# Localization of Damaged Resources in NoC Based Shared-Memory MP2SOC, using a Distributed Cooperative Configuration Infrastructure

Zhen Zhang*, Dimitri Refauvelet*, Alain Greiner*, Mounir Benabdenbi† and François Pecheux*
* *University Pierre et Marie Curie, LIP6-SoC laboratory, 4 place Jussieu, 75252 Paris, France*
{*zhen.zhang, dimitri.refauvelet, alain.greiner, francois.pecheux*}*@lip6.fr*
† *TIMA laboratory (Grenoble INP, CNRS, UJF), 46 avenue Felix Viallet, 38000 Grenoble, France*
*mounir.benabdenbi@imag.fr*

***Abstract*—In this paper, we present a software approach for localization of faulty components in a 2D-mesh Network-on-Chip, targeting fault tolerance in a shared memory MP2SoC architecture. We use a pre-existing and distributed hardware infrastructure supporting self-test and de-activation of the faulty components (routers and communication channels), that are transformed into "black hole". We detail the software method used to localize these "black holes", and centralize the information in a single point, where a modified global routing function can be defined. This embedded software makes an extensive use of a distributed fault-tolerant configuration firmware assisted by a Distributed Cooperative Configuration Infrastructure (DCCI), that is also presented. Finally, "black hole" detection and localization coverage is evaluated.**

## I. Introduction

According to the industrial forecast on high-performance computing issues [1], Network-on-Chip (NoC) based, shared memory, Massively Parallel Multi-Processor System-on-Chips (MP2SoCs) architecture, will soon be implemented in a single chip. However, with a high permanent failure rate [2] caused by poor manufacturing yields or aging defaults, fault-tolerance is a crucial issue that must be considered at a very early stage in the design.

As future MP2SoC architectures will contain a large number of replicated identical components, a simple fault-tolerant approach is to disable faulty components (such as a processor core, or an embedded memory bank), once their erratic behaviors have been detected, and to remap the embedded software application on the remaining operational hardware. Unfortunately, for the NoC itself, this approach is far from being sufficient. In order to save silicon area, and to minimize the network latency, most NoCs use dedicated routing algorithms, taking advantage of the regular micro-network topology. If a single component (a router or a communication channel) is faulty, the micro-network topology is modified and becomes irregular. If a packet is routed toward the faulty component, in the worst case, the whole NoC is blocked. Thus, the global routing function, and the NoC itself must be reconfigured to support the new topology.

In two previous works [3], [4], we presented a self-testable&cleanable, reconfigurable 2D-mesh NoC. The two key features are summarized below:

1) ***Self-testable&cleanable*** [4]: A fully distributed & decentralized hardware built-in self-test (BIST) mechanism is integrated into the NoC. At power-on or system reboot, all NoC components are tested in isolation and in parallel. Each component that is found to be fault-free is enabled. All faulty components are disabled to prevent any fault propagation: A disabled component is configured to behave as a ***black hole***, that discards any incoming packet, and produces no outgoing packet.
2) ***Reconfigurable*** [3]: A reconfigurable, dead-lock free routing function (based on the X-First routing algorithm) has been defined. In each router, the reference X-First routing function can be modified through dedicated addressable reconfiguration registers (4 bits per router), to route the packets around the faulty components and bypass the black holes.

In conclusion, the distributed hardware BIST mechanism solves the problem of detection and de-activation of the faulty components in the NoC. Moreover, when the faulty components are localized, we have a general method to define a modified global routing function, and the NoC itself contains all the reconfiguration registers to implement the modified routing function.

But the hardware test and deactivation procedure is fully decentralized, and there is no centralization of the information about the localization of the faulty components. Therefore, we still have two problems to solve:

- ***Faulty components localization*** We need to centralize the information regarding the localization of the black holes (routers and channels), to be able to compute the global modified routing function.
- ***NoC reconfiguration*** We need a configuration bus to distribute the modified routing function in the reconfiguration registers of the fault-free routers.

In this paper we present a fully software approach for the localization problem. The only hypothesis is to have in each cluster (a cluster is a node in the 2D-mesh) a programmable processor and a ROM containing the fault tolerant configuration firmware. This configuration firmware is part of the Distributed Cooperative Configuration Infrastructure (DCCI), described in this paper.

After this introduction, section II presents the related work. Section III describes the generic 2D-mesh NoC based shared memory MP2SoC architecture, i.e our reference architecture. Section IV explains how the distributed configuration infrastructure is dynamically mapped onto this architecture on

reset. Section V details the black hole localization procedure. And section VI presents experimental results for a mesh of dimension 4×4 that proves the effectiveness of the proposed approach.

## II. Related Work

The black hole model (proposed in [4]) is actually a functional fault model where the faulty components can be detected by means of a dedicated BIST approach, and deactivated prior any localization. Several papers present solutions for localization, [5], [6], [7], which rely on the use of ATE (Automatic Test Equipment) and TAM (Test Access Mechanism), to feed NoC inputs with external packets as the test vectors, and to analyze NoC outputs. Such approaches allow to test any deterministic end-to-end path (from an input to an output, defined by a deterministic routing algorithm such as X-First). The faulty components are identified by set intersection of faulty & fault-free paths, using an exclusive method.

In our 2D-mesh topology, any end-to-end path links a processor in a source cluster to a physical memory bank in a target cluster (a cluster is a node in the 2D-mesh). In a shared memory architecture, where any processor can address any memory location, such path can be tested by a simple software transaction, i.e a software task in the source cluster reads a data word mapped in the target cluster.

As we want to support "on the field" reconfiguration, we don't want to use an external ATE, and we propose an embedded and distributed software approach to detect the black holes.

It should be noted that our proposed strategy is different with the solution [8] that proposed a mechanism for discovering the faultless paths between an I/O port and the fault-free cores in a MP2SoC. This centralized discovering process is piloted by the smart I/O port, that is a critical resource. But, our proposed solution is fully distributed, it's achieved by an faultless embedded processor core, profiting the hardware redundancy of the MP2SoC architecture.

## III. A NoC-based, shared memory, MP2SoC architecture

As presented in Section I, in our previous works [4], [3], we designed and implemented a self-testable&cleanable, reconfigurable 2D-mesh micro-network DSPIN (Distributed Scalable Predictable Interconnect Network. The original DSPIN [9], [10] was designed by the LIP6 laboratory and was physically implemented by ST Microelectronics, to support MP2SoC architecture). In this paper, DSPIN is the self-testable&cleanable, reconfigurable version.

As shown in Fig.1.A, a DSPIN-based MP2SoC architecture is composed of a set of tiles called clusters.

As shown in Fig.1.B, a cluster may contain one or several processors (with their associated instruction and data caches), a local interconnect, an embedded RAM, an embedded ROM (for configuration firmware) and two routers (In order to avoid deadlocks in command/response traffic, each cluster contains two independent routers implementing two separated sub-networks for commands and responses). In addition, some special clusters contain I/O ports controllers, used to access external mass storage devices. To each processor is associated a timeout mechanism: when it executes a memory load/store operation, the timeout mechanism is triggered. In the event the memory operation fails, the timeout generates an interrupt and the processor enters its exception mode.

Fig.1.C details the generic DSPIN router, that contains 5 ports (North, East, South, West & Local) interlinked as a full 5×5 crossbar. Each port contains two fifos, one for input and one for output. An input fifo in a given router, and the output fifo in the neighbor router define a point-to-point communication channel. In the following, an half-path (HP) is defined as the enumerated set of channels and routers involved in the carrying of a command packet (resp. response packet) from the initiator cluster (resp. target) to the target cluster (resp. initiator) through the NoC. A path (P) is the concatenation of two half-paths, one for the command (HPC) and one for the response (HPR).

After power-on or system reboot, and thanks to the hardware test & initialization mechanism, the fault-free components are enabled. The faulty ones are disabled and configured as black holes. The DSPIN NoC is not only cleaned from any evil failure propagation, but the fault-free routers are configured to implement the reference X-First routing function. Therefore, the NoC supports local communications, between a cluster and its neighbors clusters, as long as the corresponding communication channel is fault-free.

Finally, when there is more than one (fault-free) processor in a cluster at the end of the local BIST procedure, a local master is elected and can run the configuration firmware that is stored in the embedded ROM of each cluster.

## IV. Distributed Cooperative Configuration Infrastructure (DCCI)

During the classical (software) initialization of a system, the boot code that performs various tests and configuration tasks is generally located in a unique ROM, even in the case of a multi-cores architecture. Our configuration firmware (called CF in the following) is similar to the BIOS in a multi-cores PC, but in a possibly damaged MP2SoC, the hardware resources can not be trusted anymore, and chip initialization takes place in an uncertain world where a processor, a memory bank, a network interface controller, a router, or the boot ROM itself may be defective.

### A. DCCI general principle

To remove this uncertainty, the key idea is to have one CF per cluster, to support a fully distributed approach, where a cluster is able to communicate and exchange information with its neighbor CFs, resulting in a Distributed Cooperative Configuration Infrastructure (DCCI).

During initialization, the role of the DCCI is to progressively build - only relying on local communications between neighbor clusters - a trusted tree of operational clusters, where

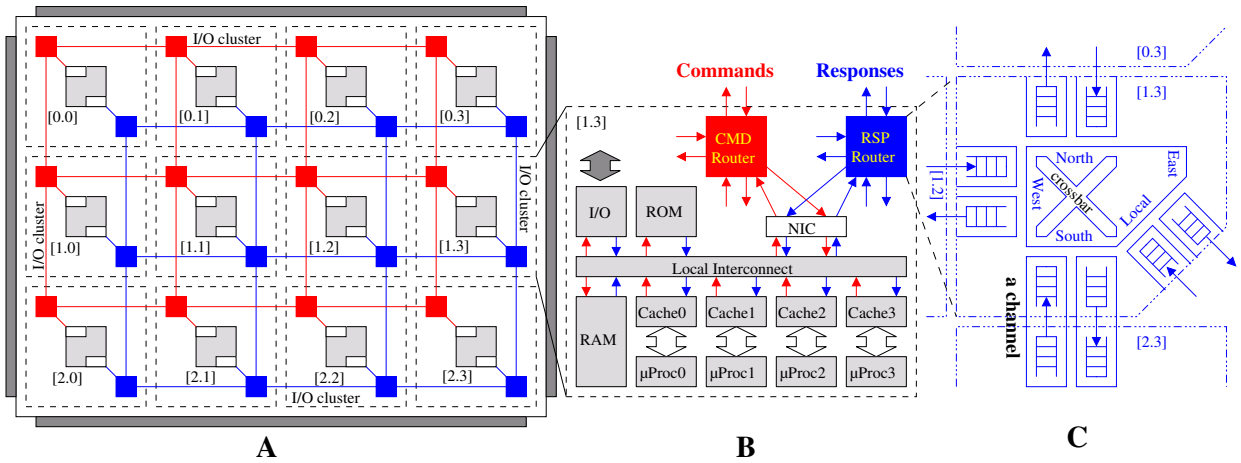


Figure 1. A typical 2D-mesh based, shared memory, MP2SoC architecture.

each operational cluster contains one operational processor running the CF. This tree is built in a bottom-up way, starting with operational clusters as leaves. This communication tree uses a limited part of the routing capabilities of the NoC (that has not yet been fully configured). It uses only the local communication channels between neighbor clusters for software based CF to CF communication through dedicated mailboxes.

The root of tree is a cluster determined by a distributed election process. The most important criterion in this election process is the capability of this root cluster to access the external mass storage where more exhaustive test programs and the final operating system itself are available, and can be loaded in the embedded RAM of the root cluster.

This software based communication tree can thus be seen as a slow and temporary communication infrastructure, dynamically constructed during the boot stage, using the NoC resources that have been identified as fault-free by the hardware BIST.

As soon as this tree is constructed, and the root is elected, this very unique master processor can access the external mass storage containing a virtually unlimited software stack outside the chip.

The master processor can use this communication tree to make any processor in the tree execute any specific software task (debug, fine-grain test), to propagate any configuration command to any child tree node, or read any status information. It can be used to complete the MP2SoC reconfiguration, and especially the configuration of the NoC itself.

### B. Cluster self-test

After hardware reset, each cluster has to test itself by executing its local CF code, before it can try to participate in the distributed procedure to build the DCCI communication tree.

This local test is a 3 stages process:

1) ***local intra-cluster test:*** It is a first, coarse grain, functional software based test (such as presented in [11]) for all IPs belonging to the cluster, if a cluster is usable to participate to the tree building procedure. For instance, a cluster which RAM does not pass its coarse march test is declared unusable.
2) ***local leader election:*** Second, an operational processor of the cluster is locally elected (the one with the smallest processor identifier). The elected processor represents the cluster with respect to the surrounding clusters. The other operational processors are put in idle state.
3) ***access to the external memory:*** The locally elected processor tries to establish a connection with the nearest external I/O controller, using the default configuration for the NoC infrastructure (standard X-first routing algorithm).

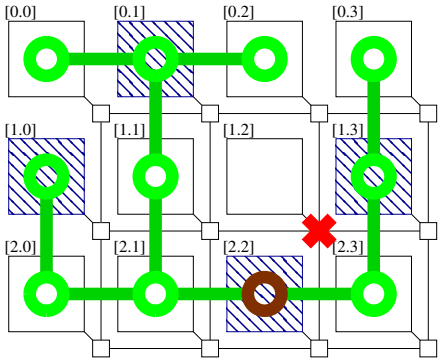


Figure 2. DCCI example in a damaged MP2SoC. At coordinates [1.2], at least one router (cmd or rsp) is faulty. A spanning DCCI communication tree is built as a result of each local CF task communicating only with its neighboring clusters. The tree node is presented as a circle, the tree root is at coordinates [2,2].

If a cluster pass successfully the two first steps, it is declared to be usable and the locally elected processor executes a specific CF code to discover its environment. Additionally, if the third test pass successfully, the cluster is a potential leader.

### C. Spanning tree building

As stated before, the idea is to build a spanning tree (as shown in Fig.2) covering all connex clusters declared as usable. Each potential leader is a possible candidate to become the root, and the active processor in the elected cluster

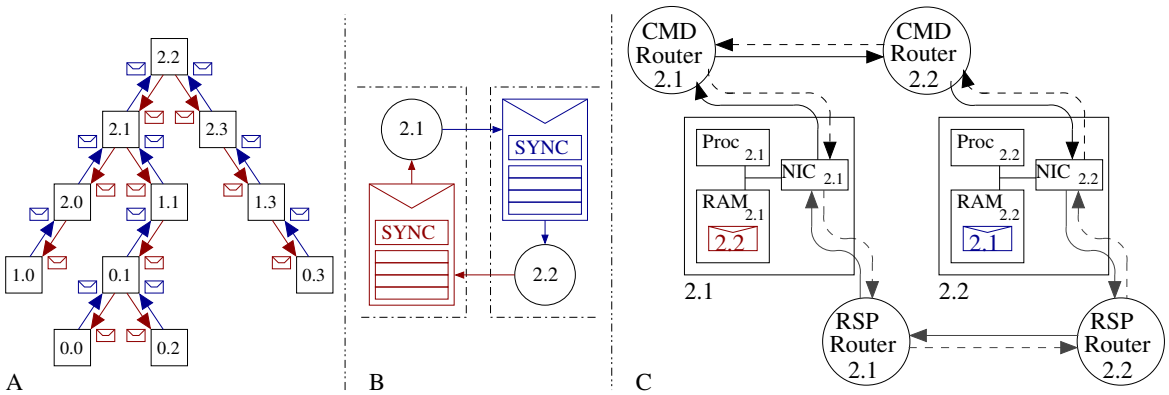


Figure 3. Correspondence between tree edges and hardware paths.

becomes the "chip leader". Fig.2 shows an example of a DCCI tree on a damaged MP2SoC (at least one router at coordinates [1.2] is faulty).

The spanning tree is built by concurrent aggregation of sub-trees. To manage concurrence between sub-trees, preference is given to clusters that are located nearby an access port to the external mass storage.

Fig.3.A presents a global representation of the final DCCI tree. To be connected, two neighbor clusters (such as clusters at coordinates [2.1] and [2.2]) must be able to communicate through a bidirectional link composed of two directed edges, for full duplex communication. Each edge corresponds to a software mailbox, as shown in Fig.3.B. In the followings, an edge from node/cluster A to node/cluster B is named $e_{AB}$.

A mailbox is a data structure in memory composed of a buffer for the message, and a synchronization flip-flop element. In the DCCI each mailbox is located in the local memory of the receiving node, and is shared by exactly one sender and one receiver. Consequently, two mailboxes are used for bidirectional communications.

Fig.3.C describes the whole set of hardware components involved in the bidirectional communication between two neighbor nodes using two mailboxes. As shown in the picture, each bidirectional communication channel uses 12 hardware channels and 4 routers. Each hardware element (channel or router) must be faultless. After the self-test, each cluster starts an handshake procedure to produce the list of neighbors clusters with which the mailbox communication is possible.

Considering two neighbors nodes A and B, this handshake algorithm performs the followings operations:

- Node A sends a PING message to node B using edge $e_{AB}$. This software transaction implies the use of $HPC_{AB}$ (Half Path Command) and $HPR_{AB}$ (Half Path Response).
- If node B receives the PING message it sends back an ALIVE message to node A using $e_{BA}$. This software transaction implies the use of $HPC_{BA}$ and $HPR_{BA}$.
- If A receives a timeout for its PING message, and still receives the ALIVE response message, it means that $HPR_{AB}$ is faulty. Thus, A sends a NOK message to B to inform it that $HPR_{AB}$ is down.

The protocol is fully symmetric. If a timeout or a NOK message is received by a node, the two involved edges are declared has faulty. If the two paths between A and B are fault-free, the A-B connection can be used for tree construction and can potentially become active edges of the DCCI tree.

The characteristics of the resulting tree are the following:

- each node has been tested and is usable
- each edge of the tree has been tested and is usable
- there exists no edges between nodes that are not direct neighbors
- any tree path between any couple of clusters belonging to the tree is usable

### D. Using the spanning tree

Like in [12], any message between 2 distant nodes N0 & N1 is propagated with the active cooperation of all intermediate nodes on the path between N0 & N1. Each intermediate node is actually acting as a software router. Therefore, and unlike [12], which uses probabilistic broadcast for fault tolerant communication, the approach presented herein is purely deterministic.

We previously indicated that a tree can be considered as a trusted launching pad for exploration and testing. By using end-to-end protocol or flooding protocol over the tree, we can send command, application test, or test results to one or to all nodes. For example, in the case of a more exhaustive NoC test application, we can dispatch the test application from the root to all tree nodes with flood message, execute application on every nodes, and retrieve results to the root.

## V. The "Black Hole" Localization Procedure

### A. General Principle

In a shared memory architecture, any path in the NoC links a processor (initiator cluster) to a physical memory bank (target cluster). The multi-threaded, distributed, software application for black hole localization is loaded by the master processor on all usable clusters, using the DCCI communication tree. All enabled NoC routers are initially configured to implement the reference X-First routing algorithm. The software application therefore tries to use the NoC routing infrastructure for inter-thread communications, but some packets may be lost in the black holes.

By collecting the informations stored in each cluster on successful and unsuccessful transactions (through the DCCI communication tree), the master processor (root of tree) is able to localize the black holes.

It should be noted that, in (DCCI) tree creation, some paths have been tested (the path between two neighboring clusters). The tree topology implicitly already contains informations of black hole locations, but these informations are very rough, so we must use a dedicated application of black hole localization to obtain most fine-grain informations.

### B. Distributed Algorithm

A task (t), running in a cluster[y.x] (coordinates in the 2D-Mesh) makes a read transaction targeting a cluster [y'.x']. According to the X-First routing policy, a path P between two clusters define a unique set of routers and channels. If the transaction succeeds (both the command packet and the response packet), the task (t) receives the expected data, and registers the path (y.x/y'.x') as OK. If not, (t) receives a timeout interrupt, indicating a packet loss. Once all paths from cluster[y.x] to all other clusters have been tested, a list of FaultLess Paths (FLP) can be constructed. Therefore, two sets of FaultLess Routers FLR(y,x) and FaultLess Channels FLC(y,x) can be derived from the FaultLess Paths list (FLP).

$PathRegistration()$ :
**Require:** $[Y.X]$ is the index of the source (the current cluster).
**Require:** $FLP$ is the set of faultless paths seen by the source.
**Require:** $FLC$ is the set of faultless channels seen by the source.
**Require:** $FLR$ is the set of faultless routers seen by the source.
{In the following, $CHC/R_{YX\_N/S/E/W/L\_I/O}$ represents a channel. C/R means cmd or rsp subnetwork, YX means that the channel belongs to router[Y.X], N/S/E/W/L means the port that the channel belongs to, and I/O means input or output. The same, $RTC/R_{YX}$ represents a router. C/R means cmd or rsp subnetwork, YX means the index of router. For example, $CHC_{YX\_L\_I}$ means the input channel of local port of cmd router [Y.X]. And $RTC_{YX}$ means the cmd router [Y.X].}
1: Construction of $FLP$
2: $FLC \leftarrow NIL$
3: $FLR \leftarrow NIL$
4: **for** $i = 0$ to $|FLP|$ **do**
5: $[y.x] \leftarrow FLP[i]$
{Extract the target cluster index of a faultless path.}
6: $FLC \leftarrow FLC \bigcup CHC_{YX\_L\_I} \bigcup CHC_{yx\_L\_O} \bigcup CHR_{yx\_L\_I} \bigcup CHR_{YX\_L\_O}$
7: $FLR \leftarrow FLR \bigcup RTC_{YX} \bigcup RTC_{yx} \bigcup RTR_{yx} \bigcup RTR_{YX}$
8: **if** $x > X$ **then**
9: **for** $i = X$ to $x - 1$ **do**
10: $FLC \leftarrow FLC \bigcup CHC_{Yi\_E\_O} \bigcup CHR_{yi\_E\_I}$
11: $FLR \leftarrow FLR \bigcup RTC_{Y(i+1)} \bigcup RTR_{yi}$
12: **end for**
13: **else if** $x < X$ **then**
14: **for** $i = X$ to $x + 1$ **do**
15: $FLC \leftarrow FLC \bigcup CHC_{Yi\_W\_O} \bigcup CHR_{yi\_W\_I}$
16: $FLR \leftarrow FLR \bigcup RTC_{Y(i-1)} \bigcup RTR_{yi}$
17: **end for**
18: **end if**
19: **if** $y > Y$ **then**
20: **for** $j = Y$ to $y - 1$ **do**
21: $FLC \leftarrow FLC \bigcup CHC_{jx\_S\_O} \bigcup CHR_{jX\_S\_I}$
22: $FLR \leftarrow FLR \bigcup RTC_{jx} \bigcup RTR_{(j+1)X}$
23: **end for**
24: **else if** $y < Y$ **then**
25: **for** $j = Y$ to $y + 1$ **do**
26: $FLC \leftarrow FLC \bigcup CHC_{jx\_N\_O} \bigcup CHR_{jX\_N\_I}$
27: $FLR \leftarrow FLR \bigcup RTC_{jx} \bigcup RTR_{(j-1)X}$
28: **end for**
29: **end if**
30: **end for**
31: **return** FLC
32: **return** FLR

### C. Localization and reconfiguration

The two sets FLR(y,x) and FLC(y,x) are distributed in each cluster[y.x]. These information data can be collected by the master processor, using the DCCI communication infrastructure, and merged in two global sets, GFLR and GFLC, as shown in Fig.4. Finally, according to this method, any router or communication channel that is not present in these GFLR and GFLC sets is a black hole. So, theoretically, the black hole Detection Coverage (DC) is 100%, for a defective NoC with any number of fault. And this is confirmed in the section VI by the experimental results.

It is worth noticing that the number of fault-free (usable) communication resources (routers or communication channels) identified by this procedure can be much larger than the resources used by the DCCI covering tree.

Once all the black holes are identified and localized, it is possible (if the number of faulty component is not too large) to compute a modified routing function, and to use the DCCI communication infrastructure to load this modified routing function into the addressable configuration registers distributed in the NoC.

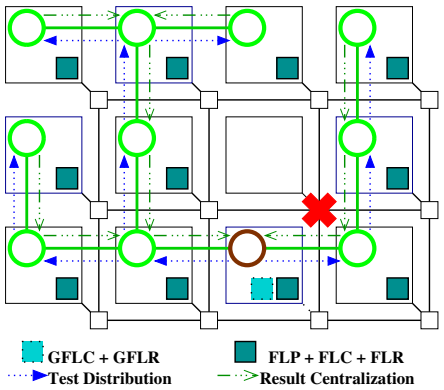

Figure 4. Thanks to the DCCI communication tree, the root can load the test code from the external memory, and to distribute the code to each tree node, to do the black hole detection. Once all of detections have been achieved, the root can centralize the FLC and FLR from each node, and to merge these lists into two global sets, GFLC and GFLR.

## VI. Experimental Results

### A. Detection Coverage

In this subsection, we present experimental results of Detection Coverage (DC) evaluation for the black hole localization procedure. We have simulated two types of fault in a $4 \times 4$ clusters MP2SoC, 1st, single fault injection (one faulty router or one faulty channel); 2nd, multi-faults injections. These fault injections were simulated on a dedicated C simulator.

*1) Single Injection:* As in a $M \times N$ DSPIN 2D-mesh, there are C cmd&rsp channels, and there are R cmd&rsp routers.
$C = (M \times (N-1) \times 2 + N \times (M-1) \times 2 + M \times N \times 2) \times 2$
$R = M \times N \times 2$

The Detection Coverage has been evaluated for all the situations where the NoC contains one single fault: one faulty router or one faulty channel defining a total of $(C+R)$ different faulty networks. In our example, with $M = 4, N = 4$, there are 160 channels and 32 routers.

In all cases, the black hole has been identified and located, resulting in a Detection Coverage of 100% for a single fault.

However, in some cases, some fault-free components are wrongly identified as black holes, which is explained in the following.

As shown in Fig.5 (a partial description of a cluster), there are some dependencies between channels. For example, channel 1 and channel 3 depends on each other, because any path between the local processor and a RAM of another cluster will contain this couple. Channel 1 is a black hole, channel 3 can not be tested with any faultless path, and it will be identified as a black hole. But this result is acceptable for the reconfigurable routing [3], since in this case the whole router is deactivated.

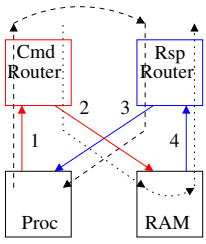


Figure 5. Dependency exists between channel 1 & channel 3 or between channel 2 & channel 4.

*2) Multi-faults injection:* The Detection Coverage of multi-faults injection has been evaluated for all the situations of

- 1 faulty router and 1 faulty channel
- 2 faulty routers
- 2 faulty channels
- 2 faulty routers and 1 faulty channel
- 1 faulty routers and 2 faulty channels
- 2 faulty routers and 2 faulty channels

All these simulations resulted in a 100% coverage.

### B. Application Execution Time Evaluation

The execution time is an important issue for a software procedure that will be executed at each reboot of the system. For this experiment we simulated the complete procedure on a 4×4 2D-mesh MP2SoC architecture containing 16 processors, modeled with the cycle-accurate SoCLib virtual prototyping platform [13]. This architecture contained one single fault. The total time is $7.1 \times 10^6$ cycles (without hardware test process):

- Time for (DCCI) tree construction: $1.9 \times 10^6$ cycles
- Time for for test task distribution: $1.2 \times 10^6$ cycles
- Time for test execution: $3.5 \times 10^6$ cycles
- Time for test result centralization: $0.5 \times 10^6$ cycles

As this procedure is executed using the system clock, we obtain 0.014 second at 500Mhz. Which is fully acceptable.

### C. Application Code Size

For a MIPS32 processor, the application code is split in: DCCI : 5 Kbytes; Test and localization procedure : 2.5 Kbytes. Embedding this application in a MP2SoC is thus affordable.

## VII. Conclusion

In this paper, we presented a software approach to localize faulty hardware components in a 2D-mesh NoC used in a shared memory MP2SoC. The localization of faulty components is mandatory to implement an "on the field" reconfiguration mechanism supporting fault tolerance in the context of permanent failures. These faulty components can either be a point-to-point communication channel, or a complete router. They can be transformed into black holes by a built-in self-test (BIST) mechanism that is the basis of our fault model. The localization algorithm relies on a Distributed Cooperative Configuration Infrastructure that dynamically builds a software based communication tree, covering all the nodes that have successfully passed the local BIST. This DCCI communication infrastructure is a distributed software mechanism that can be used as a configuration bus. It does not use the full routing capabilities of the NoC, but only the local communication channels between two neighbor nodes. The proposed black hole detection software algorithm has been evaluated in a 16 nodes MP2SoC architecture (4×4 mesh) modeled as a SystemC virtual prototype, in the framework of the cycle accurate SoCLib environment. It reaches a Detection Coverage of 100% in all tested cases. The same DCCI communication tree can be used to distribute the resulting modified routing functions to the fault-free routers.

It should be noted that, the method proposed in this paper, can be used in any shared memory multi-core architecture with a 2D-Mesh NoC.

## References


[1] International Technology Roadmap for Semiconductors. [Online]. Available: http://www.itrs.net

[2] S. Furber, "Living with Failure: Lessons from Nature?" in *Proc. of ETS'06, the 11th IEEE European Test Symposium*, 2006.

[3] Z. Zhang, A. Greiner, and S. Taktak, "A reconfigurable routing algorithm for a fault-tolerant 2D-Mesh Network-on-Chip," in *Proc. of DAC'08, the 45th Design Automation Conference*, 2008.

[4] Z. Zhang, A. Greiner, and M. Benabdenbi, "Fully Distributed Initialization Procedure for a 2D-Mesh NoC, Including Off Line BIST and Partial Deactivation of Faulty Components," in *Proc. of IOLTS'10, the 16th IEEE International On-Line Testing Symposium*, 2010.

[5] C. Grecu, P. Pande, B. Wang, A. Ivanov, and R. Saleh, "Methodologies and Algorithms for Testing Switch-Based NoC Interconnects," in *Proc. of DFT'05, the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005.

[6] K. Stewart and S. Tragoudas, "Interconnect Testing for Networks on Chips," in *Proc. of VTS'06, the 24th IEEE VLSI Test Symposium*, 2006.

[7] J. Raik, R. Ubar, and V. Govind, "Test Configurations for Diagnosing Faulty Links in NoC Switches," in *Proc. of ETS'07, the 12th IEEE European Test Symposium*, 2007.

[8] E. Kolonis, M. Nicolaidis, D. Gizopoulos, M. Psarakis, J. Collet, and P. Zajac, "Enhanced self-configurability and yield in multicore grids," in *Proc. of IOLTS'09, the 15th IEEE International On-Line Testing Symposium*, 2009.

[9] I. Panades, A. Greiner, and A. Sheibanyrad, "A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach," in *Proc. of NanoNet'06. the 1st International Conference on Nano-Networks and Workshops.*, 2006.

[10] I. Miro-Panades, F. Clermidy, P. Vivet, and A. Greiner, "Physical implementation of the dspin network-on-chip in the faust architecture," in *Proc. of NoCS'08, the 2nd ACM/IEEE International Symposium on Networks-on-Chip*, 2008.

[11] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded processor-based self-test*. Kluwer Academic Pub, 2004.

[12] T. Dumitraş, S. Kerner, and R. Mărculescu, "Towards on-chip fault-tolerant communication," in *Proc. of ASPDAC'03, the 8th Asia and South Pacific Design Automation Conference*, 2003.

[13] LIP6 et al. SoClib. [Online]. Available: https://www.soclib.fr