# A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips

Mohammad Fattah[1], Antti Airola[1], Rachata Ausavarungnirun[2], Nima Mirzaei[3],
Pasi Liljeberg[1], Juha Plosila[1], Siamak Mohammadi[3], Tapio Pahikkala[1],
Onur Mutlu[2] and Hannu Tenhunen[1,4]

[1]Department of Information Technology, University of Turku, Turku, Finland
[2]Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA
[3]School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran
[4]Department of Electronic Systems, Royal Institute of Technology– KTH, Stockholm, Sweden
{mofana, ajairo, pakrli, juplos, aatapa, hanten}@utu.fi
{rachata, onur}@cmu.edu
n.mirzaei@ut.ac.ir, smohammadi@ece.ut.ac.ir

## ABSTRACT

This paper introduces a new, practical routing algorithm, Maze-routing, to tolerate faults in network-on-chips. The algorithm is the first to provide all of the following properties at the same time: 1) fully-distributed with no centralized component, 2) guaranteed delivery (it guarantees to deliver packets when a path exists between nodes, or otherwise indicate that destination is unreachable, while being deadlock and livelock free), 3) low area cost, 4) low reconfiguration overhead upon a fault. To achieve all these properties, we propose Maze-routing, a new variant of *face routing* in on-chip networks and make use of *deflections* in routing. Our evaluations show that Maze-routing has 16X less area overhead than other algorithms that provide guaranteed delivery. Our Maze-routing algorithm is also high performance: for example, when up to 5 links are broken, it provides 50% higher saturation throughput compared to the state-of-the-art.

## Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—
*Fault tolerance*

## General Terms

Algorithms, Performance, Reliability, Design

## Keywords

Network-on-chips, permanent fault, link failure, routing algorithm, face routing, deflection routing, distributed algorithms

## 1. INTRODUCTION

Aggressive scaling of transistor feature size comes with benefits and curses. The key benefit is the ability to integrate many more computational and storage components, including many processors, cache slices, memory controllers, and specialized acceler-

ators, on the same die, which leads to greatly increased computational power in a System-on-a-Chip (SoC). To take advantage of this computational power effectively, these integrated components need to be connected to each other with an effective communication substrate. Network-on-chips (NoCs) are a promising way to interconnect on-chip components as they are shown to be more scalable than traditional bus-based interconnects [9, 14, 23, 41].

A major curse of aggressive transistor scaling is the reduced reliability of the on-chip components [8]. For example, fault mechanisms such as device wear-out caused by oxide breakdown, electromigration, and thermal cycling [8, 10], are expected to be exacerbated in future technology nodes, leading to more failed components during operation, as the SoC device ages [13]. Hence, it is critical to design both the components and the interconnect to operate in the presence of faulty components.[1]

While a faulty computational or storage component (e.g., a core or a cache slice) degrades SoC performance, a fault in a network-on-chip component (e.g., a router or a link) can be even more serious as the NoC provides the communication substrate between multiple components. Such a fault can potentially cripple system performance and perhaps even more severely become a single point of failure [19]. It is projected, therefore, that NoCs need to tolerate at least tens of randomly-located failures [32], in order to keep up with fault-tolerant processor designs. A (large) number of randomly-located failures can easily convert a regular network topology (with a simple routing algorithm) into a complicated maze of routers connected by links with some connections (i.e., routers or links) missing [3]. In such an SoC with some faulty NoC components, it is therefore critically important to have an efficient routing algorithm that can deliver packets to destinations through the available links and routers (or otherwise find out the destination is unreachable), in order to guarantee continued operation.

We posit that a practical, effective and efficient routing algorithm that can tolerate NoC faults should satisfy the following properties. First, *guaranteed delivery*: it should guarantee to deliver a packet to its destination when a path between the source and the destination exists or else indicate that the destination is unreachable, regardless of the number and location of faults. We also call this property *full (fault) coverage*. Second, it should be *fully distributed*: it should have no centralized component in finding non-faulty paths through the network to deliver a packet because a centralized component is not only a scalability bottleneck but also can constitute a single

---

[1]Faults can be permanent, transient and intermittent [24]. While we do not target a particular category of faults in this work, our discussion is focused more on permanent faults, i.e., faults that stay after they appear until the faulty components are repaired.

point of failure in a fault-tolerant system. Third, it should have *low hardware area cost*, as any additional area overhead increases not only implementation cost but also vulnerability of the SoC to faults. Finally, it should have *low reconfiguration overhead*: when a new faulty component is identified, the network should continue normal operation and adapt the routing mechanisms without affecting the system and applications running on it. We find that, while some past fault-tolerant routing algorithm designs satisfy one, two or at most three of these goals, no previous work provides an algorithm that satisfies all four requirements.

Our goal in this work is to devise a new, practical fault-tolerant routing algorithm that achieves all of these four goals. To this end, we develop a new algorithm called Maze-routing, taking inspiration from the idea of *face routing*, which was originally proposed for ad-hoc wireless networks [11]. Our algorithm, called Maze-routing, provides *guaranteed delivery* in a *fully-distributed manner* at *low cost* and *low reconfiguration overhead*. We make several key choices in our design. First, to keep the algorithm simple and fully-distributed, each router makes standalone decisions based on limited local information. While this sometimes leads to routing via suboptimal paths, it greatly reduces hardware complexity and cost without compromising fault coverage. Second, to eliminate any need for routing tables, we use an algorithmic approach to routing that is based on face routing. This reduces the hardware cost and complexity in each router. Third, by eliminating the need for routing tables and ensuring that our algorithm is fully-distributed, we achieve the goal of low reconfiguration overhead as each router discovers feasible paths on the fly based only on local information without the need for global information or routing table updates. Finally, to avoid any deadlocks (and livelocks) when routing with limited connectivity, we make use of *deflection routing* mechanisms [16] that have been proven to be deadlock- and livelock-free [15, 16]. While our four major design choices can potentially (but not always) lead to routing via suboptimal paths in the network, resulting in slightly increased network latency under some conditions (which we evaluate in Section 4.2), they 1) greatly reduce hardware complexity and cost, 2) enable the four goals in the design of a fault-tolerant routing algorithm, and 3) lead to increased saturation throughput with our algorithm compared to state-of-the-art fault-tolerant routing algorithms as face routing (and, hence, Maze-routing) can better exploit the NoC path diversity (which we evaluate in Section 4.2).

Maze-routing works as follows at a high-level. A router tries to forward each packet to a *productive* output port[2], as long as possible. If this is possible, the packet is considered to be in *normal mode*. If the packet enters a router with no productive output port (i.e., encounters an obstacle due to faulty NoC components), the packet traverses around the faulty region hop by hop (called *traversal mode*) until it enters a node where it is safe to revert back to *normal* mode. This procedure continues until the packet reaches its destination or one of the routers it visits detects that the destination is unreachable. We describe the details of our algorithm, including exact conditions for entry and exit into the *traversal* mode and proof of its delivery guarantee, in the rest of the paper (Sections 3.2 and 3.3).

We make the following contributions in this paper:

• We propose the first fault-tolerant routing algorithm, Maze-routing, for mesh-based NoCs that provides *all* of the following properties at the same time: 1) fully-distributed with no centralized component, 2) guaranteed delivery with livelock and deadlock freedom, 3) low area cost, 4) low reconfiguration overhead upon a fault (§3.2). We made the source code of our algorithm and our simulator publicly available [1].

• We *prove* that our algorithm finds the path to destination, if one exists, regardless of the number and location of faults in the NoC (Section 3.3). We show that our algorithm can also detect when no such path exists, i.e., when the packet cannot reach its destination

due to disconnected partitions in the network (Section 3.4).

• We show that our proposed design avoids deadlocks (and livelocks) when routing with limited connectivity, via the use of deflection routing mechanisms [16] that have been proven to be deadlock- and livelock-free [15, 16] (Section 3.5).

• We extensively evaluate the hardware cost, performance and fault tolerance capability of Maze-routing in comparison to other works that aim to provide guaranteed delivery, with three major conclusions. First, our HDL synthesis results show that the area overhead of Maze-routing is 16 times smaller than the mechanism with the smallest routing table in literature (Section 4.1). Second, our performance evaluation shows, among other things, that Maze-routing is also high performance: for example, when up to 5 links are broken, it provides 50% higher saturation throughput compared to the state-of-the-art [3, 32] (Section 4.2). Third, Maze-routing incurs much smaller reconfiguration overhead than the state-of-the-art when a new fault is discovered in the NoC (Section 4.3).

## 2. MOTIVATION AND RELATED WORK

There is an enormous amount of research carried out in detection and tolerance of different types of faults (permanent, transient, etc.) affecting operation of NoCs. In the following, we motivate four goals of this work in the design of a *practical* fault-tolerant routing algorithm, by reviewing some of the existing works. However, tolerating non-permanent faults, as well as proposing fault detection techniques fall beyond scope of this paper. As such, we narrow down our literature review to existing routing algorithms for tolerating *permanent faults* in NoC links and routers, a summary of which is reported in Table 1.

**Table 1: Comparison of state-of-the-art. Desirable characteristics are in bold.**

|  | Coverage | Reconfiguration | O(Area) | O(Reconf.) |
|---|---|---|---|---|
| Zhang et al. [43] | few | **fully dist.** | **low** | **on the fly** |
| LBDR [35] | moderate | central | **low** | N/A |
| $d^2$-LBDR [7] | moderate | central | **low** | N/A |
| OSR-Lite [38] | moderate | central | **low** | moderate |
| TOSR [5] | moderate | distributed | high | **fast** |
| BLINC [25] | moderate | distributed | high | **fast** |
| uLBDR [36] | high | central | high | N/A |
| Wachter et al. [39] | high | distributed | high | slow |
| Fick et al. [19] | high | distributed | high | slow |
| Face routing [11] | high | **fully dist.** | excessive | **on the fly** |
| FTDR-H [18] | high | **fully dist.** | high | **fast** |
| uDIREC [32] | **full** | central | high | excessive |
| ARIADNE [3] | **full** | distributed | high | slow |
| *Maze-routing* | **full** | **fully dist.** | **low** | **on the fly** |

**Goal 1: Full Fault Coverage.** As mentioned before, a large number of randomly placed failures may occur in NoC components, which may lead to disconnected (unreachable) destinations. As such, this work sets its first goal as to achieve *full (fault) coverage*, which is to guarantee to deliver a packet to its destination when a path between the source and the destination exists or else indicate that the destination is unreachable, regardless of the number and location of faults. However, most of the past works only support a limited number and placement of faults [43]. For instance, the routing algorithm in [43] handles only one faulty router/region.

On the other hand, some works (e.g. [11, 19, 36, 39]), denoted as *high coverage*, overcome a high number of faults, but do *not* guarantee a full coverage in all circumstances and/or cannot detect or tolerate unreachable destinations. Works with *moderate coverage* are also not limited to only few failures, but they place strict limitations on tolerable fault patterns as the number of faults grows.

Some researchers (e.g. [7, 32, 34–36, 38]) propose methods with *full/high/moderate coverage*, where upon new failures, a central controller collects the fault information, computes the new configuration, and distributes it back among routers. Such centralized methods suffer from two main challenges. First, the central controller can easily become the single point of failure as it is

---

[2]A productive output port is one that moves the packet closer to its destination.

on the critical path of the reconfiguration process [5]. Moreover, there needs to be an extremely reliable means for fault information collection, and routing distribution, usually addressed by use of TMR[3]-based methods [32]. Thus, centralized approaches may become impractical and/or expensive to tolerate run-time faults.

**Goal 2: Fully Distributed Operation.** Several researchers propose distributed reconfiguration methods [3, 5, 19, 25, 39], where new failures are handled through a cooperative reconfiguration process of routers. Though the reconfiguration process is carried out without any central component, these methods are prone to failure within their reconfiguration hardware/network. That is, a failed reconfiguration unit in any single router cripples the functionality of the whole system, or a portion of it. For instance, [3, 5, 25, 39] require the use of TMR-based methods and/or error-correcting codes to shield their reconfiguration process from run-time failures. This introduces a single points of failure in such otherwise distributed methods.

We posit that a practical fault-tolerant method must work in a *fully-distributed* manner: any failed component of the network should have only local impact and there should not be a single point of failure in the network. Few works, including ours, propose methods with this property. The ability of the network to route around failures and find the path to destination is an innate ability of our proposed routing algorithm. There is no separate reconfiguration unit, and a failure in any parts of the network, including our algorithm instantiations, results only in the disabling of specific network link(s).

**Goal 3: Low Area Overhead.** The imposed area overhead of a routing algorithm is of a great concern. The imposed overhead adds up not only to the implementation costs and power dissipation, but also to the vulnerability of the algorithm to run-time faults. However, most of the works that overcome a high number of faults make use of one or several routing tables [3,5,18,19,25,32,33,39]. For instance, routing tables of each node in a 8x8 network may need 256 bits [3], ∼500 bits [5, 25], or even ∼8K bits [39] to store healthy paths to destinations. As we will show in Section 4.1, routing tables significantly contribute to the area overhead of fault-tolerant algorithms. Specially, since they need to have five read ports for simultaneous accesses of different ports of a mesh router. As a failed routing table or reconfiguration logic cripples the functionality of the whole router, their imposed area overhead directly translates to increased fault probability within the whole router.

**Goal 4: Low Reconfiguration Overhead.** Finally, but also importantly, a practical fault-tolerance algorithm should provide uninterrupted operation of the network once some components become disabled. Fast reconfiguration becomes a necessity especially with the use of aggressive online testing, where network components (e.g. network link) become unavailable not only because of detected faults, but also periodically and frequently during their online testing [25, 27]. As a result, it has attracted attention newly as studied in recent literature [5,25,38]. However, most of the existing works with moderate/high/full fault coverage suffer from long reconfiguration phases [3, 19, 32, 39], during which they interrupt the normal operation of the network and pause the delivery of packets until the end of reconfiguration. For instance, in a 10x10 network, upon a new fault occurrence, it takes up to 10K cycles [3], ∼ 100K cycles [39] or even ∼ 100ms [32] to reconfigure the network. As opposed to works with fast reconfiguration, works with *on the fly* reconfiguration, including this work, have no separate reconfiguration phase: a new path to destination is dynamically calculated per packet as the packet travels through the network.

**Summary.** As shown in Table 1, to the best of our knowledge, this is the first work that can satisfy all of the four mentioned goals. Our Maze-routing algorithm *guarantees to deliver a packet to its destination* when a path exists and to indicate it when destination is unreachable, i.e. it provides *full coverage*. In addition, our routers *do not require* any routing table, work in a *fully distributed* manner, and forward packets through the faulty network *on the fly*.

---

[3]Triple Modular Redundancy.

# 3. MAZE-ROUTING

We first provide definitions and assumptions needed for the description and proof of our Maze-routing algorithm (§3.1). Afterwards, we introduce our proposed fully-distributed algorithm (source-code available at [1]) for fault-tolerant routing in meshes (§3.2), and prove that it guarantees to find a path when exists (§3.3). Further, we extend our algorithm to detect when the network is partitioned and the destination is unreachable (§3.4). Then, we explain our use of deflection-based router design to provide deadlock freedom, and changes required to keep the original properties of our algorithm (§3.5). Finally, we provide a qualitative comparison between this work and some of recent related works (§3.6).

## 3.1 Preliminaries

*Fault Model:* as mentioned above, we propose our fault-tolerant routing algorithm for mesh-based faulty NoCs, as shown in Fig. 1 (a). We model permanent failures in transistors of network components as one or several disabled links [3]. Accordingly, a failed router is modeled by disabling all of its four links.
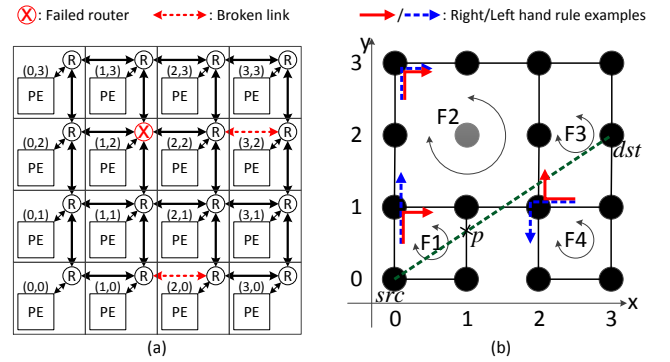


**Figure 1: (a) Architectural diagram of a 4x4 mesh NoC with faulty routers and links. (b) The equivalent planar graph.**

*Planar Graph:* In graph theory, a graph ($G$) is called planar if one can draw it on a plane without having edges crossing each other [42]. Accordingly, a (potentially faulty) mesh NoC can be represented as a planar graph, shown in Fig. 1 (b).

Note that in this work we only consider mesh topologies. However, one can modify our algorithm to fit in other topologies, as long as the topology can be represented as a planar graph.

*Face:* A planar graph partitions the plane into regions (faces) that are bounded by edges of the graph. Each connected component of a graph may have several inner faces (F1–F4 in Fig. 1 (b)), and has one outer face.

*Right/Left hand rule:* Assuming a packet entering a node (e.g. $(0, 1)$ in Fig. 1) from a direction (e.g. *south*), right/left hand rule (denoted as ↱ / ↰) decides that the packet exits the node from the first output on the packet's right/left side (e.g. *east/north*). Note that following either ↱ or ↰ rules makes the packet traverse the boundary of inner (outer) faces in clockwise (counterclockwise) and counterclockwise (clockwise) directions, respectively.

PROPERTY 1: Let $src$ and $dst$ be nodes of a planar graph $G$ with an existing path between them. Starting from $src$, by traversing the face $F$ underlying $line_{(src,dst)}$, the packet will definitely intersect $line_{(src,dst)}$ at some point ($p$) other than $src$ [20]. □

For instance, starting from $src$ in Fig. 1 (b), traversing the underlying face (F1) in both directions will intersect $line_{(src,dst)}$ in point $p$. Note that $p$ can be on either a node or an edge of the face. We refer interested readers to [20] regarding the proof of PROPERTY 1.

## 3.2 Maze-routing Algorithm

Algorithm 1 shows the pseudo-code of our distributed routing algorithm. Our algorithm routes a packet from source node ($src$)

to its destination ($dst$). As listed in Table 2, a packet carries along 4 new pieces of data. The first two are required for guaranteeing the delivery of packets, while the two other are meant for detecting unreachable destinations (§3.4). $MD_{best}$ stores the closest (Manhattan) distance (MD) to $dst$ that the packet has reached so far.[4] $Mode$ denotes the routing mode used for the packet. Initially, the packet is routed in $normal$ mode. In some circumstances, a packet might enter the $traversal$ mode, wherein, the packet follows either the right hand (↱) or left hand (↰) rules.

---

**Algorithm 1** The basic Maze-routing algorithm.

---

**Inputs:**

  *Packet header* as listed in Table 2, $src$ and $dst$.
  $cur$ indicates the coordinates of the current router.
  *Productive output ports* are calculated based on $cur$ and $dst$.
  *Healthiness of output ports* is given by online testing modules.

**Outputs:**

  $dir_{out}$: the direction in which the packet is forwarded.
  *Algorithm might also update values of packet header.*

1: **if** $cur = dst$ **then**
2:     $dir_{out} \leftarrow$ **LOCAL**;
3: **else if** $MD_{best} = MD_{cur,dst}$ **and** $\exists$(*healthy* & *productive* output) **then**
4:     $MD_{best} \leftarrow MD_{best} - 1$;
5:     $dir_{out} \leftarrow$ one of the *healthy* and *productive* outputs;
6:     $mode \leftarrow normal$;
7: **else if** $mode \in traversal$ **then**
8:     $dir_{out} \leftarrow$ The direction given by ↱ / ↰ rule;
9: **else**
10:     $mode \leftarrow$ ↱ / ↰;
11:     $dir_{out} \leftarrow$ The first output in the left/right of $line_{(cur,dst)}$;
12: **end if**
13: **return** $dir_{out}$;

---

While the packet is in $normal$ mode, each router tries to forward it to a *productive* output port (lines 3–6), until the packet reaches its destination. However, during $normal$ mode, the packet might enter a router with no productive output port. In this case (line 9 onwards), the packet enters the $traversal$ mode and starts traversing the face $F$ underlying the $line_{(cur,dst)}$. This is implemented by taking the first healthy output to the left or right of the $line_{(cur,dst)}$ (line 11) and following the ↱ and ↰ rules[5], respectively, in the next routers (line 8). In this work, we select the hand rule (↱ or ↰) randomly. This does not void the *full coverage* guarantees of our algorithm (§3.3), but can potentially lead to suboptimal paths.[6] The packet stays in $traversal$ mode until it enters a router with 1) $MD_{cur,dst} = MD_{best}$ and 2) at least one productive and healthy output towards $dst$. In other words, a packet reverts to $normal$ mode when it enters a router that can forward the packet closer to its destination than what is stored as $MD_{best}$.

Later in Section 3.3, we prove that if a packet with a reachable destination enters $traversal$ mode, it will definitely exit this mode at some node; i.e., $MD_{best}$ is definitely decremented. Accordingly, since it is guaranteed that $MD_{best}$ is gradually decrementing until zero, the packet is guaranteed to eventually reach the destination.

**Example.** We now explain our algorithm in detail with an example that is based on the faulty NoC on Fig. 1 (b). Let us assume,

---

[4]MD is computed assuming a fault-free mesh.
[5]Once a hand rule is picked (as ↱ or ↰), it cannot be changed while the packet is in $traversal$ mode.
[6]We leave it to future work to develop a mechanism to pick a hand rule that can maximize performance.

**Table 2: Packet header fields needed by our algorithm.**

|  | Valid values | Initial value | Notes |
|---|---|---|---|
| $MD_{best}$ | integers $\geq 0$ | $MD_{src,dst}$ | §3.2 |
| $mode$ | $normal$ ($N$), or $traversal = \{$↱, ↰$\}$ | $normal$ | §3.2 |
| $N_{trav}$ | node coordinates | – | §3.4 |
| $DIR_{trav}$ | $\{\uparrow, \rightarrow, \downarrow, \leftarrow\}$ | – | §3.4 |

as shown in Fig. 2, a packet that is injected to network from $src_{(0,0)}$, destined to $dst_{(3,2)}$. Starting from $src$ ($MD_{best} = 5$, and $mode = normal$), the router looks for at least one productive and healthy output (line 3 of Alg. 1). There are two candidates: $north$ and $east$ directions. The router arbitrarily chooses one ($dir_{out} \leftarrow north$ in our example), forwards the packet to $north$ and decrements $MD_{best}$ (lines 4–6). The same happens in the next router$_{(0,1)}$ and the packet is forwarded to north router$_{(0,2)}$ (with $MD_{best} = 3$ and $mode = normal$). Now, in the current router$_{(0,2)}$, the only productive output ($east$) is disabled (because router$_{(1,2)}$ is faulty) and the packet is *not* in $traversal$ mode. Hence (according to lines 9–11), without any changes to the $MD_{best}$ value, the packet enters $traversal$ mode and starts to traverse face F2 using either of the hand rules ($mode \leftarrow$ ↱ in this example, determined randomly). The packet is forwarded to the $north$ direction (the first healthy output to the left of $line_{(cur,dst)}$ – line 11 of algorithm). Traversing the face with ↱ rule (lines 7–8), the router$_{(0,3)}$ forwards the packet to the $east$ direction. Now in router$_{(1,3)}$, $MD_{best} = 3$ happens to be equal to $MD_{cur,dst}$ and there exists a productive and healthy output ($east$ direction). So, the packet exits $traversal$ mode (lines 4–6), decrements $MD_{best}$ and follows the productive output ($east$). Now, there are two productive outputs in router$_{(2,3)}$ to forward the packet ($east$ and $south$ directions). Assuming $east$ direction is taken, the next router$_{(3,3)}$ forwards the packet to the only productive output ($south$). Thus, the packet reaches its destination.
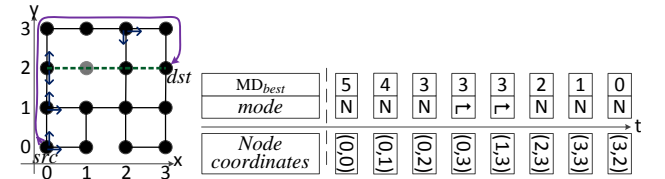


**Figure 2: An example showing a possible path to deliver a packet from $src$ to $dst$. The right figure shows the coordinates of the visited nodes and $MD_{best}$ and $mode$ values of the packet upon entering each node. The dashed line in the left figure is the $line_{(cur,dst)}$, upon entering $traversal$ mode in the router$_{(0,2)}$. Short arrows on each node show cases where more than one output is eligible according to our algorithm.**

Note that, as also seen in the given example, it is an innate ability of our algorithm to route packets to their destinations in a *fully-distributed* manner; i.e., there is no routing table or reconfiguration phase/hardware. This is a key advantage over previous works which either are not distributed, or require a routing table and/or reconfiguration hardware to achieve full fault coverage.

On the other hand, as in the example, there are two cases where the router needs to select between different output directions: 1) when there are more than one productive and healthy output ports, and 2) when selecting the hand rule for traversing around a face. Due to the distributed manner of our algorithm and thus lack of a global view, local selections done by the router in these two cases can potentially (but not always) lead to routing via suboptimal paths. For instance, in our given example, the packet would have

routed through a minimal path, if router$_{(0,0)}$ (and consequently router$_{(2,1)}$) took the *east* direction. At the same time, the ability to choose from multiple output ports offers a high path diversity to our routing algorithm, which leads to improved latency compared to state-of-the-art when the network load is high (§4.2). We leave it for future work to develop better heuristics to pick the best output port when more than one selection is available.

## 3.3 Delivery Proof

We now prove that our proposed algorithm guarantees to find a path to destination when one exists. In other words, it does not lead to livelock as packets will eventually reach their destination. Note that, in our proof, we refer to PROPERTY 1 of planar graphs, introduced in Section 3.1.

THEOREM 1. *When a path exists between src and dst, using Alg. 1, a packet from src reaches dst in a finite number of hops.*

PROOF. Alg. 1 sends the packet closer to $dst$ through productive outputs as long as they are healthy (lines 3–6). This continues until the packet reaches $dst$ (line 1), or reaches a point ($u$) which does not have a productive and healthy output towards $dst$ (line 9). In the latter, the packet enters $traversal$ mode, during which, according to to PROPERTY 1, the packet eventually will cross $line_{(u,dst)}$, at $p \neq u$. Because $p$ is on $line_{(u,dst)}$, $\text{MD}_{p,dst} < \text{MD}_{u,dst}$. Before reaching $p$, thus, the packet must have entered a node ($v$) with a productive and healthy output where $\text{MD}_{u,dst} = \text{MD}_{v,dst}$. That is, before intersecting $line_{(p,dst)}$ at $p$, the condition of line 3 of Alg. 1 gets satisfied at $v$. At that point, the packet exits to $normal$ mode and $\text{MD}_{best}$ is decremented. Thus, $\text{MD}_{best}$ is guaranteed to be decremented in $traversal$ mode. Since, MD is a discrete value and $\text{MD}_{best}$ is guaranteed to be decremented in either modes ($normal$ or $traversal$), it eventually reaches zero in a finite number of steps; i.e. the packet reaches $dst$ in a finite number of hops.□

To get an intuition of the presented proof, one may replace $u$ with router$_{(0,2)}$, $p$ with router$_{(2,2)}$, and $v$ with router$_{(1,3)}$ in Fig. 2.

## 3.4 Detecting Unreachable Nodes

When there is no path between $src$ and $dst$, it means that the packet cannot decrement its $\text{MD}_{best}$ value down to zero, but only to a minimum value of $\text{MD}_{min} \neq 0$. Accordingly, when the packet enters a node (called $\text{N}_{trav}$) with $\text{MD}_{\text{N}_{trav},dst} = \text{MD}_{min}$, there is no productive and healthy output (Otherwise, $\text{MD}_{\text{N}_{trav},dst} > \text{MD}_{min}$). In such a case, the packet traverses the current face forever in $traversal$ mode and does not exit to $normal$ mode again (Otherwise, $\text{MD}_{min}$ is not the minimum MD possible).

To develop an algorithm that can recognize an unreachable destination, we utilize the above reasoning. Algorithm 2 shows our resulting algorithm. In this algorithm, a packet needs to store the node it enters $traversal$ mode ($\text{N}_{trav}$) and the direction $\text{DIR}_{trav}$ that it is first forwarded to (lines 15 and 16). The destination is declared unreachable when the packet is revisiting $\text{N}_{trav}$ again and the $\nearrow$ / $\nwarrow$ rule guides to the same output as $\text{DIR}_{trav}$ (lines 9 and 10).

An example is shown in Fig. 3, where destination $dst$ is unreachable. The time-line on the right shows different values in packet header when entering a node. In the last step, $dst$ is detected unreachable as 1) the packet is revisiting the router$_{(0,2)}$
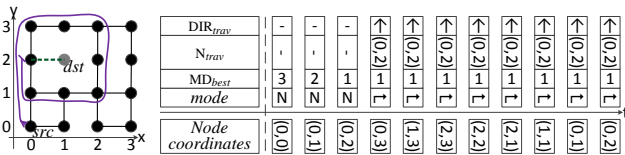


**Figure 3: (a) The path a packet takes until its destination $dst$ is detected unreachable by router$_{0,2}$. Right figure shows the coordinates of visited nodes and $\text{MD}_{best}$, $mode$, $\text{DIR}_{trav}$ and $\text{N}_{trav}$ values of the packet upon entering each node.**

---

**Algorithm 2** The enhanced Maze-routing algorithm that detects unreachable nodes.

```
1: if cur = dst then
2:     dir_out ← LOCAL;
3: else if MD_best = MD_cur,dst and ∃(healthy & productive out-
       put) then
4:     MD_best ← MD_best − 1;
5:     dir_out ← one of the healthy and productive outputs;
6:     mode ← normal;
7: else if mode ∈ traversal then
8:     dir_out ← The direction given by ↗ / ↖ rule;
9:     if cur = N_trav and dir_out = DIR_trav then
10:        Indicate dst as unreachable;
11:    end if
12: else
13:    mode ← ↗ / ↖;
14:    dir_out ← The first output in the left/right of line_(cur,dst);
15:    DIR_trav ← dir_out;
16:    N_trav ← cur;
17: end if
18: return dir_out;
```

($cur = \text{N}_{trav}$) and 2) the $\nearrow$ rule guides the packet to the same direction ($north$) as stored in $\text{DIR}_{trav}$ ($\text{DIR}_{trav} = north$).

Note that our algorithm detects such network partitioning due to faulty nodes and links, and guarantees the packet delivery within each partition; i.e., it provides *full coverage*. However, a higher level mechanism is required to recover the system from a fault; e.g., the operating system needs to migrate disconnected threads to connected nodes. This is a complementary problem and is out of this paper's scope.

## 3.5 Deadlock and Livelock Avoidance

As mentioned before, we use deflection based routing to provide our algorithm with the flexibility to take any output port, while avoiding deadlocks and livelocks. At the algorithm level, livelocks are also prevented as our algorithm guarantees to find the path to destination, as described in Section 3.3.

Deflection based routing and the design choices to avoid deadlocks and livelocks are thoroughly studied in literature [4, 6, 16, 21, 28, 29]. Deflection routing can provide deadlock freedom by deflecting flits when there is contention. In order to guarantee that our design has no deadlock, our mechanism always ensures that the number of output ports is equal to the number of input ports. If there is a broken input or output link in the router, a corresponding output/input link will also get disabled to ensure the number of input ports equals to the number of output ports at all times.

In this work, we have modified the deflection router used in minBD [16] to incorporate our proposed algorithm. In order to avoid livelocks, minBD applies two mechanisms: 1) *golden and silver flits*, which guarantee that each flit will eventually arrive at its destination [4, 15, 16] and 2) *re-transmit once*, which guarantees that flits can be reassembled at the destination in order to make forward progress [4, 15, 16]. Furthermore, re-transmit once flow-control avoids additional buffering cost, otherwise necessary for deadlock avoidance, by using *cache miss buffers (MSHRs)* as reassembly buffers. In conclusion, the combination of deflection routing, golden and silver flits, and re-transmit once provides end-to-end guarantees for all flits and ensures that the network is always deadlock- and livelock-free.

Note that the exact proof of deadlock and livelock freedom of deflection based mechanisms is beyond the scope of this paper. We refer interested readers to the original works for details of each mechanism [15–17]. Moreover, our algorithm can work with any deflection based mechanism proposed in literature. We chose minBD [16] due to its ease of adaptation and high performance.

**Deflection implications.** With a deflection-based router, in some cases, a flit might not be forwarded to the output port indicated by our routing algorithm (Alg. 1 and 2). As such, a deflected flit is no longer following our algorithm: it exits the face it is traversing or takes a non-productive output, both of which, result in inconsistency of the $\text{MD}_{best}$ and $mode$ values of the packet. In order to ensure correct functionality of our algorithm, in the presence of deflection, the $\text{MD}_{best}$ and $mode$ values of the flit are reset as if the packet were just injected to the network in the next router, $r_D$ ($\text{MD}_{best} \leftarrow \text{MD}_{r_D,dst}$ and $mode \leftarrow normal$).

## 3.6 Qualitative Comparison with Previous Methods

Basic face routing algorithm [11] draws $line_{(p,dst)}$ between sour-ce $src$ and destination $dst$, and traverses the underlying face until an edge intersects the $line_{(p,dst)}$. The procedure of shortening $line_{(p,dst)}$ is continued until reaching $dst$, as shown in Fig. 4. The face routing algorithm is not a suitable candidate for on-chip implementation in its basic form. It needs to store coordinates of $p$, which are real (vs. integer) numbers, and requires realization of floating-point operations (with the associated area and energy overheads) at every input port. Moreover, the intersection check can lengthen the critical path of the algorithm and add up to the hardware overhead. Finally, the basic face routing algorithm does not provide a method to detect it when nodes become unreachable due to faults.
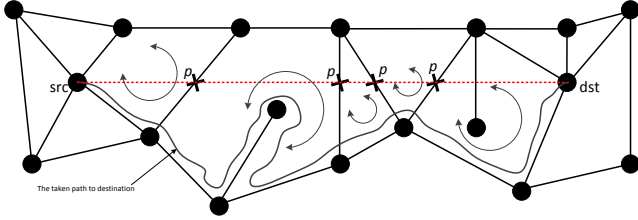


**Figure 4: An example of the basic face-routing algorithm.** $line_{(src,dst)}$ **is shown in red, and intersection points are indicated by ✕.**

Tunneled OSR (TOSR) [5] tries to overcome two main drawbacks of its predecessor, OSR-Lite [38]. TOSR reconfigures the network in a distributed manner, and achieves much faster reconfiguration. Their reconfiguration speed clearly stands out among related state-of-the-art [25, 38]. However, TOSR (like OSR-Lite) has limited coverage capabilities; e.g., it cannot always recover the network when there are two failed links. Moreover, it makes use of routing tables (464 bits per router) imposing a high area overhead (§4.1). In addition, the reconfiguration logic of TOSR needs to be shielded by use of TMR-based methods (as mentioned in Section 2); otherwise, a failed unit can damage the functionality of the technique. In contrast, in our method, failed routing logic of a router can be simply taken offline by disabling the associated input and output ports.

Logic based distributed routing (LBDR) [35] based methods are introduced to remove the area overhead of routing tables. LBDR works with few configuration bits (around 26) per router, and can implement several routing algorithms. Nevertheless, naïve LBDR is not capable of providing *high coverage* in case of failures, when shortest path routing is not possible. Universal LBDR (uLBDR) [36] is proposed to overcome this shortcomings. Unfortunately, $d^2$-LBDR [7] reports 3x area overhead for uLBDR due to the need of virtual cut through switching, use of FORK modules, and complex arbitration. Moreover, LBDR and its derivatives are not routing algorithms by themselves and need a central module to collect fault data, compute new configuration bits, and update each unit accordingly. Our technique is free of such overheads due to its four desirable characteristics, explained in Sections 1 and 2.

FTDR-H [18] is a hierarchical reinforcement learning based algorithm which uses deflection based routing. It achieves *high coverage* and can route packets to their destination on the fly upon new failures. However, it uses large routing tables (around 600 bits) per router and leads to livelock once a destination gets disconnected. Our Maze-routing mechanism is livelock free and does not need routing tables.

## 4. RESULTS

We evaluated our proposed algorithm using a publicly available simulator, NOCulator [2], modeling out-of-order x86 CPUs each with a private L1 cache and a share L2 cache. The simulator has been used and verified in many previous works [4,12,15,16,30,31]. L2 cache slices are connected together with a mesh interconnect. In this work, we model 64 CPU cores in an 8x8 network, each of them running a SPEC2006 application [22]. In addition, we present synthetic traffic analysis.

## 4.1 Area Overhead

For the area overhead evaluation, we implemented our algorithm using Verilog HDL. We then synthesized the design using Cadence RC (RTL compiler) tool at the STMicro 60nm technology node. Five copies of the algorithm are instantiated, one for each input port of a router.

We also implemented the routing table used by related works [3, 5, 19, 25, 32, 33, 39]. Among these, ARIADNE [3] uses the smallest table in comparison to others, and we compare to it. Note that we did not implement the reconfiguration logic of each work, which will lead to even a higher area overhead for these past works. In addition to ARIADNE, for the sake of fair comparison, we implemented LBDRe, a logic based table free method based on the LBDR [35] method.

Table 3 shows the silicon area required by our mechanism compared to ARIADNE and LBDRe, for 8x8 and 16x16 meshes. Compared to $ARIADNE$, the area reduction of our mechanism is 3.8 and 15.9 times for 8x8 and 16x16 meshes, respectively. Though LBDR based method imposes a lower area overhead compared to our method, both methods scale well as network size increases. However, LBDR suffers from being a central approach with limited coverage, and uLBDR [36], an extension to achieve *high coverage*, is reported to impose 3x overhead in the entire router area [7].

**Table 3: Silicon area (in $\mu m^2$) required by five instances of our algorithm (one for each port), routing table of ARIADNE [3], and logic-based LBDRe mechanism [35].**

| Mesh size | Maze-routing | ARIADNE | LBDRe |
|-----------|--------------|---------|-------|
| $8 \times 8$ | 1184 | 4471 | 568 |
| $16 \times 16$ | 1505 | 23921 | 606 |

In addition to the efficiency and scalability of our algorithm, the use of deflection routing significantly reduces our technique's area overhead as it reduces buffering in routers. The minBD router we use has 39% smaller area than a buffered wormhole router [16]. In addition, the use of *retransmit-once* scheme [15] reduces the buffering overhead at receiving nodes in our technique.

On the other hand, because each flit of a packet is treated independently in deflection-based methods [29], the packet header information listed in Table 2 should be sent along with all flits. This means that the channel width of routers needs to be lengthened to accommodate this information, which results in increased area in channels and routers (there is an almost quadratic relation between the channel width and the router area [26]). The fields of Table 2 can be coded in 14/17 bits in 8x8/16x16 meshes. Assuming a baseline router with 144-bit channel width [23] , we need to widen the channel by 10%/12%, which results in almost 20%/25% increase in the router area. Note that, even with the imposed overhead of the widened channels, the area cost of our method is far less than

methods with routing tables. We leave a more precise and accurate study/comparison of the area overhead to future work.

## 4.2 Throughput

In order to evaluate the performance of our algorithm, we extracted the average flit latency in the network under different injection rates using a uniform traffic. Fig. 5 shows the result for both our algorithm (blue lines) and the up*/down* [37] (red lines) using an 8x8 network. We study two cases (1 and 5 randomly failed links), generate 10 different fault patterns for each case, and provide average results across all of them. We run the simulations for 10 million cycles.
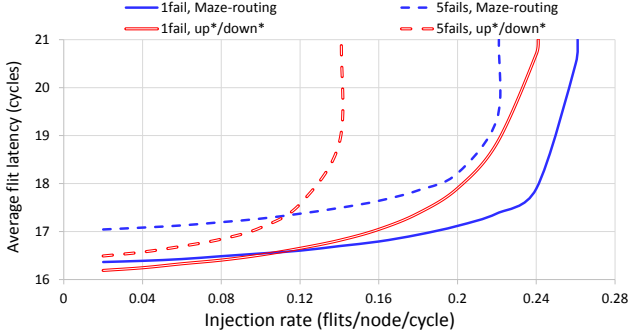


**Figure 5: Average flit latency for both up*/down* algorithm and our mechanism, over different injection rates with uniform random traffic. Solid and dashed lines, respectively, correspond to 1 and 5 randomly failed links.**

We compare our technique against the up*/down* proposal, as the latter is the core algorithm implemented in several related works [3, 32, 35, 36] and can provide *full coverage*. Our algorithm utilizes a router design proposed in minBD [16] with 16 buffer entries. Up*/down* is implemented using normal buffered wormhole routers [40], with 40 buffer entries in each router (8 to each port).

As up*/down*, on average, provides a shorter path to destination compared to our technique, its average packet latency is smaller at low network loads. However, at high injection rates, our algorithm achieves better latency results as well as higher saturation throughput due to its better exploration of path diversity compared to up*/down*.

In another set of experiments, we run a mix of 15 different workloads from SPEC CPU2006 [22]. Accordingly, we split benchmarks into three categories based on their L1 misses per kilo instruction (MPKI), as these are the misses that will go over the network. High intensity benchmarks have MPKI greater than 50, Medium intensity ones fall between 5 and 50 MPKI, and Low intensity benchmarks are the remainder. Based on these categories, we randomly pick a number of applications and form the following mixes: L (all Low), ML (Medium/Low), M (all Medium), and H (all High).

Table 4 shows the average network latency of the mentioned workload mixes, when there is no failure and when there are up to 5 failed links in the network. Our mechanism achieves lower average packet latency compared to up*/down*, especially in the high load cases, due to the offered path diversity.

## 4.3 Reconfiguration Overhead

In this subsection, we evaluate the performance of our algorithm in achieving our fourth goal, minimal reconfiguration overhead. As described in Section 3.2, there is no reconfiguration phase that a router enters upon a failure detection. Instead, each router tries to traverse around the failed link by forwarding the packets in *traversal* mode.

In contrast, in related work [3,5,25,32,38,39], a router enters the reconfiguration phase and network operation is interrupted until a

**Table 4: Average packet latency with our techniques vs. up*/down* on an 8x8 mesh, running different mixtures of SPEC benchmarks.**

| workload mix | Up*/Down* 5 failures | no failure | Maze-routing 5 failures | no failure |
|---|---|---|---|---|
| L | 16.7 | 16.4 | 17.8 | 16.4 |
| ML | 18.8 | 18.2 | 18.9 | 17.2 |
| M | 27.7 | 25.7 | 21.6 | 19.2 |
| H | 54.4 | 50.5 | 25.8 | 23.1 |
| AVG | 29.4 | 27.7 | 21.0 | 19.0 |

new routing solution is found. Accordingly, these works' intrusions on normal operation of the network increases with the time it takes to find an alternative solution. For an 8x8 mesh, it is reported that it can take ∼20 [5], ∼30 [25], ∼350 [38] and even 4K [3] cycles to reconfigure the network. However, most of these works can work only for 1 failure per their defined segment [5, 25, 38], and only ARIADNE can guarantee *full coverage* as our method.

To demonstrate the *on-the-fly* reconfiguration capability of our proposal, we simulate a fault free 8x8 network with injection rate constantly set to $0.2\ flits/node/cycle$. We inject two random link failures to the network, one at cycle 200K, and the other at cycle 400K. As shown in upper part of Fig. 6, while ARIADNE takes only 4K cycles to reconfigure the network, it takes over 40K cycles for the network to return to its steady state latency. This becomes even worse when the second failure occurs. In contrast, using our proposed algorithm, the network latency only slightly increases (∼0.25 cycles) upon every failure (Fig. 6– lower part). Continuing new failure injection, ARIADNE design gets saturated once the 5th link fails, while our algorithm continues to deliver flits with a slight increase in latency (from 16.5 cycles to 18.4, with 6 link failures).

We believe the main advantage of on-the-fly reconfiguration is that, using our algorithm, one can freely use online testing methods. Links and network components can be silently disabled to run test methods on them, without any major impact on performance.
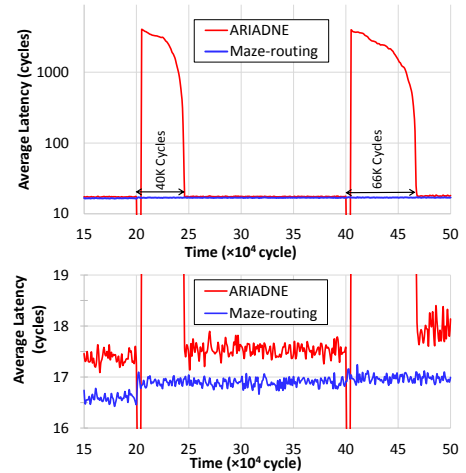


**Figure 6: The moving average of network latency when new failures are injected during normal operation of the network. (upper) It takes several kilo cycles for ARIADNE to return to steady state. (lower) The zoom-in of the same experiment: using our algorithm, flits experience around 0.25-cycle additional latency for each new failure**

## 5. CONCLUSION

We introduced Maze-routing, a new, practical fault-tolerant routing algorithm with delivery guarantees for networks-on-chip. We have shown that Maze-routing 1) is fully-distributed, 2) guarantees delivery when a path exists and, otherwise, indicates that the destination is unreachable, 3) has low hardware area overhead, 4) incurs low reconfiguration overhead upon a new fault. We experimentally evaluated the area overhead, performance, and fault tolerance capability of Maze-routing, with comparisons to state-of-the-art routing algorithms that provide guaranteed delivery. Our evaluations show much lower area overhead and much higher saturation throughput (when faults exist in the NoC) than the state-of-the-art. We conclude that our proposal can provide an efficient and high-performance routing substrate for future NoCs, where fault tolerance is expected to become increasingly important.

## Acknowledgments

## REFERENCES

[1] Maze-routing. https://github.com/CMU-SAFARI/NOCulator/tree/Maze-routing.

[2] NOCulator. https://github.com/CMU-SAFARI/NOCulator/.

[3] K. Aisopos et al. ARIADNE: Agnostic reconfiguration in a disconnected network environment. In *PACT*, 2011.

[4] R. Ausavarungnirun et al. Design and evaluation of hierarchical rings with deflection routing. In *SBAC-PAD*, 2014.

[5] M. Balboni et al. Synergistic Use of Multiple On-Chip Networks for Ultra-Low Latency and Scalable Distributed Routing Reconfiguration. In *DATE*, 2015.

[6] P. Baran. On distributed communications networks. *IEEE Trans. on Comm.*, 1964.

[7] R. Bishnoi et al. d2-LBDR: Distance-Driven Routing to Handle Permanent Failures in 2D Mesh NoCs. In *DATE*, 2015.

[8] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *MICRO*, 2005.

[9] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, 2007.

[10] S. Borkar et al. Microarchitecture and design challenges for gigascale integration. In *MICRO*, 2004.

[11] P. Bose et al. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless networks*, 2001.

[12] K. K.-W. Chang et al. HAT: Heterogeneous Adaptive Throttling for On-Chip Networks. In *SBAC-PAD*, 2012.

[13] K. Constantinides et al. Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *MICRO*, 2007.

[14] W. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *DAC*, 2001.

[15] C. Fallin et al. CHIPPER: A low-complexity bufferless deflection router. In *HPCA*, 2011.

[16] C. Fallin et al. MinBD: Minimally-Buffered Deflection Routing for Energy-Efficient Interconnect. In *NoCS*, 2012.

[17] C. Fallin et al. Bufferless and Minimally-Buffered Deflection Routing. *Book chapter*, 2014.

[18] C. Feng et al. Addressing Transient and Permanent Faults in NoC With Efficient Fault-Tolerant Deflection Router. *IEEE TVLSI*, 2013.

[19] D. Fick et al. A highly resilient routing algorithm for fault-tolerant NoCs. In *DATE*, 2009.

[20] H. Frey and I. Stojmenovic. On delivery guarantees of face and combined greedy-face routing in ad hoc and sensor networks. In *MobiCom*, 2006.

[21] M. Hayenga et al. Scarab: A single cycle adaptive routing and bufferless network. In *MICRO*, 2009.

[22] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[23] J. Howard et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.

[24] I. Koren and C. M. Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2010.

[25] D. Lee et al. Brisk and Limited-Impact NoC Routing Reconfiguration. In *DATE*, 2014.

[26] J. Lee et al. Do We Need Wide Flits in Networks-on-Chip? In *ISVLSI*, Aug 2013.

[27] Y. Li et al. Concurrent Autonomous Self-Test for Uncore Components in System-on-Chips. In *VTS*, 2010.

[28] Z. Lu, M. Zhong, and A. Jantsch. Evaluation of On-chip Networks Using Deflection Routing. In *GLSVLSI*, 2006.

[29] T. Moscibroda and O. Mutlu. A case for bufferless routing in on-chip networks. 2009.

[30] G. Nychis et al. Next Generation On-Chip Networks: What Kind of Congestion Control Do We Need? In *HOTNETS*, 2010.

[31] G. P. Nychis et al. On-Chip Networks from a Networking Perspective: Congestion and Scalability in Many-core Interconnects. *SIGCOMM*, 2012.

[32] R. Parikh and V. Bertacco. uDIREC: unified diagnosis and reconfiguration for frugal bypass of NoC faults. In *MICRO*, 2013.

[33] V. Puente et al. Immunet: A Cheap and Robust Fault-Tolerant Packet Routing Mechanism. In *ISCA*, 2004.

[34] P. Ren et al. Fault-tolerant routing for on-chip network without using virtual channels. In *DAC*, 2014.

[35] S. Rodrigo et al. Efficient implementation of distributed routing algorithms for NoCs. *Computers Digital Techniques, IET*, 2009.

[36] S. Rodrigo et al. Cost-Efficient On-Chip Routing Implementations for CMP and MPSoC Systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2011.

[37] M. Schroeder et al. Autonet: a high-speed, self-configuring local area network using point-to-point links. *Selected Areas in Communications, IEEE Journal on*, 1991.

[38] A. Strano et al. OSR-Lite: Fast and Deadlock-Free NoC Reconfiguration Framework. In *SAMOS*, 2012.

[39] E. Wachter et al. Topology-Agnostic Fault-Tolerant NoC Routing Method. In *DATE*, 2013.

[40] H. Wang et al. Power-Driven Design of Router Microarchitectures in On-Chip Networks. In *MICRO*, 2003.

[41] D. Wentzlaff et al. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 2007.

[42] D. B. West et al. *Introduction to Graph Theory*, volume 2. Prentice Hall, Upper Saddle River, 2001.

[43] Z. Zhang et al. A Reconfigurable Routing Algorithm for a Fault-Tolerant 2D-Mesh Network-on-Chip. In *DAC*, 2008.