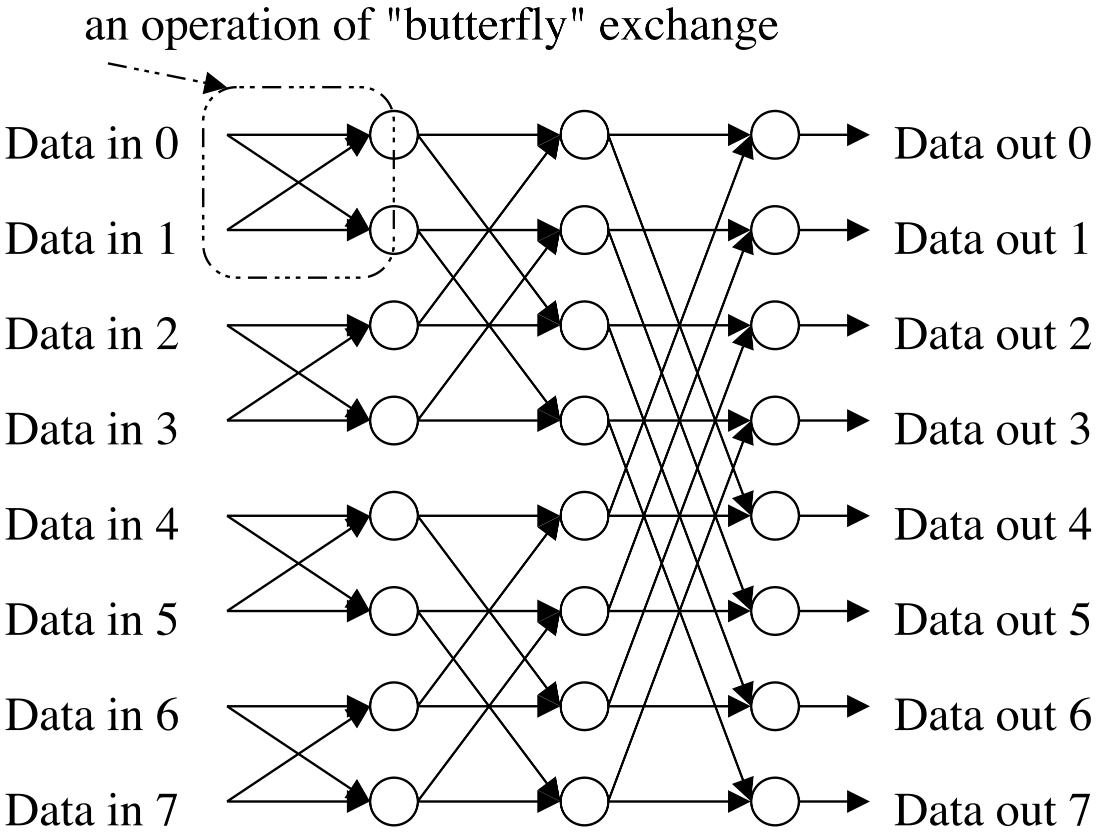
- Existing FFT application , already mapped
  - Hand-coded Cooley-Tukey FFT implementation (1D, radix 2, 2^{14}-2^{16} size FFT on a 4x4 MPSoC with 1,2,4,8,16 CPUs/tile)
- FFT-dedicated area is just a part of a larger MPSoC
- Other NoC applications transit data through the NoC of the FFT-dedicated area



- **Objective**: Allowing NoC traffic not belonging to the FFT without slowing down the FFT
- **How?** Schedule external traffic packets in the time slots where the NoC is free

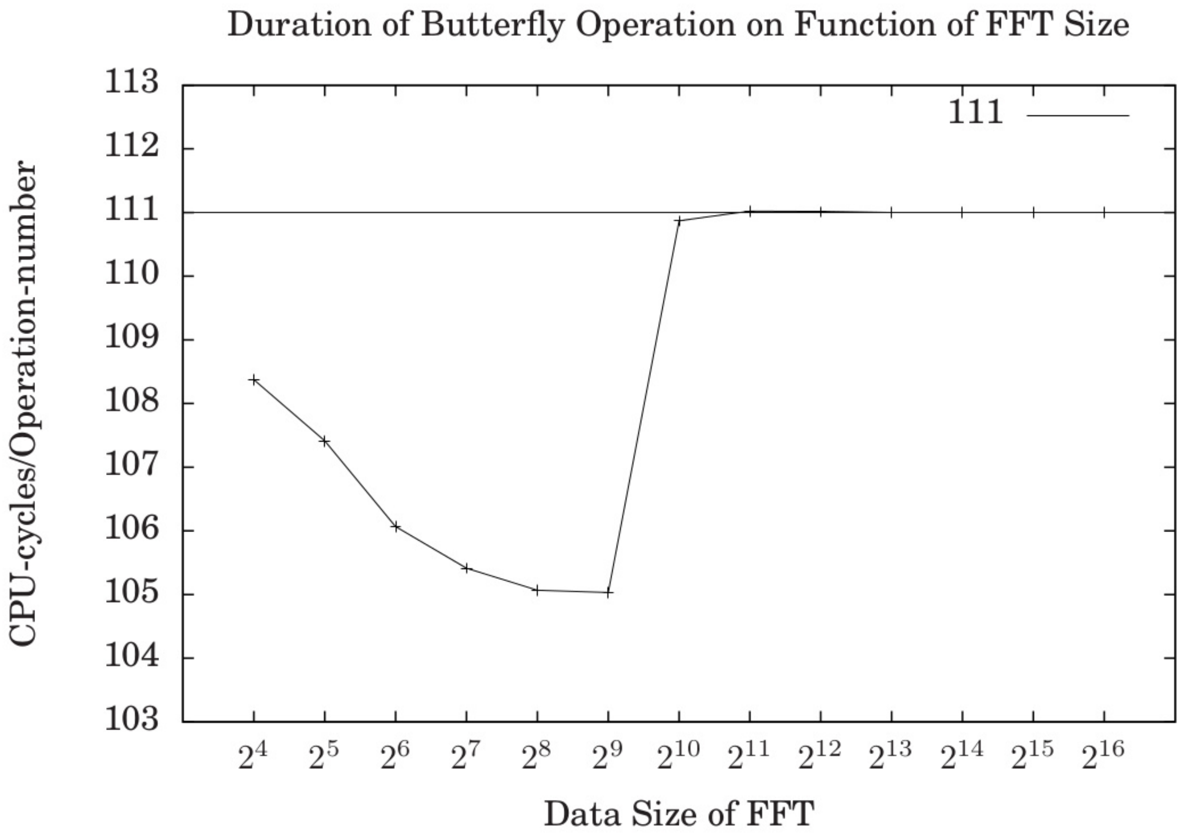
Understanding the FFT

- Succession of “butterfly” operations
- 2^n data => $n*2^{n-1}$ butterfly operations



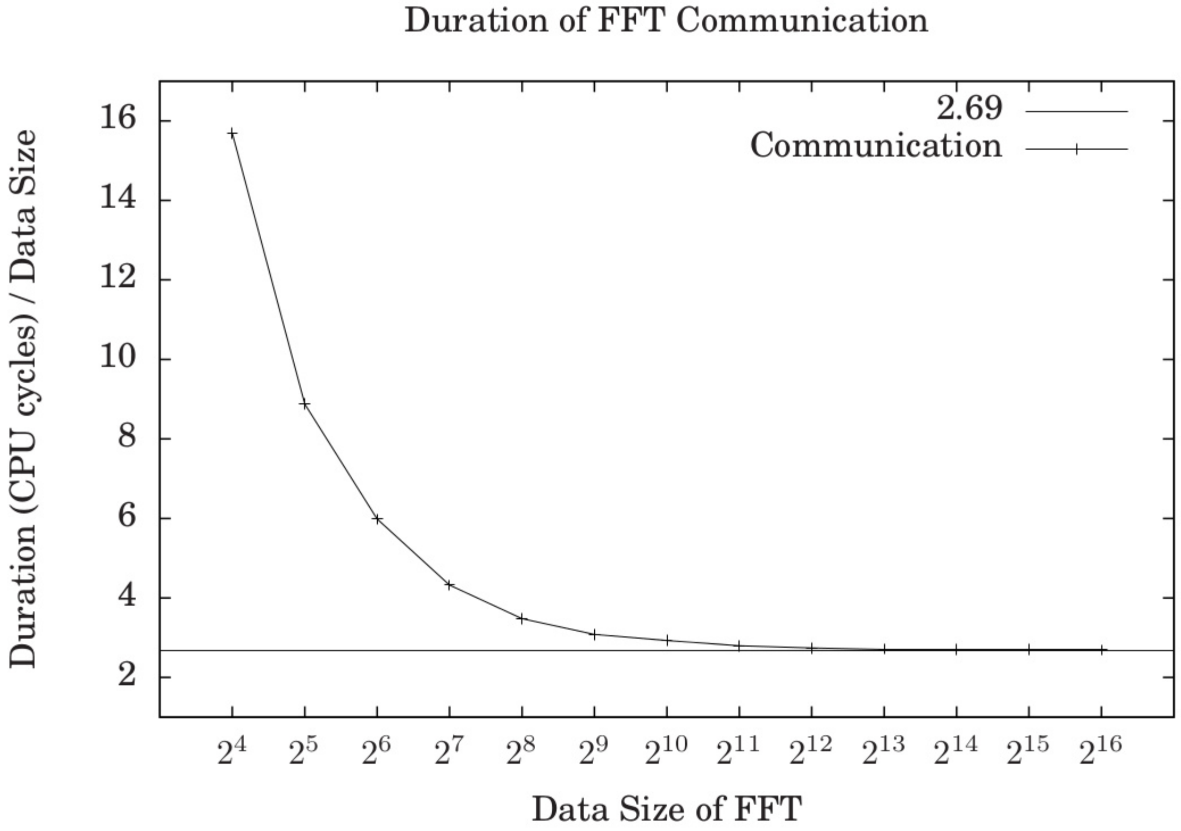
Understanding the FFT

- Succession of “butterfly” operations
- 2^n data => $n*2^{n-1}$ butterfly operations
- Duration of one butterfly operation – 111 cycles, asymptotically



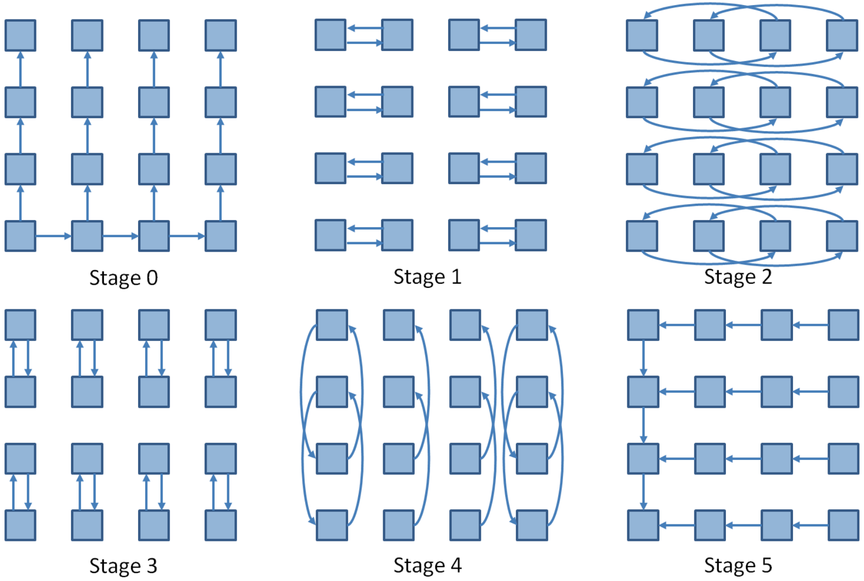
Understanding the FFT

- Succession of “butterfly” operations
- 2^n data => $n*2^{n-1}$ butterfly operations
- Duration of one butterfly operation – 111 cycles, asymptotically
- Duration of data transmissions – 2.69 cycles/data(dword), asymptotically



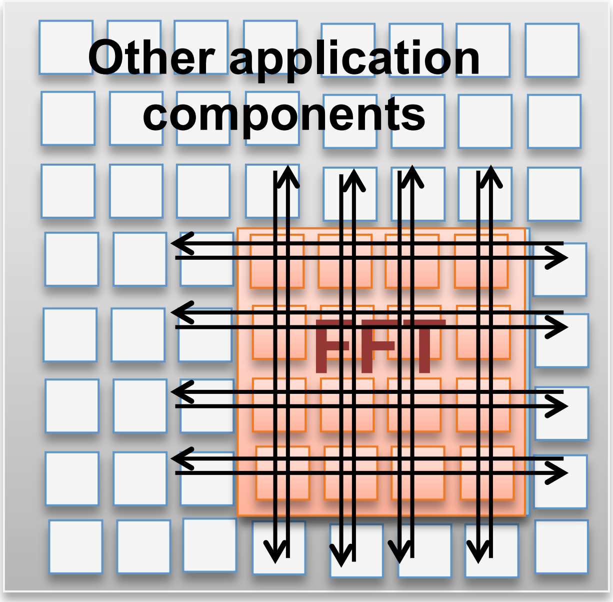
Understanding the FFT

- Succession of “butterfly” operations
- 2^n data => $n*2^{n-1}$ butterfly operations
- Duration of one butterfly operation – 111 cycles, asymptotically
- Duration of data transmissions – 2.69 cycles/data(dword), asymptotically
- Parallelisation: FFT of size 2^{n+1}, parallelized on 2^{k+1} processors = 2 FFTs of size 2^n on 2^k processors each, followed by 2^{n+1} butterflies (parallelized)
  - Attention to communications
  - In practice: doubling of the number of CPUs => ~1.8x acceleration If data is large enough
- Long computation phases separated by global synchronizations



Understanding the FFT

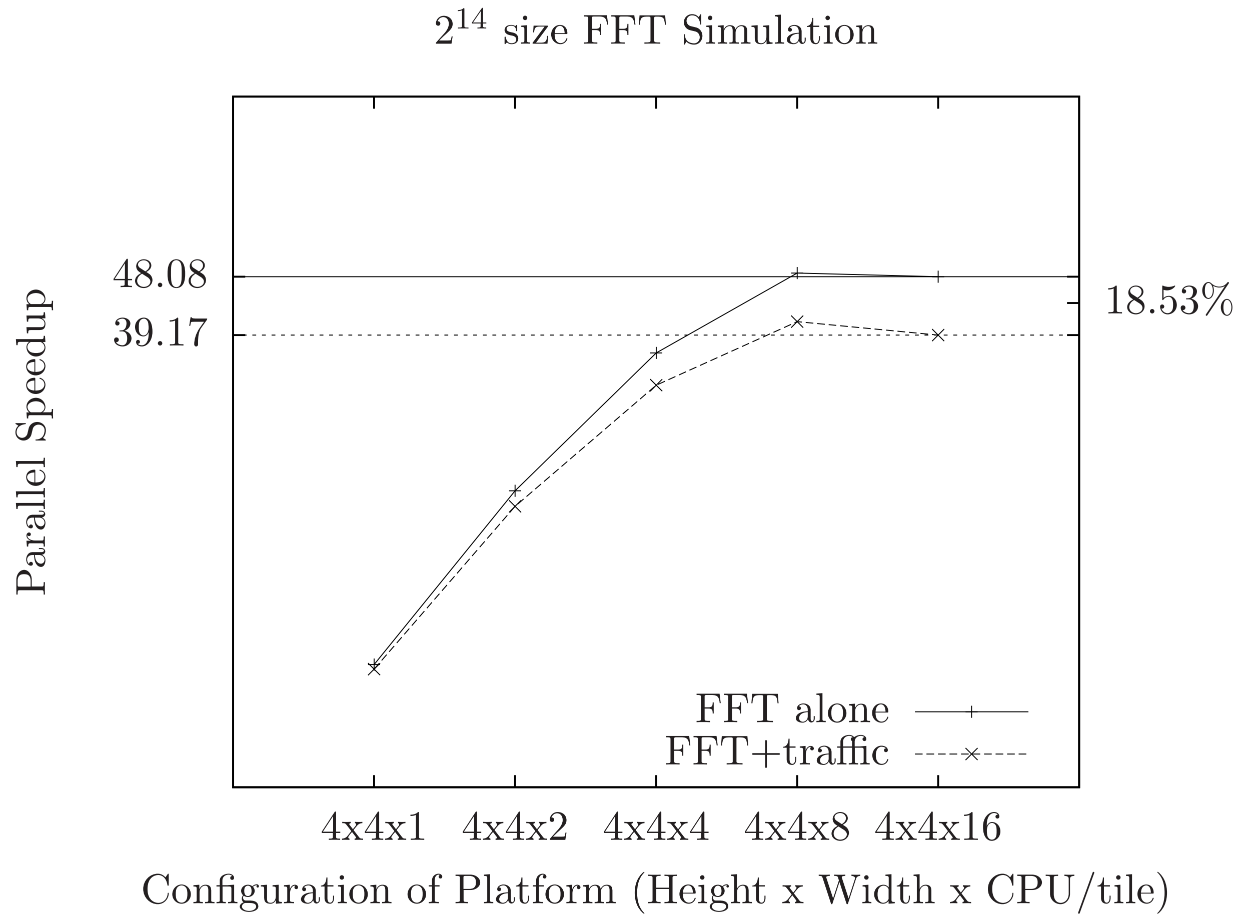
- Succession of "butterfly" operations
- 2^n data => $n*2^{n-1}$ butterfly operations
- Duration of one butterfly operation – 111 cycles, asymptotically
- Duration of data transmissions – 2.69 cycles/data(dword), asymptotically
- Parallelisation: FFT of size 2^{n+1}, parallelized on 2^{k+1} processors = 2 FFTs of size 2^n on 2^k processors each, followed by 2^{n+1} butterflies (parallelized)
  - Attention to communications
  - In practice: doubling of the number of CPUs => ~1.8x acceleration If data is large enough
- Long computation phases separated by
- Traffic injection:



Case study: the FFT

- Does traffic injection slow down the FFT?

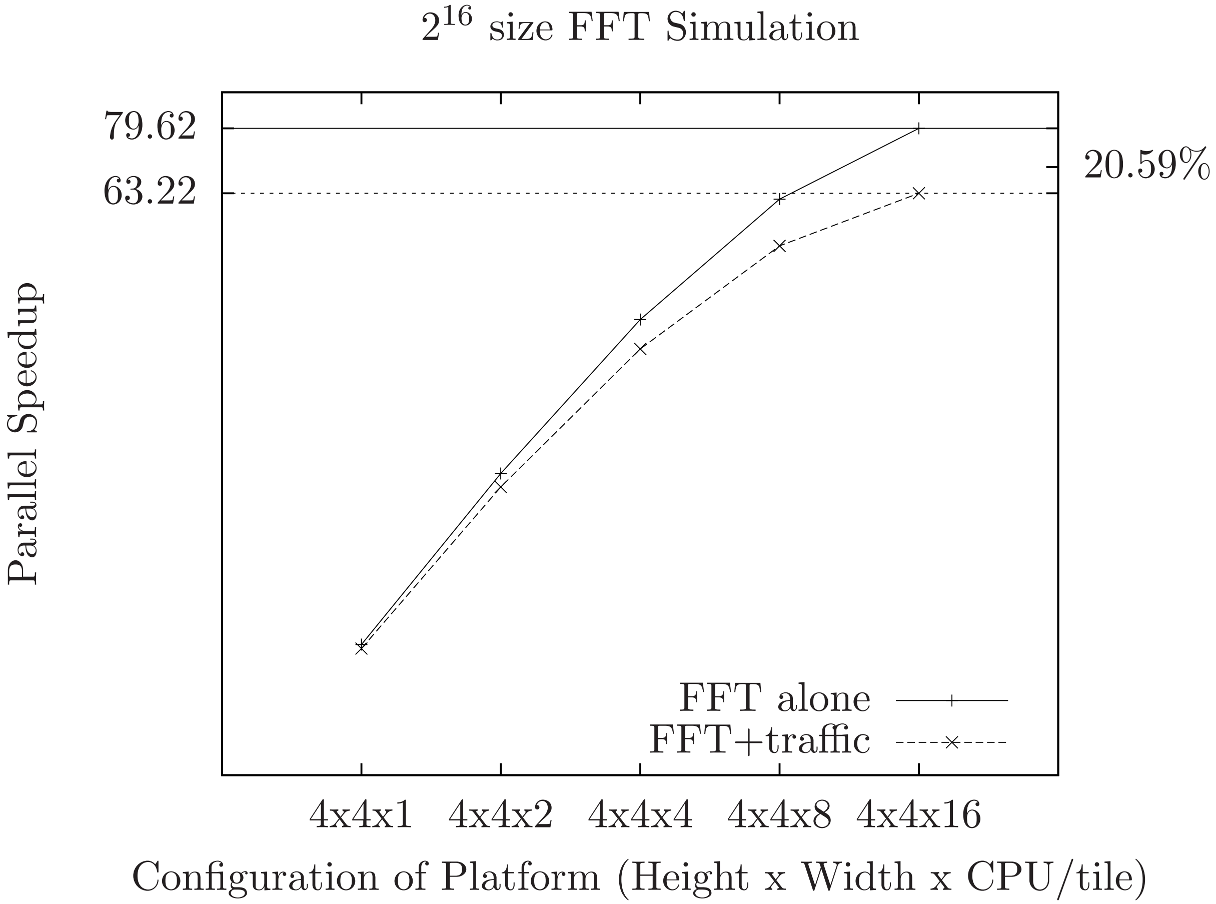
YES (significantly)



Case study: the FFT

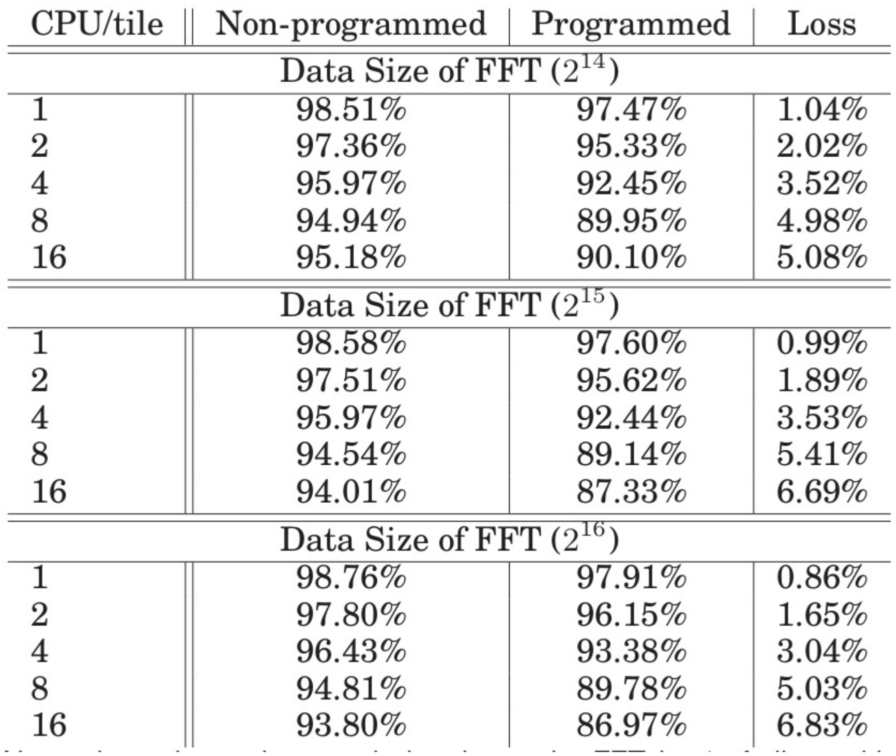
- Does traffic injection slow down the FFT?

YES (significantly)



Case study: the FFT

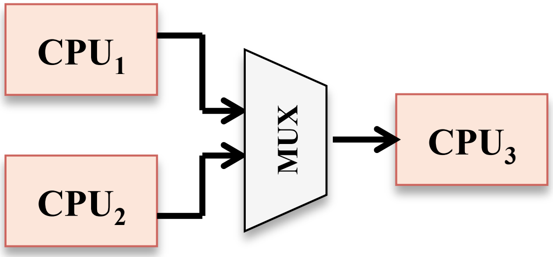
- Programming removes slow-down ? **YES (fully)**
- Cost in permeability? **SMALL**

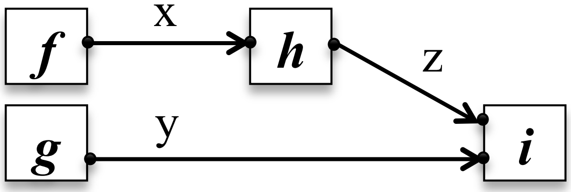


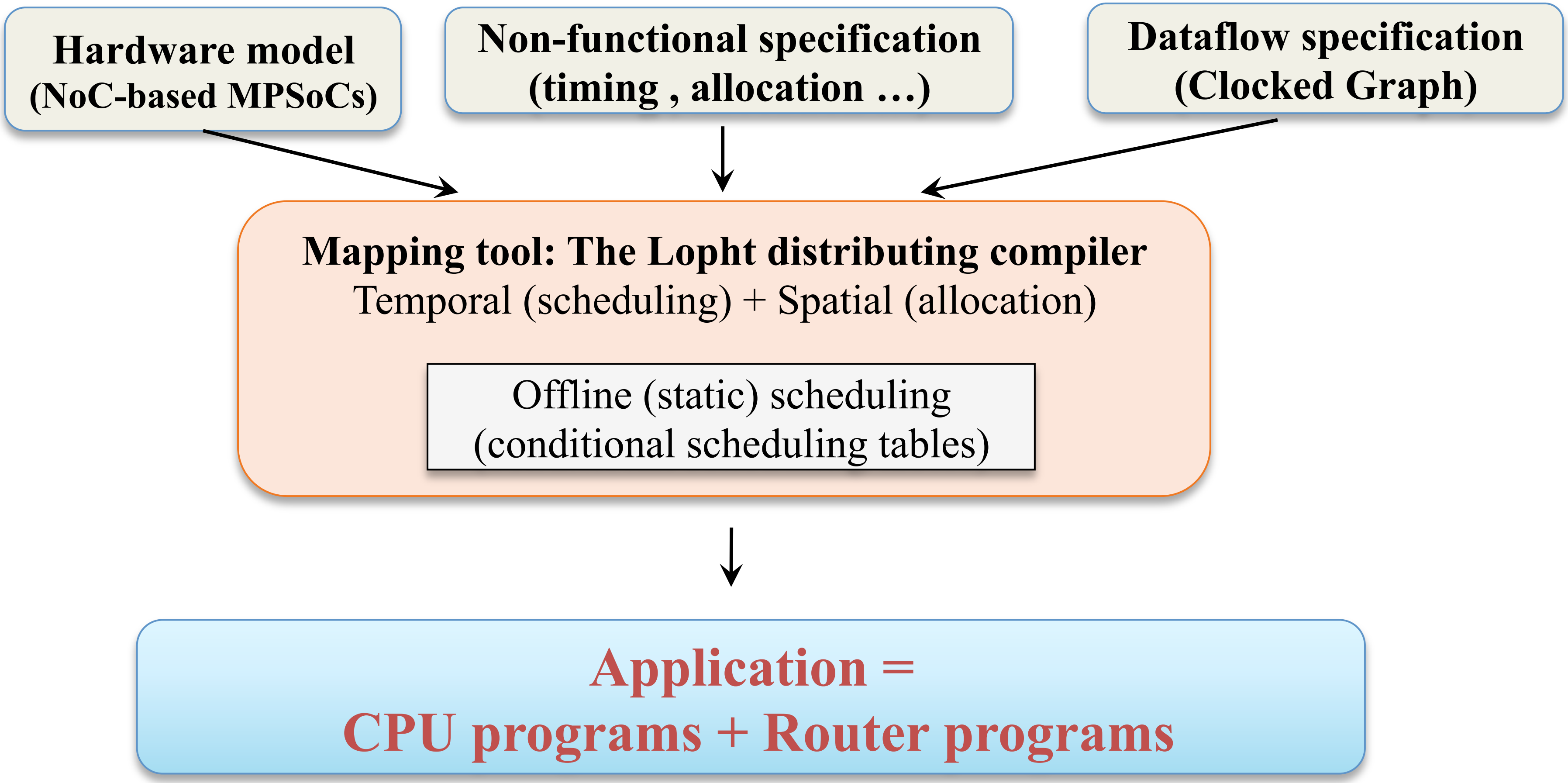
CPU/tile	Non-programmed	Programmed	Loss
Data Size of FFT (2^{14})			
1	98.51%	97.47%	1.04%
2	97.36%	95.33%	2.02%
4	95.97%	92.45%	3.52%
8	94.94%	89.95%	4.98%
16	95.18%	90.10%	5.08%
Data Size of FFT (2^{15})			
1	98.58%	97.60%	0.99%
2	97.51%	95.62%	1.89%
4	95.97%	92.44%	3.53%
8	94.54%	89.14%	5.41%
16	94.01%	87.33%	6.69%
Data Size of FFT (2^{16})			
1	98.76%	97.91%	0.86%
2	97.80%	96.15%	1.65%
4	96.43%	93.38%	3.04%
8	94.81%	89.78%	5.03%
16	93.80%	86.97%	6.83%

Automatic application mapping

a.k.a. Y-chart,
platform-based
design, AAA







Hardware model (NoC-based MPSoCs)

Non-functional specification (timing , allocation …)

Dataflow specification (Clocked Graph)

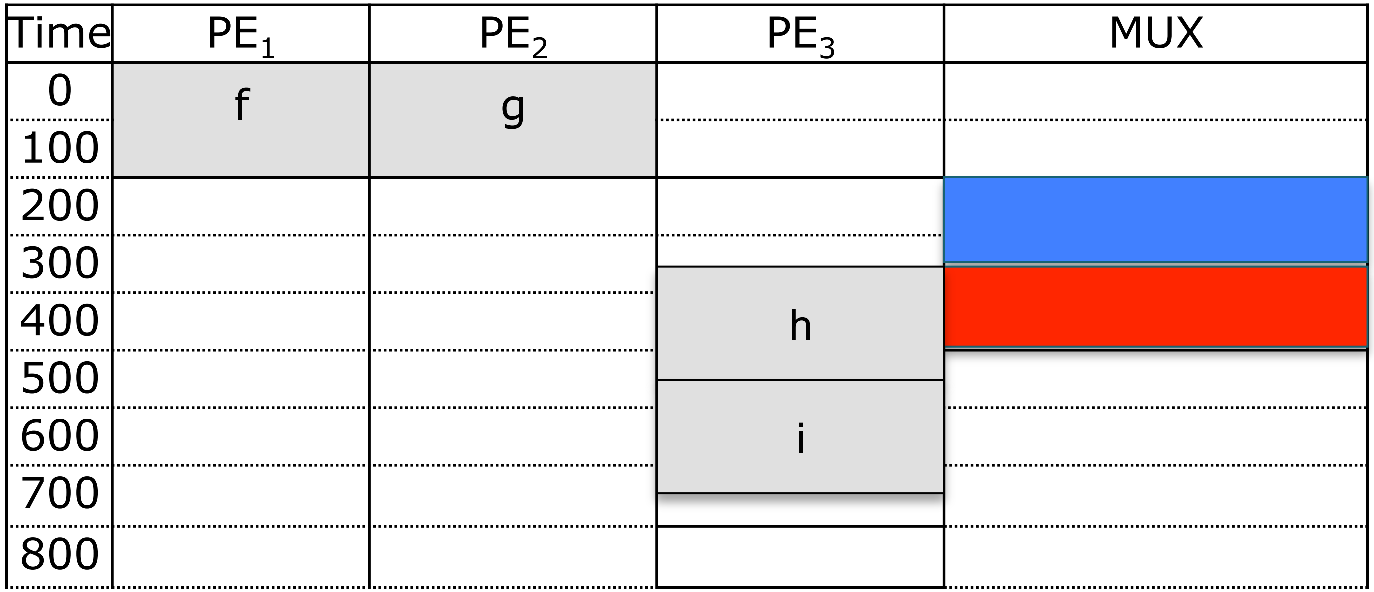
Mapping tool: The Lopht distributing compiler
Temporal (scheduling) + Spatial (allocation)

Offline (static) scheduling
(conditional scheduling tables)

Application =
CPU programs + Router programs

Automatic application mapping

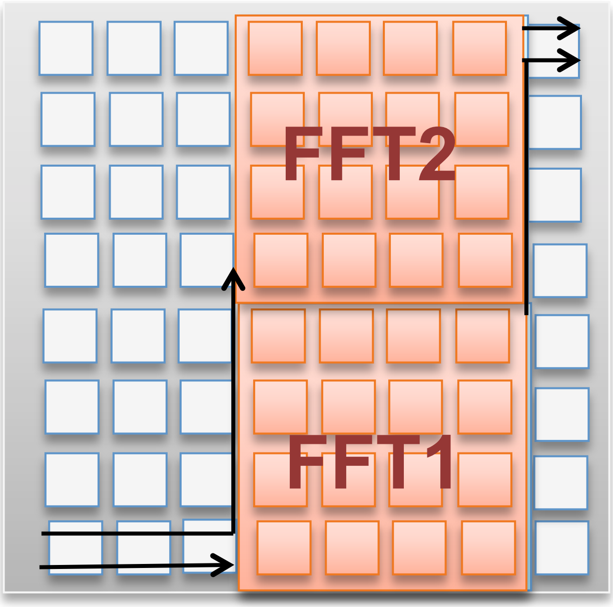
- Worst-case/Exact-case allocation
  - Real-time guarantees



  - Speed gains on very regular applications
- But: Few fully regular algorithms
  - Variable-duration operations (e.g. multiplications)
  - Fine-grain control
  - Intrinsically dynamic algorithms (ray tracing, sparse representations, etc.)

Automatic application mapping

- So, what you would need is something mixt: predefined patterns + priorities



Lopht distributing compiler : input specification

- **MPSoC hardware model:**
  - Computation resources: MPSoC tile = one CPU (x,y)
  - Communication resources: Command NoC router output ports (MUXes)
- **Dataflow : Clocked Graphs** [Potop et al. EMSOFT'09]
  - Non-hierarchic synchronous dataflow
  - Separation between dataflow and control
    - Clocks on blocks and dataflow arcs replace classical blocks such as *when*, *current*, *condact*
- **Non-functional constraints**
  - Allocation constraints ("function f must be executed on either CPU1 or CPU2")
  - Durations of computations ("executing f on CPU1 takes 10000 cycles in the worst case")

Lopht distributing compiler : scheduling heuristic

- **List scheduling on dataflow blocks, optimize allocation and scheduling for each block**
- **Communication scheduling**
  - Communication path = set of MUXes along the path, reserved together for a transmission
  - Paths are reserved as a whole → packets cannot be blocked in the NoC.
- **Pipelined scheduling, to improve throughput**
  - Successive execution cycles can overlap
- **Generated code for CPUs and Routers:**
  - Communicating sequential processes: One sequential program per CPU or router output port
  - Locality: All computations of a dataflow block are done on local tile data
  - Inter-tile data transfers are realized by the producer
  - Synchronization through locks and active wait mechanisms

LoPhT: currrent status

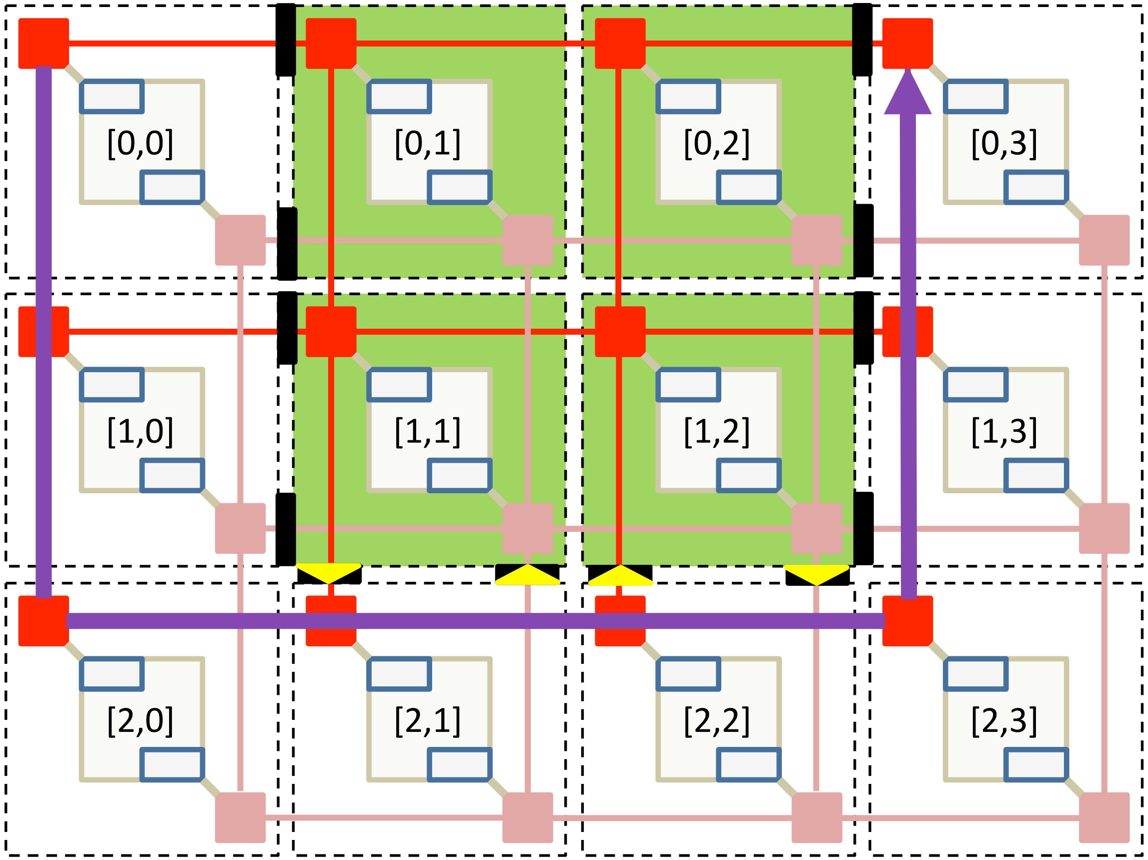
- Works on small examples and on an embedded application model (CyCab)
- Work in progress to add:
  - Multiple CPUs per tile
    - more memory banks
    - memory allocation on these banks ensuring absence of temporal interference
  - Regular applications (A. Cohen)
  - WCET of parallel code (I. Puaut, Rennes)
- Case studies needed

Conclusion

- First, what we started from
  - 4 CPUs/tile
- Then, what the archi looks like
  - Caches: write-back or prefetch engine
  - RAM: multi-bank
  - Sync: predictible, no interrupts (cost of one uncached RAM access)

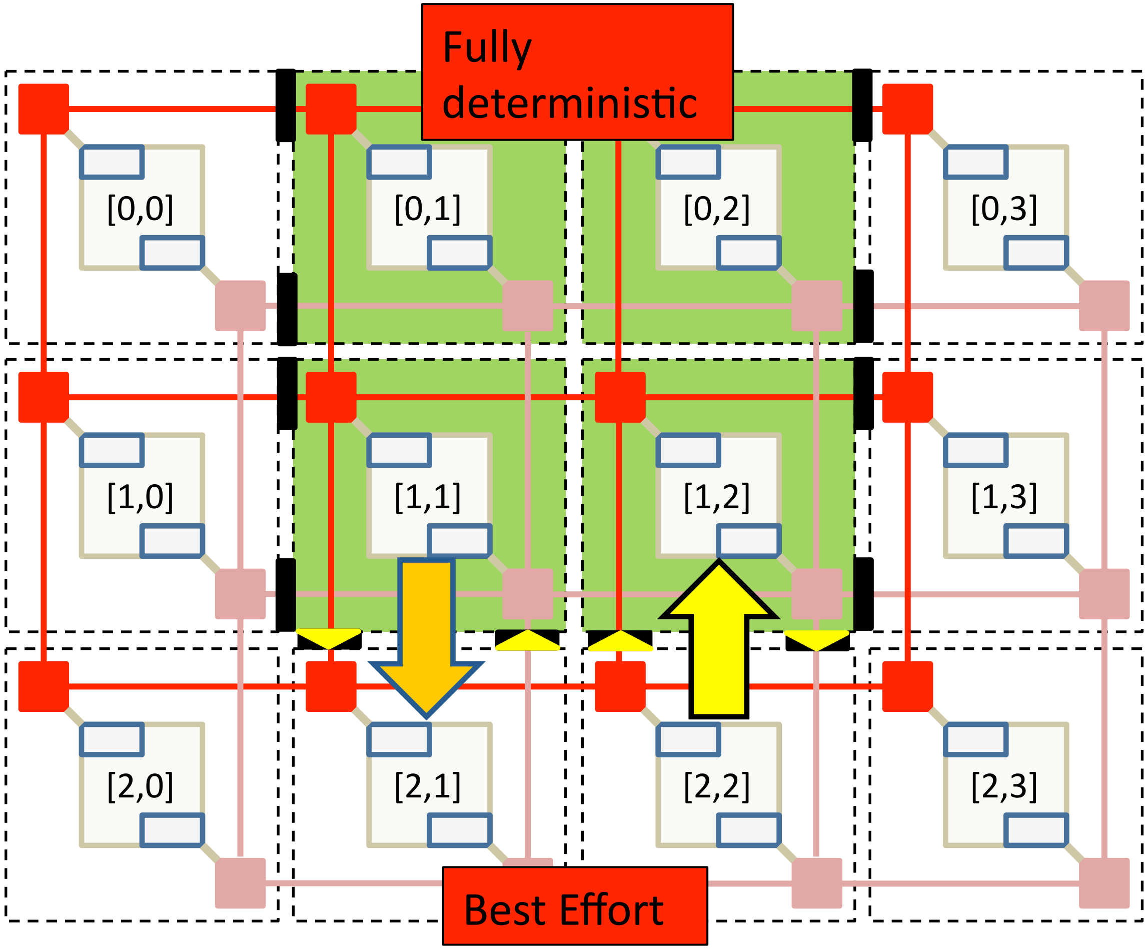
Field Programmable Tile Array

NoC avec routeurs programmables, tables de routage



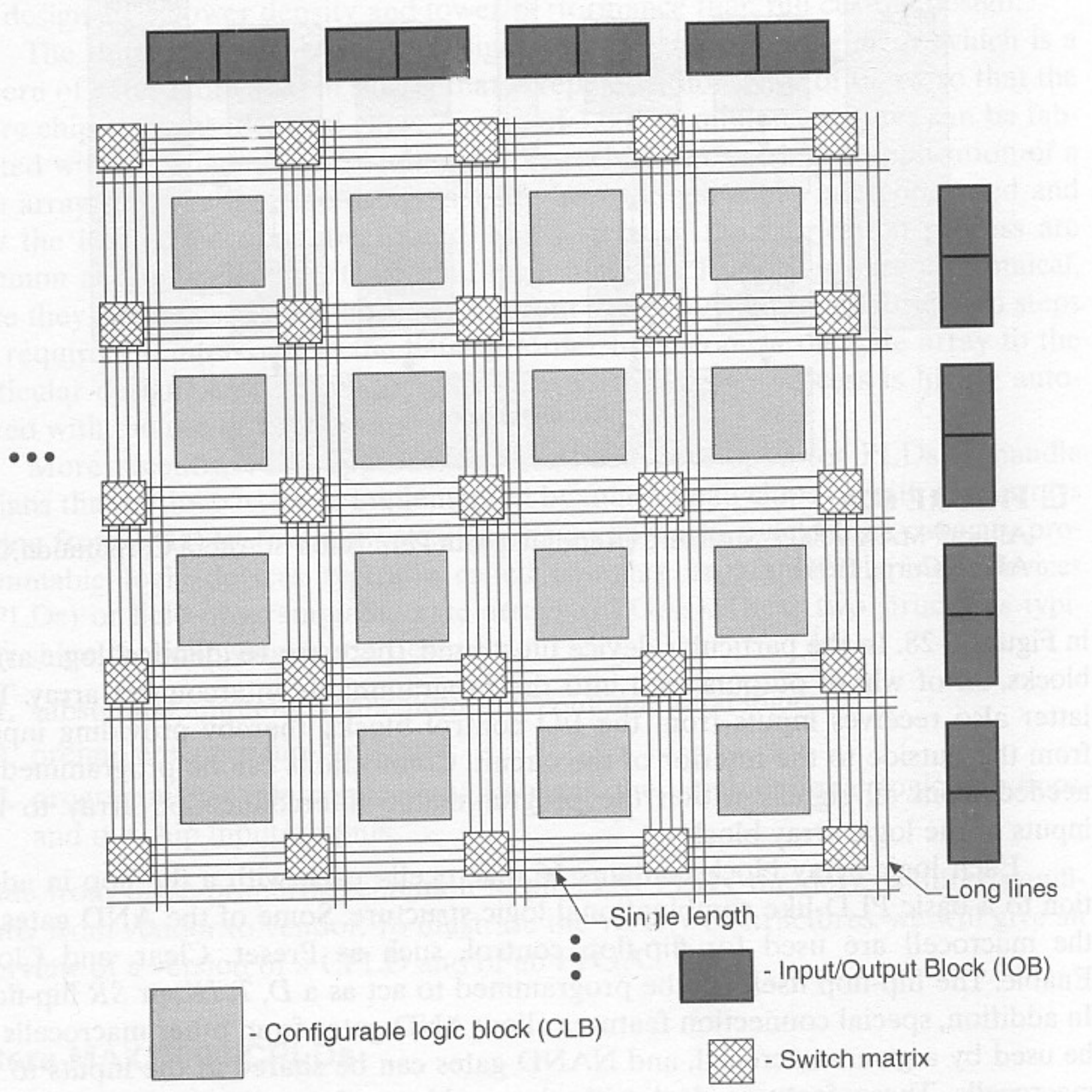
- Les routeurs, déjà modifiés pour la robustesse, deviennent programmables
  - Sortie N,S,E,W, L en fct de la table de routage programmée localement
  - Contours quelconques
  - Confinement précis des ressources défectueuses

NoC avec multiplexeurs de sortie programmables : déterminisme fin



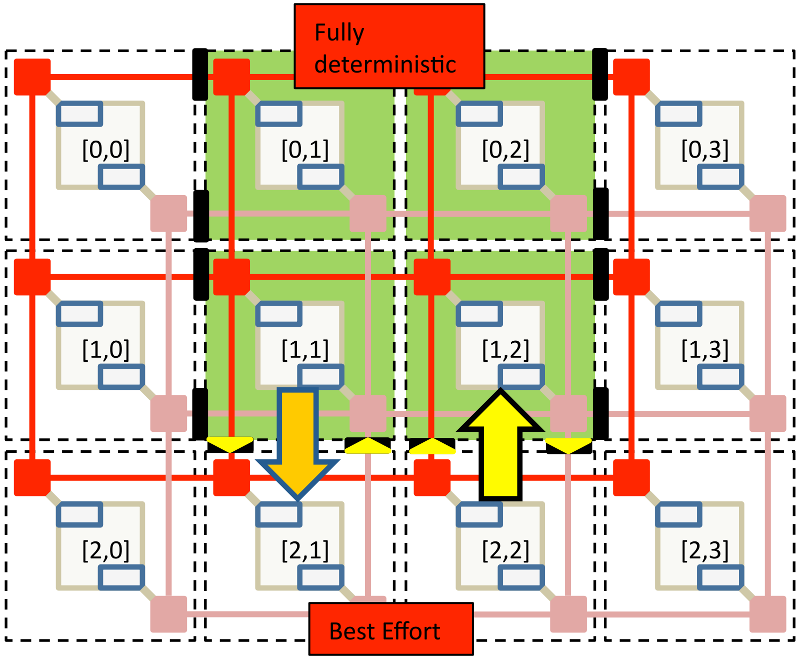
- Le NoC prend encore plus d'importance en permettant de réguler les échanges (NoC-Centric)
- **Ilôts fonctionnels** autonomes déterministes, IPs(x,y)

FPGA



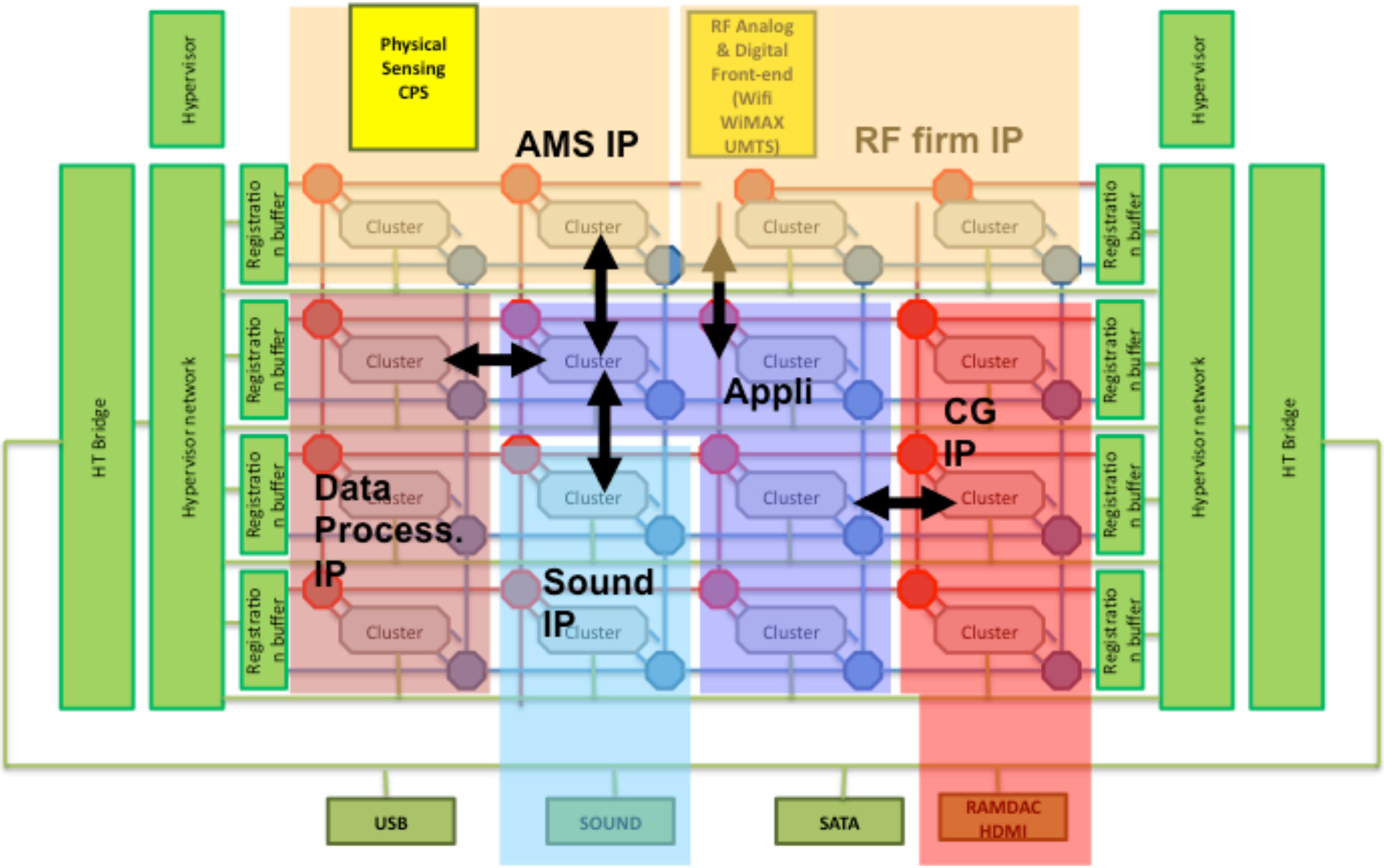
□ **FIGURE 6-29**
Xilinx® XC4000™ FPGA Structure (Adapted with Permission of Xilinx, Inc.)

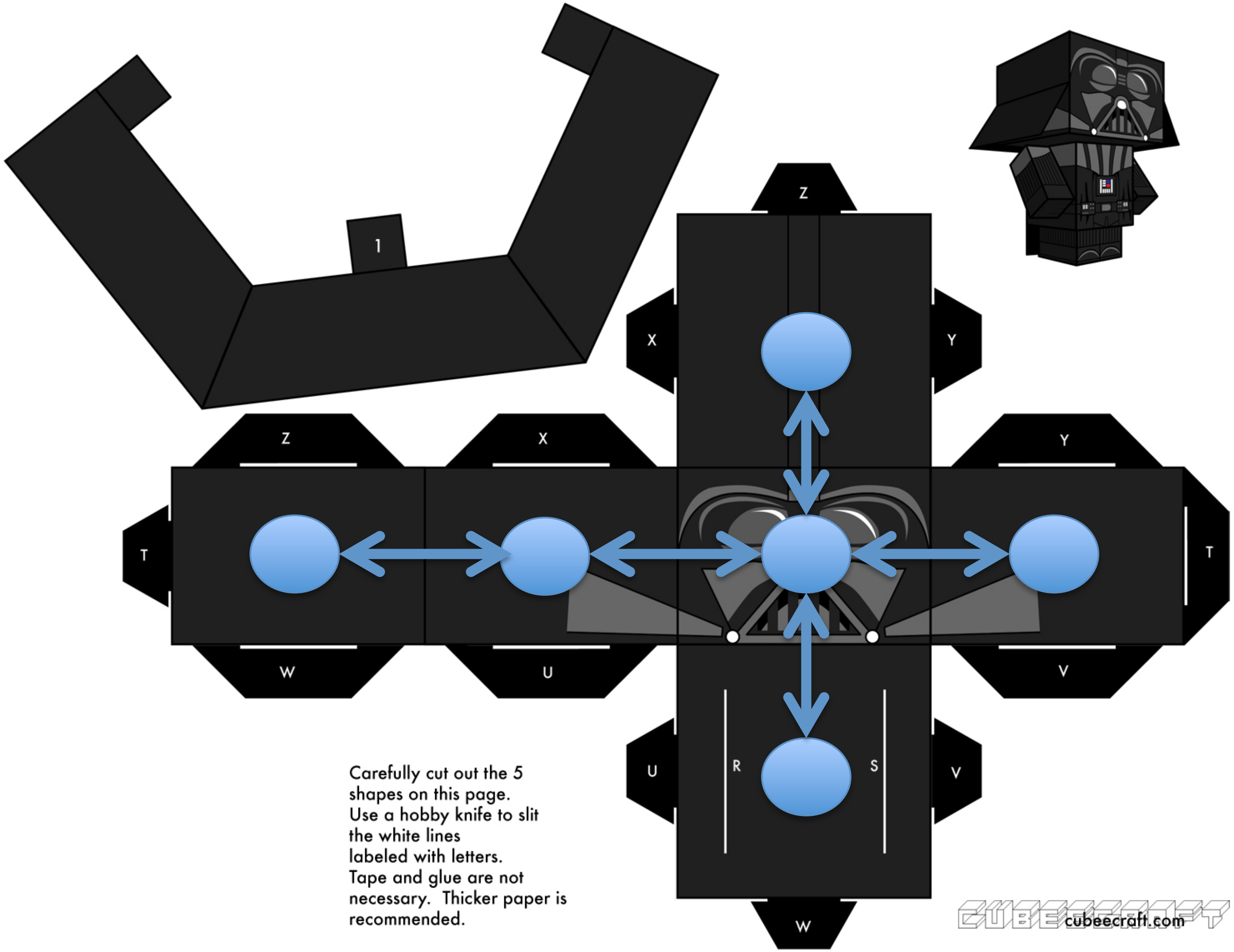
FPTA : Field Programmable Tile Array



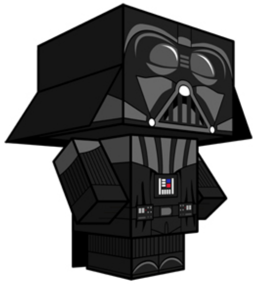
- Modèle de programmation/d’utilisation calqué sur celui des FPGA, VBA
- Notion de sous-graphe d’application préplacé avec communications préprogrammées (Soft-Ips), avec facteur de forme et référenceXY
- Fonctionnement nominal garanti
- Ips réalisées par des spécialistes
- Exemples d’Ips :
    - Pipeline rendu 3D,
    - Son 2.1, 5.1, 7.1
    - Moteur physique
    - Macro-processeur
    - Macro-mémoire

Déterminisme	**FPGA**	**FPTA**
Best Effort	Synthèse logique	Placement des threads par l’OS
Fully deterministic	Software IPs	Placement des threads à la main dans le facteur de forme, programmation des routeurs


Hypervisor
Physical Sensing CPS
AMS IP
RF Analog & Digital Front-end (Wifi WiMAX UMTS)
RF firm IP
Hypervisor
HT Bridge
Hypervisor network
Registration buffer
Cluster
Appli
CG IP
Data Process. IP
Sound IP
USB
SOUND
SATA
RAMDAC HDMI



Carefully cut out the 5
shapes on this page.
Use a hobby knife to slit
the white lines
labeled with letters.
Tape and glue are not
necessary. Thicker paper is
recommended.



cubeecraft.com