Highly Fault-tolerant NoC Routing with Application-aware Congestion Management

Doowon Lee, Ritesh Parikh and Valeria Bertacco
Department of Computer Science and Engineering, University of Michigan
{doowon, parikh, valeria}@umich.edu

ABSTRACT

Silicon devices are becoming less and less reliable as technology moves to smaller feature sizes. As a result, digital systems are increasingly likely to experience permanent failures during their lifetime. To overcome this problem, networks-on-chip (NoCs) should be designed to, not only fulfill performance requirements, but also be robust to many fault occurrences. This paper proposes a fault- and application-aware routing framework called FATE: it leverages the diversity of communication patterns in applications for highly faulty NoCs to reduce congestion during execution. To this end, FATE estimates routing demands in applications to balance traffic load among the available resources. We propose a set of novel route-enabling rules that greatly reduce the search for deadlock-free, maximally-connected routes for any faulty 2D mesh topology, by preventing early on the exploration of routing configuration options that lead eventually to unviable solutions. Our experimental results show a 33% improvement on average saturation throughput for synthetic traffic patterns, and a 59% improvement on average packet latency for SPLASH-2 benchmarks, over state-of-the-art fault-tolerant solutions. The FATE approach is also beneficial in the complete absence of faults: indeed, it outperforms prior fully-adaptive routing techniques by improving the saturation throughput by up to 33%.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors—*Interconnection architectures*; B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Reliability, Performance, Algorithms, Design

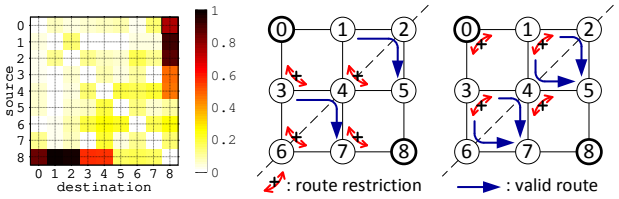
Keywords

Network-on-Chip, Fault-Tolerance, Adaptive Routing

1. INTRODUCTION

Advances in semiconductor fabrication technology have enabled the design of modern chip multiprocessors (CMPs) and systems-on-chip (SoCs) consisting of billions of transistors. They deploy tens, or even hundreds, of communicating components and, therefore, efficient on-chip communication is increasingly becoming a critical design bottleneck. Networks-on-chip (NoCs) are a promising interconnect solution because they provide massively concurrent, scalable and power-efficient communication. However, the increasing susceptibility to faults of nano-scale semiconductor devices [13] makes it extremely challenging to maintain the correctness and low-latency characteristics that are desired for NoCs. To make matters worse, NoCs constitute a single-point-of-failure for the entire system.

NOCS'15, September 28 - 30, 2015, Vancouver, BC, Canada
 ISBN 978-1-4503-3396-2/15/09 ...$15.00
DOI: http://dx.doi.org/10.1145/2786572.2786590



(a) Traffic pattern for *lu_con* (b) Routing function 1 (c) Routing function 2

Figure 1: **Traffic-aware routing restrictions**. (a) The normalized traffic pattern for a portion of *lu_con* [25] shows that the largest fraction of traffic is between nodes 0-4 and 8. (b) When selecting the routing function 1, which forbids the north-east turns at node 3, 4, 6 and 7, there are only two allowed turns to cross the mid-diagonal line. (c) In contrast, the placement of routing restrictions as in the routing function 2 allows four distinct turns across the mid-diagonal.

Fault-tolerant NoC routing solutions [1,14,20,21] tackle this challenge by leveraging their inherent routing flexibility. In other words, the routing solutions react to faults by limiting communication to flow only along fault-free paths. In this context, topology-agnostic routing algorithms [10,22,23] offer highly flexible routing, preserving network connectivity even in the presence of many faults. Nonetheless, they often lead to severe performance degradation after only a few faults, making the continued deployment of the chip impractical. For example, in [20], the throughput of an 8×8 mesh network drops by over 20% after only 10 faults. This steep loss is mainly due to increased traffic congestion on the remaining healthy paths. Thus, runtime-management of traffic flow has a critical impact on the performance of faulty networks.

Adaptive routing techniques to manage traffic congestion have been extensively investigated [2,12,17,19]. They mitigate interference among packets by routing some of them through underutilized network links. Prior adaptive routing solutions can be partitioned in two main groups: one group includes those optimized for regular topologies (*e.g.*, mesh) [12,17], while the other targets very general, topology-agnostic solutions [7,8,19]. Unfortunately, the former is inadequate to tackle faulty regular networks (which reduce to irregular topologies), while the latter entails almost always extremely complex and resource-demanding computations.

CMP applications [5,25] exhibit communication patterns [4] where most packets flow among only a subset of the NoC nodes. This aspect could be leveraged to focus the limited routing options available towards a handful of high-traffic communication paths. This is the key observation that led to our solution: if the high-traffic communication patterns in a faulty NoC are known (or can be estimated), then we can **select the routing function so as to provide both deadlock**

freedom and maximum path diversity (and thus low congestion) among the high-traffic nodes. In other words, the routing function eliminates only underutilized routes. Consequently, we can provide high-bandwidth, low-latency communication even in faulty networks. Consider, as an example, the traffic pattern shown for a portion of the *lu_con* benchmark from SPLASH-2 [25] in Figure 1a, where the majority of the packets are transferred between nodes 0-4 and node 8. Using the route restrictions shown in Figure 1b to break deadlock cycles, there are only two turns that allow transferring traffic from the upper-left region of the network to its lower-right region. In contrast, the route restrictions shown in Figure 1c allow four distinct turns. As a result, the routing solution of Figure 1c is less likely to cause congestion. Application-aware solutions of this type have been investigated in [7,8,16,19,24], but they often entail high routing computation overheads.

Contributions. Our proposed solution is called FATE, **F**ault- and **A**pplication-aware **T**urn model **E**xtension. It manages congestion in faulty networks by selecting routing restrictions appropriately. Specifically, restriction placements are optimized to the application's network traffic. We attain this goal by introducing a set of *turn-enabling rules* that allow to quickly prune the search of deadlock-free routes in faulty network topologies. We leverage these rules to provide adaptive, application-aware routes for packets flowing in the network, maximizing the number of distinct available routing paths. The rules can be applied to any irregular topology derived from a 2D mesh network by injecting faults. Unlike previous topology-agnostic routing solutions, FATE takes into consideration bandwidth demands, in addition to fault locations, when placing the turn restrictions. Moreover, it also keeps its computation lightweight, when compared to existing application-aware solutions. In summary, we make the following contributions:

- We present a novel, fault- and application-aware routing restriction placement solution, called FATE. FATE improves the performance of faulty 2D mesh networks by leveraging the application's communication patterns.
- We demonstrate a method to quickly prune the exploration of viable routes in faulty networks, in order to consider only deadlock-free options, and we reduce the number of routes evaluated by two orders of magnitude over prior application-aware solutions.

2. RELATED WORK

Fault-tolerant routing. Fault-tolerant, deadlock-free routing solutions have been extensively investigated in the past. Glass and Ni propose three turn-models that provide adaptive, fault-tolerant routing [11]. Note, however, that their fault-tolerance is limited to only a few faults [21]. In contrast, Ariadne [1], uDIREC [20], CBCG [21] and Hermes [14] put no constraints on the number and location of faults, and hence, they are more reliable. While these latter works provide deadlock-free, maximally-connected routes, none of them take traffic flow into consideration to optimize for throughput.

Similar to the above on-chip solutions, in the off-chip network domain, topology-agnostic routing algorithms [10,22,23] focus mostly on placing routing restrictions using topological characteristics, while ignoring traffic flow during the placement. On the other hand, we exploit communication patterns to find better routing restrictions.

Application-aware routing. Application-aware routing solutions tune the routing function to traffic flow. APSRA [19] strives to meet the bandwidth requirement of an application, while achieving deadlock freedom by breaking cyclic dependencies. However, deploying this solution to identify optimal routes in faulty networks has an extremely high computational cost. Addressing APSRA's limitation, ACES [8] investigates a quick application-aware routing heuristic in irregular topologies by partitioning VCs to break cycles. LBDRx [7] develops a resource-mapping tool and a routing algorithm for SoC applications, while reducing routing overheads by leveraging logic-based routing instead of routing tables. Note that all of the above solutions use adaptive routing to mitigate congestion. In contrast, BSOR [16] selects load-balanced, oblivious paths via either mixed integer linear programming (MILP) or Dijkstra's shortest path algorithm to approximate optimally balanced paths. Similarly, ETM [24] reduces the computation requirements of the MILP problem with the aid of a genetic algorithm. These latter two oblivious routing solutions, however, suffer from low performance as they do not manage runtime congestions well. Although all of these application-aware solutions can be deployed in faulty networks, they often require either intractable computation or extra hardware.

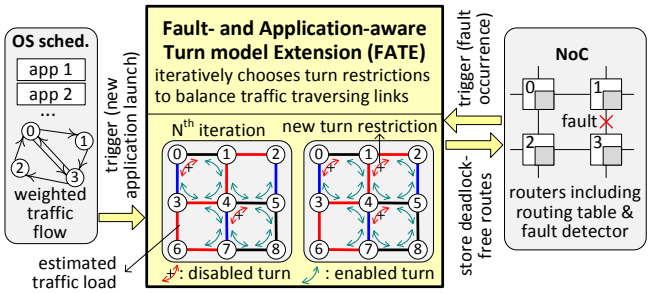


Figure 2: **Overview of FATE**. FATE is triggered by either a new application launch or a new fault occurrence. It leverages a traffic load estimator to compute a deadlock-free routing function that minimizes congestion. The new routing function is then stored at the routers.

Application-aware routing techniques can manage congestion more efficiently when employing runtime congestion monitoring. Various monitoring schemes have been explored in 2D mesh NoCs. For example, DyXY [17] observes buffer occupancy in adjacent routers, and then favors forwarding packets to routers with more vacancies. NoP [2] and RCA [12] extend this idea by monitoring congestions in routers farther away. All such solutions are fully-adaptive, and thus, require dedicated VCs to guarantee deadlock freedom, a costly endevour in area- and power- constrained NoCs.

3. OVERVIEW AND BACKGROUND

FATE is a software-based solution that generates a deadlock-free routing function, while maximizing the number of distinct paths available between nodes with high communication requirements, for applications running on faulty 2D mesh networks. Figure 2 shows how FATE operates: it is triggered by a new fault occurrence or an application launch. It then uses the CMP's idle cores to compute a new routing function. If no idle core is available, the start of the application may be delayed to compute the routing function, or a prior, possibly non-optimal, function may be used, until a core becomes available. While finding optimal, deadlock-free routes is an intractable problem [8,16,24], FATE's efficient heuristic quickly discovers a near-optimal routing function: it first computes the minimal number of turn restrictions that must be placed in the network. This value can be easily derived from the total number of cycles in the topology. The construction is then based on an iterative exploration, where turn restrictions are placed one at a time. After placing each turn restriction, FATE deploys its *turn-enabling rules* (Section 4) to enable turns that must be active in order to maintain connectivity in light of the most recent turn-restriction choice. Moreover, in choosing the location of each new turn restriction, FATE leverages traffic load estimates that it derives from the communication patterns extracted via application analysis (Section 5). If a deadlocked or disconnected routing configuration is encountered during the exploration, FATE uses backtracking to broaden the search until it finds a

satisfiable solution. Finally, the network's routers are re-programmed using the new routing function.

FATE assumes that information about the application's communication patterns is available: indeed, these can be observed in CMP and SoC applications using runtime profiling, and then modeled through Markov chains [3,4]. As in many fault-tolerant routing solutions, we also assume that the NoC is equipped with a fault diagnosis solution (*e.g.*, switch-to-switch detection [18]). In addition, we assume that our solution is deployed in systems where the OS is notified of new fault detections and can launch FATE's routing computation software, updating routing tables accordingly. This assumption may not hold in some systems. For those systems, FATE can be adopted by deploying a dedicated network manager. In this setup, FATE calculates in advance optimized routing functions for representative communication patterns, and stores them in memory. At runtime, the network manager evaluates current traffic patterns, and chooses the best routing function by comparing against the patterns of the stored ones.

3.1 Avoiding Deadlock by Removing Cyclic Resource Dependencies

Deadlock situations can happen when packets wait for each other in a cyclic manner. Such situations can be avoided by breaking cyclic resource dependencies [9], disabling at least one of the turns that contribute to the cycle [11]. For instance, Figure 1b forbids the turn 0-3-4 so that packets' routes do not include the sequence 0→3→4 or 4→3→0. This turn restriction breaks the cycle along nodes 0, 1, 4 and 3. Note that FATE uses bidirectional turn restrictions.

This deadlock-avoidance technique has been proposed in the past: for instance, the turn models [11] forbid certain turns in 2D meshes to break every possible cycle. For instance, the west-first turn model disallows all turns towards the west direction, so that packets can only go west at the beginning of their routing path. However, this baseline turn model often fails to provide either deadlock freedom or maximal connectivity in the presence of many faults [1,21].

A class of topology-agnostic routing algorithms, however, can ensure both deadlock freedom and connectivity in any topology, providing strong fault-tolerance capabilities [1,10,21,22,23]. For instance, the up*/down* routing algorithm [1,22,23] builds a spanning tree of links. Links towards the root node are marked as *up* links, while links towards the leaves are marked as *down* links [20]. The routing algorithm then disallows down-up turns: once a packet takes a down link, it is not allowed to turn onto an up link. While this restriction rule can be applied to any up*/down* tree in general, the routing algorithm usually spans the tree in a fixed manner (*e.g.*, breadth-first [1,23] or depth-first [22]). Consequently, these topology-agnostic approaches often provide limited options in the placement of turn restrictions.

Application-specific routing solutions aim at offering full flexibility in the turn-restriction placement [7,8,19]. They analyze an application's communication paths, and selectively remove some of those paths until no cyclic resource dependency exists. As discussed in [8], however, this type of approaches (*e.g.*, [19]) scales poorly with network size. [8] tries to overcome this scalability issue in two steps. It first minimizes the number of cyclic dependencies, and then uses VCs to break the remaining cycles. However, using VCs to avoid deadlock is a costly option in resource-constrained NoCs. In contrast, our approach does not require VCs for deadlock freedom, and VC resources can be entirely dedicated to prioritize traffic or reduce congestion.

4. FATE'S TURN-ENABLING RULES

This section discusses the rules that determine which turns must

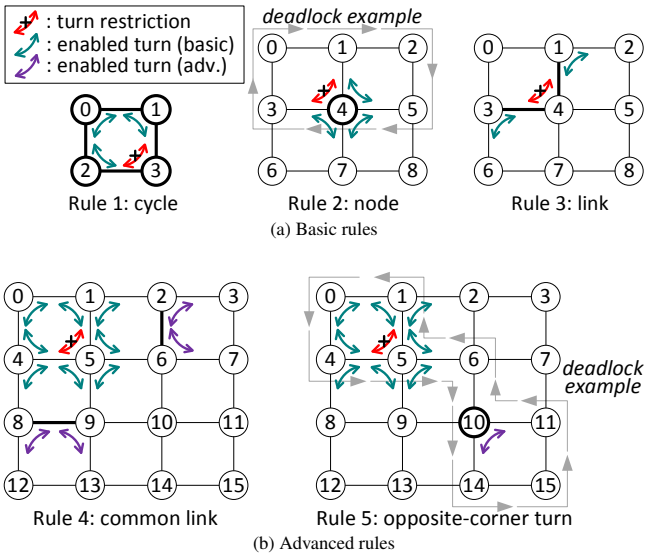


Figure 3: **FATE's turn-enabling rules**. Whenever FATE selects a new turn restriction, it can infer a number of other turns that should be enabled at other locations to obtain an optimal routing function (minimal number of disabled turns) faster. Basic rules (a) enable turns in adjacent locations, while advanced rules (b) impact remote turn locations. In the diagram for Rules 2 and 5, we show a deadlock cycle that we would obtain if we did not enable the turns indicated.

be enabled as a consequence of another turn being disabled. These rules, grouped into *basic* and *advanced*, can be applied in any order, and are illustrated in Figure 3. We first describe how the rules operate in regular meshes, and then extend them to faulty topologies. Note that our approach minimizes the number of turn restrictions, and thus maximize cumulative bandwidth to all destinations, but it does not necessarily enable minimal-length routes.

1) Basic rules identify which turns must be enabled because of a turn restriction on the surrounding cycle, node and links, and they are illustrated in Figure 3a.

Rule 1 (**cycle**) — *Once a turn in a cycle is disabled, all other turns in the same cycle should be enabled*, so that all nodes in the cycle can still communicate.

Rule 2 (**node**) — *Once a turn in a node (router) is disabled, all other turns insisting on the same node should be enabled.*

Rule 3 (**link**) — *Once a turn adjacent to a link is disabled, the turn to the same link, on the opposite side with respect to this turn and not insisting on the same router should be enabled.*

In a fault-free mesh network, there is only one turn that should be enabled for each link affected by a disabled turn as a result of Rule 3. Note that Rules 2 and 3 are not necessary to guarantee the connectivity of the network, but any violation of these rules would lead to a superfluous number of turn disabling. For instance, if both the turns 1-4-3 and 1-4-5 were disabled in the middle network of Figure 3a, then the network would still allow a dependency along the path 0→1→2→5→4→3→0 as shown in the figure (gray arrows), and we would need to include an additional turn restriction to break it. Similarly, with reference to the right network in Figure 3a, if turn 4-1-2 were disabled, a cyclic dependency would exist along the path 0→1→2→5→4→3→0.

2) Advanced rules force FATE to enable turns that are remote with respect to the restricted turn. They allow to aggressively prune the search space towards a solution with a minimal number of disabled turns. Figure 3b illustrates the two rules below; we marked in red the restricted turn, in teal the turns enabled by the basic rules, and in purple the turns enabled by the advanced rule being illustrated.

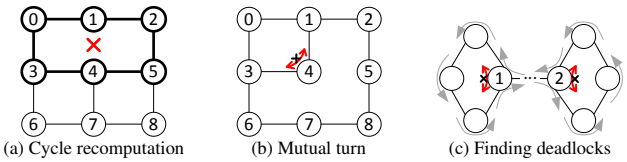


Figure 4: **Extending turn-enabling rules to faulty topologies**. The rules in Figure 3 can be applied in the presence of faults with a few modifications. For instance, (a) cycles must be recomputed in faulty regions, and (b) a turn shared by convex- and concave-shaped cycles should be used to break only one of the cycles. (c) A deadlock may occur when there is a path between two nodes belonging to different cycles and both nodes are the one with the disabled turn for its cycle.

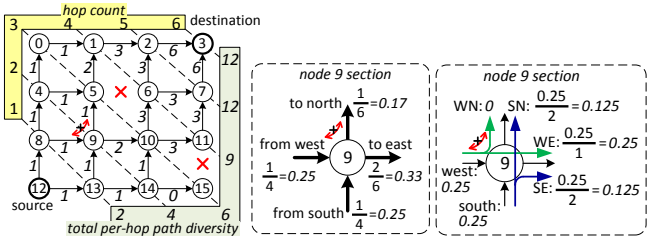


Figure 5: **Link-load and turn-load estimation example**. We compute traffic load of links and turns using path diversity. (a) The path diversity is calculated at each link considering fault locations and turn restrictions. (b) The link load is computed by dividing its path diversity by the per-hop path diversity. (c) The turn load is derived from the load of its input link and the number of permissible outputs.

Rule 4 (**common link**) — *If a cycle has only two undecided turns that share a common link, then all turns that are adjacent to that link and lie outside the cycle, should be enabled*. This rule can be inferred from Rules 1 and 3. For a cycle with two undecided turns, by Rule 1, one of the two should be disabled. If these two turns are adjacent (*i.e.*, sharing a link), disabling either of the two turns will always involve the shared link. We apply Rule 3 to this link so that we do not allow another cycle to also place its turn restriction on this link. An example is shown on the left side of Figure 3b, where the cycle 1-2-6-5 has two undecided turns after placing the turn restriction at 1-5-4: the turns 1-2-6 and 2-6-5, sharing the link 2-6. This shared link is also adjacent to two other turns: 3-2-6 and 2-6-7 (both belong to the cycle 2-3-7-6), which should be enabled by Rule 4. Note that Rule 4 can be applied to enable turns both on the horizontal and vertical directions. Also, it can be applied iteratively, to further reduce the disabled-turn-placement search space.

Rule 5 (**opposite-corner turn**) — *If two turns are located on opposite nodes in a cycle, and the two turns are not adjacent to any of the cycle's links, then only one of them can be disabled.* We say that such two turns are in opposite-corner locations. The reasoning behind this rule can be understood by considering the example on the right part of Figure 3b: if we had disabled both the opposite-corner turns 1-5-4 and 11-10-14, then we could no longer avoid a deadlock. Indeed, by Rule 2, only two turns would remain undecided in the central cycle (5-6-10-9): 5-6-10 and 5-9-10, and by Rule 1, we would have to disable one of them. However, disabling either of these turns creates a cyclic dependency that could lead to deadlock. As an example, the figure shows the cyclic dependency we obtain if we disable turn 5-9-10, as well as the two opposite-corner turns. Note that it is possible to apply Rule 5 repeatedly by considering increasingly larger cycles. For instance, the rule could be applied to the cycle 5-7-15-13, and force the south-east turn on node 15 to be enabled (assuming that we had a larger network where that turn existed).

4.1 Turn-enabling Rules in Faulty Topologies

Among the basic rules, Rule 1 is the one that is affected the most by faults: cycles may become merged because of faults, as shown in the example of Figure 4a (link 1-4 is faulty). In the figure, the two cycles 0-1-4-3 and 1-2-5-4 no longer exist, and they are merged in the cycle 0-1-2-5-4-3. Moreover, faulty networks may have both concave and convex cycles (unlike fault-free mesh networks, which have only convex cycles) and Rule 1 must be appropriately applied in the case of concave cycles. When a turn is common to two cycles (*e.g.*, turn 3-4-1 in Figure 4b), then disabling that turn can only be counted towards breaking one of the two cycles, not both.

Rules 2 and 3 remain unchanged for faulty topologies. Consider link 3-4-5 in Figure 4a as an example: if we were to disable the turn 2-5-4, Rule 3 would require the turn 5-4-7 to be enabled.

In addition, we limit the application of Rule 4 to links contributing to cycles that have not been affected by faults. For instance, with reference to the left network in Figure 3b, if the link 5-6 were to be faulty, we would not apply Rule 4 to link 2-6, because it contributes to a cycle that has been opened by a fault; but we would still apply the rule to link 8-9. The reason we limit the application of Rule 4 is that it could become complex to identify which turns should be enabled when a link spans multiple routers.

Finally, we simply apply Rule 5 as we described for regular meshes. Applying this rule in faulty networks allows us to aggressively prune the search for an optimal disabled-turn placement. However, some turns enabled are not necessarily causing deadlock in faulty networks, and thus they should not be enabled. If the enabled turns should have been disabled, our backtracking step (Section 5.3) would correct the situation. To avoid the backtracking, it is also possible to pre-emptively check whether a turn enabled via Rule 5 could lead to a deadlock configuration. Figure 4c shows how to check deadlock for this purpose: when there are two distinct cycles connected through a path starting at node 1 and ending at node 2, we cannot disable both the turns indicated in the figure, because this placement would enable the deadlock cycle shown in gray.

5. FATE ROUTING

In this section, we propose FATE's heuristic algorithm to identify a routing function with deadlock-free routing and minimal routing restrictions. FATE relies on the information it receives about the application's communication patterns to strive to place turn restrictions on low-traffic links. When the FATE algorithm begins, all turns are undecided. Turn restrictions are then placed one at a time, starting from the regions transferring the most traffic. Upon placing each restriction, the turn-enabling rules are applied to enable the related set of turns. This process is repeated until each turn is either enabled or disabled.

To identify which turn to disable next, we first estimate the traffic load on each link, turn and cycle. We then disable the turn that minimally worsens the heaviest load-transferring link, choosing among turns on the heaviest load-transferring cycle. The intuition behind this choice is that we want to resolve the routing function first around the regions (*i.e.*, cycles) transferring the most traffic, so that we have plenty of flexibility in our choices. Within each region, we want to disable the turn that minimally affects the congestion in the hotspot.

5.1 Link, Turn and Cycle Load Estimates

To estimate the load on each network's link, turn and cycle, we consider one source-destination pair provided by the application at a time. For each pair, we compute all the possible paths that packets can take from source to destination, and we then derive the fraction

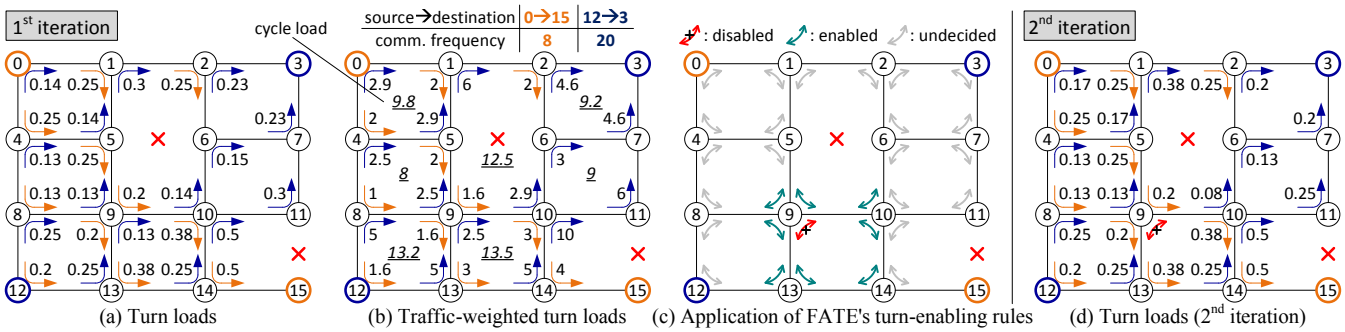


Figure 7: **FATE's routing algorithm example**. We show the first two iterations for a network with two failed links and an application with two communicating pairs. (a) Computation of turn loads. (b) Communication weights and cycle loads indicate that 10-9-13 is the most promising turn-disabling location. (c) Eight turns are then enabled by our turn-enabling rules. (d) Loads computation for the second iteration.

of traffic that would go through each link. The computation of all the loads proceeds with the four steps below (see Figure 5).

Step 1 — Compute path diversity. We calculate the number of different routes (*i.e.*, path diversity) to reach each link from the source node. For instance, in Figure 5a, the east link of node 9 can be used by two different routes from the node 12: 12→8→9 and 12→13→9, while only one route can use the north link of node 9: 12→13→9. In this process, we only allow minimal-length routes within the limits of the turn restrictions that are already in place.

Step 2 — Compute link-load estimates. We now use the results of Step 1 to estimate the load on each link based on the path diversity available. We calculate the total path diversity at each hop from the source (as illustrated in Figure 5a), and divide the link's path diversity by the total diversity. Figure 5b shows the computation for all the links associated with node 9.

Step 3 — Compute turn-load estimates. To estimate the load at each turn, we divide the input load from the source direction of the turn by the number of output links allowed for that source. Figure 5c shows the computation for all the turns at node 9.

Step 4 — Compute cycle-load estimates. For each cycle, the load is computed by simply summing the loads on all the turns in the cycle.

5.2 FATE Route-calculation Algorithm

Once all load estimates have been computed, we can apply the FATE routing algorithm, as illustrated in Figure 6. Note that we weigh the load estimates by multiplying each estimate by the traffic weight associated to its source-destination pair. The algorithm starts by selecting a turn to disable, choosing the turn with the lightest impact on the heaviest link load, among those in the highest-load cycle (lines 2-4). Once the turn to be disabled is selected, we apply the turn-enabling rules to enable as many other turns as possible (line 5). If the set of turns enabled/disabled leads to a deadlock or a disconnected network (line 6), we backtrack, and update the list of conflicting selections (line 8). Our backtracking algorithm is discussed in more detail in the next subsection.

In designing our routing algorithm, we evaluated a few other strategies to select the next turn to be disabled: beside the one just described, we also tried (1) the turn with the absolute lowest load in the network, (2) the turn with the lowest load among those in the highest-load cycle, and (3) the turn connected with the highest-load link in the highest-load cycle. Experimentally, we found that those strategies performed slightly worse than the one we described.

```
1: repeat until there is no undecided turn
2:    compute_link,turn,cycle_loads()
3:    cycle = cycle_with_heaviest_load()
4:    disabled_turn = turn_with_smallest_link_load_increase(cycle)
5:    enabled_turns = apply_turn_enabling_rules(disabled_turn)
6:    if( not (check_deadlock() or check_disconnected()) )
7:       disable(disabled_turn), enable(enabled_turns)
8:    else update_conflict_history(), backtrack()
```

Figure 6: **FATE routing algorithm**

Figure 7 illustrates the algorithm with an example. The application provided two communication pairs: 0→15 (shown in orange), with a weight of 8, and 12→3 (shown in blue), with a weight of 20. We first compute link and turn loads, as shown in Figure 7a. Then we apply the weights and compute cycle loads in Figure 7b. Cycle 9-10-14-13 is the one with the highest load. By analyzing each turn, one at a time, we find that the one that entails the smallest link-load increase is 10-9-13, so we disable it. Figure 7c shows the network after the application of the FATE's turn-enabling rules. At this point, 21 turns are left undecided, thus we start a second iteration by updating the link, turn and cycle load estimates as shown in Figure 7d.

5.3 Avoiding Illegal Routing Function (Backtracking)

As mentioned earlier, the FATE routing algorithm may require backtracking if the set of turn restrictions in place allows for deadlock (usually along a complex cycle) or disconnects the network. These issues may arise because the FATE's rules do not take into account all the implications of a turn restriction, but only the simpler ones, so that their application is computationally cheap. When these situations occurs, we record the location of all restrictions and we add the current set of turn-disabling placements to a *conflict history*, which we use to avoid repeating the same configurations. While the majority of topologies and communication patterns in our experiments have successfully completed with only a small amount of backtracking, we found a few cases (less than 1%) that result in more than 1,000 iterations to finish. We believe these situations occurred because of our simplistic backtracking model that rolls back to the most recent decision, instead of a more intricate model that selects a promising roll-back point. This limitation is partially contained by a random restart technique that we implemented, similar to that in some SAT solvers. Upon a restart trigger, previous decisions are forgotten, and a new initial turn location is selected. We set the restart threshold to 1,000 backtracking events in our experiments.

6. EXPERIMENTAL EVALUATION

We evaluated FATE with a cycle-accurate NoC simulator [15] modeling an 8×8 2D mesh NoC. The network's 3-stage routers are capable of look-ahead routing and speculative switch allocation (*i.e.*, switch allocation is concurrent with VC allocation). Each input port within a router contains 2 VCs per protocol class, and 5 flits per VC, unless otherwise noted. We utilized 3 protocol classes to support the

MESI cache coherence protocol when running SPLASH-2 benchmarks, and a single protocol class when evaluating synthetic traffic patterns. We apply the FATE algorithm to place turn restrictions, then deploy a minimal adaptive routing approach where packets choose at each router which direction to take among those enabled. In addition, we deploy a local congestion monitoring scheme based on the credit count, so that output channels towards non-congested input buffers are favored.

We analyzed FATE's performance both on fault-free and faulty 2D mesh networks, injecting a varying number of link faults in the latter scenario. Specifically, we experimented with configurations containing 1 (1%), 3 (3%), 6 (5%), 11 (10%) and 17 (15%) faulty links, and averaged the results over 10 different random sets of fault placements for each data point. Note that we do not consider configurations that lead to disconnected nodes, as this situation would require thread migration support when running parallel benchmarks. This requirement, in turn, would make it difficult to reason about performance scaling due to a different number of active cores in different configurations. Even though we model only link failures, FATE is equally effective in tackling routers' logic failures by mapping such failures to one or more links connected to the failed routers [20]. Please note that we evaluated our solution under various corner-case topologies to take into account unpredictability of fault locations. These random topologies include, for instance, nodes with only a single surviving link, cycles consisting of more than 10 nodes, *etc*.

We evaluated our testbed with both synthetic traffic [15] and traces from the SPLASH-2 benchmark suite [25]. We used 5 different synthetic patterns: bit complement (*bitcomp*), bit reversal (*bitrev*), shuffle (*shuffle*), transpose (*transpose*) and uniform random (*uniform*). Our synthetic traffic consists of a mix of equal amounts of 1- and 5-flit packets. The 11 SPLASH-2 traces we experimented with, on the other hand, were obtained from full-system simulation using gem5 [6] in the *syscall* emulation mode. Our gem5 model was configured as shown in Table 1. The traces were collected for 10 million cycles after spawning threads and initializing caches. The traffic weights were calculated by counting the number of flits per each source-destination pair. The latency values were capped at 1,000 cycles during the trace-based simulation to avoid extremely large latencies due to network saturation. We did not evaluate FATE with the PARSEC benchmark suite, because PARSEC is known to underutilize the network, and its traffic is more evenly-distributed than SPLASH-2. We expect PARSEC to yield results similar to those of *uniform*.

Table 1: **gem5 configuration for SPLASH-2 traces**

core	64 cores, x86 ISA, 2GHz, out-of-order, 8-wide issue
cache coherence	MESI, CMP directory, 64-byte block
network	8×8 mesh, 3 classes/port, 2 VCs/class, 5 flits/VC
L1 cache	private, 16KB inst. + 16KB data, 4-way set, 3-cycle lat.
L2 cache	shared, distributed, 256KB/node, 16-way set, 15-cycle lat.
memory	1GB/controller; 1 controller at each mesh corner

6.1 Performance on Faulty Networks

We compared our solution against fault-tolerant, application-oblivious routing solutions that are based on the construction of spanning trees: up*/down* with breadth-first search (BFS) [1,23], and up*/down* with depth-first search (DFS) [22]. For BFS, we experimented with 4 different configurations: each configuration placed the root node at a different corner of the mesh. Then we calculated the geometric mean of the results obtained over all four configurations. For DFS, we implemented the two heuristics from [22]: *Ht1* for selecting the root node, and *Fv* for choosing a tree-spanning direction.

We also compared our solution against two existing application-aware routing solutions: Application-Specific Routing Algorithm (APSRA) [19] and Bandwidth-Sensitive Oblivious Routing (BSOR) [16]. Both algorithms are modified for distributed routing. For APSRA, we applied APSRA's turn cost calculation instead of our load estimation method in Section 5.1. For BSOR, we first placed turn restrictions according to the negative-first turn model, then applied the Dijkstra's shortest path algorithm. We tried four different turn restrictions, each with four rotations. The algorithm is then applied iteratively by reducing the link capacity to aggressively optimize for congested links, as shown in [16].

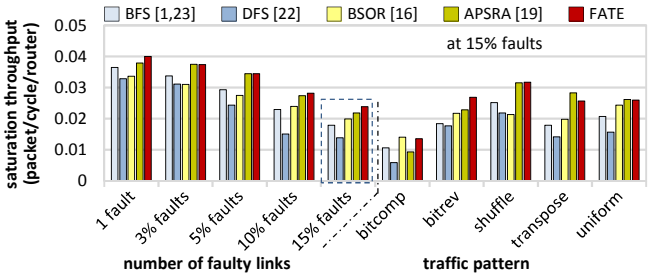


Figure 8: **Saturation throughput for synthetic traffic patterns over various fault rates and traffic patterns**. FATE provides 10%-33% better saturation throughput over BFS, and up to 9% better throughput over APSRA.

Figure 8 reports the average saturation throughput (*i.e.*, when latency reaches 3 times the zero-load latency) across various fault rates and traffic patterns. In the left half of the figure, we observe that the performance of our scheme degrades more gracefully than both application-oblivious spanning-tree solutions (BFS and DFS) as faults increase. FATE achieves a 10% improvement over BFS when there is only one faulty link. Although the network at this low fault rate maintains an almost-regular topology, FATE still offers a better throughput than BFS by leveraging distinct traffic patterns. FATE's improvement over BFS increases to 33% when 15% of the links are faulty. At this high fault rate, the spanning-tree solutions often fail to ensure minimal routing restrictions, leading to high performance loss.

FATE also achieves consistently higher throughput at all fault rates than both APSRA and BSOR. While it shows comparable throughput with APSRA at low fault rates, FATE provides a 9% higher throughput at the 15% faulty-link rate. We believe that this is because APSRA's traffic estimation becomes inaccurate at high fault rates, as the estimation relies on the path diversity between sources and destinations using source routing. In contrast, FATE estimates traffic load by considering hop-by-hop routing-decision using distributed routing. Finally, BSOR provides lower throughput than both APSRA and FATE, most probably because it lacks a dynamic approach to congestion management.

FATE also performs better than BFS across various traffic patterns at the 15% faulty-link rates, as shown in the right half of Figure 8. We observe that traffic patterns where packets utilize a few turns more frequently (*e.g.*, *transpose* and *bitrev*) are those that benefit the most from FATE. Our solution provides at least at-par performance in the patterns where turns are used evenly (*e.g.*, *uniform*).

Figure 9 reports the average packet latency from our trace-driven SPLASH-2 simulations across various fault rates and benchmarks. As shown in the left half of the figure, FATE experiences negligible latency increase up to the 5% faulty-link rate. Beyond that point, the latency increase is more significant. However, note that how the latency increase begins at lower fault rates: it is much steeper in BFS and DFS routing. This dramatic increase comes from the earlier saturation effect in resource-constrained faulty networks. Note also that in networks with only one faulty link, FATE performs worse than other solutions. This result is due to the high-impact contribution to the average by *ocean_con*, which exhibits phases with very high injection rates for short time periods. These phases are not

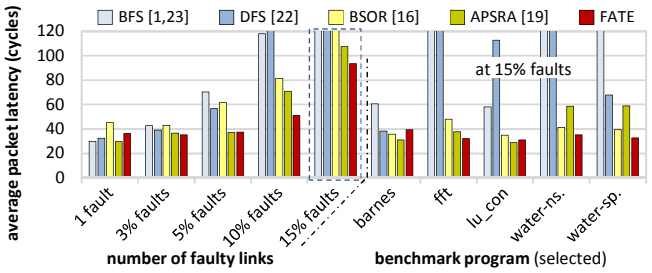


Figure 9: **Packet latency for SPLASH-2 traces over various fault rates and benchmarks**. Except for the 1-fault case, FATE shows 18%-59% improvements in packet latency over BFS, and up to a 13% improvement over APSRA.

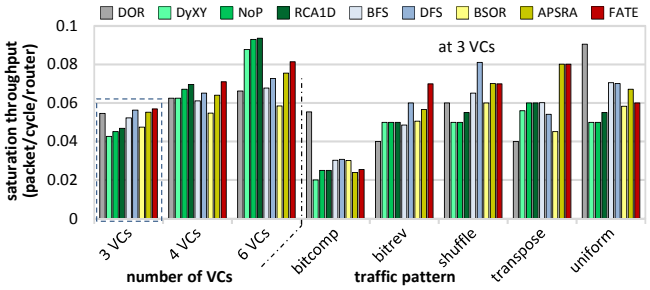


Figure 10: **Saturation throughput for synthetic traffic patterns over a varying number of VCs and traffic patterns**.

representative of the entire benchmark execution, and hence our solution is unable to optimize for them. In addition, FATE attains lower latency than APSRA at all fault rates except one faulty link. The two solutions show comparable latency until the 5% faulty-link rate. At the 15% rate, however, FATE shows 13% lower packet latency than APSRA. FATE also outperforms BSOR at all fault rates.

Finally, we show results for five selected benchmarks at the 15% faulty-link rate on the right side of Figure 9. FATE consistently provides much lower latency than BFS and DFS. For instance, when running *fft*, each node accesses frequently memory nodes located at the corners of the mesh, and communicates with a few other nodes. As a result, FATE enables more routes among these frequently communicating nodes. On the other hand, both BFS and DFS are application-oblivious, so their disabled-turn placements are not so favorable to those nodes.

6.2 Performance on Fault-free Networks

We compared FATE's fault-free operation and congestion management capabilities against 3 fully-adaptive routing techniques: DyXY [17], NoP [2] and RCA1D [12]. For those solutions, we implemented deadlock detection based on timeout, and reserved one VC for deadlock recovery. We also considered prior fault-tolerant solutions and application-aware solutions for fault-free networks.

Figure 10 shows the average saturation throughput of FATE against all the solutions above across 3 different VC settings (3, 4 and 6 VCs), and synthetic traffic patterns. As shown in the left part of the figure, FATE outperforms DOR: FATE achieves a 4.5% improvement when using 3 VCs, and up to 23% with 6 VCs. FATE's advantage grows with the number of VCs since it manages resources more efficiently than oblivious routing techniques.

However, the reverse trend holds when FATE is compared against fully-adaptive solutions, *i.e.*, DyXY, NoP and RCA1D. FATE's improvement over the fully-adaptive solutions diminishes as the number of VCs increases, because the cost of reserved VCs in the fully-adaptive solutions is amortized when more VCs per class are available. For networks with only 3 VCs, FATE shows a 33% improvement over DyXY and a 21% improvement over RCA1D. This advantage is lost at 4 VCs, while at 6 VCs FATE provides 13% lower throughput than RCA1D. In the area- and power-constrained on-chip environment, NoCs with fewer VCs are more prominent, and hence our solution is widely applicable.

We further analyze FATE's fault-free performance by considering various synthetic traffic patterns. The right part of Figure 10 shows the average saturation throughput for networks with 3 VCs under various traffic patterns. Our solution outperforms the fully-adaptive solutions for most patterns in this setting: the exceptions are *bitcomp* and *uniform*, where DOR is the best performer. Considering the benefits of FATE for faulty networks (Section 6.1), we believe that a slight performance loss on fault-free networks for rare adverse traffic patterns is acceptable.

Note that in our experiments, FATE leverages local congestion information to adaptively choose better routes at runtime. However, it has been shown in prior research [2,12] that global network-level congestion monitoring schemes deliver superior performance. In future work, we plan to evaluate the benefits of applying some of these schemes to FATE.

6.3 Overheads

Routing-function computation overhead. We evaluated the overhead of computing the routing function, and compared our findings against APSRA [19]. Table 2 reports the average computation time to derive a routing function, for both FATE and APSRA. In evaluating computation time, we only included the total time spent in estimating traffic load (Section 5.1 for FATE), since that is by far the major contributor to the algorithm's computation time, for both ASPRA and FATE. The other activities contributing to the routing function computation (*e.g.*, backtracking, deadlock checking, *etc.*) are minor contributors and identical for both solutions. Execution times were measured by averaging over 5 executions for each routing function computation on an Intel Xeon E5520. Overall, it can be noted that FATE is a significantly faster solution than APSRA.

In the right portion of Table 2, we also compare the number of attempts of turn-disabling placement, averaged over 10 different faulty topologies and 16 traffic weights (5 synthetic traffic patterns and 11 SPLASH-2 benchmark traces) at each fault rate, in order to gain insights on the gap between the computation of FATE and APSRA. This value is the cumulative sum of each turn-disabling placement attempt, including all the placements that had to be removed because they led to a conflict or deadlock. Note that FATE's number of attempts is minuscule compared to APSRA: this result comes from our turn-enabling rules, which greatly prune the search space for an optimal set of turn-disabling placements. In many situations, APSRA's routing algorithm made extremely large turn-disabling placement attempts before reaching a stable solution. We capped those algorithm's runs at 200,000 placement attempts and we report the fraction of occurrences where we reached the cap value. Note that FATE never had a case that required 200k placement attempts, while APSRA had a noticeable fraction of algorithm's runs that went over the limit.

Table 2: **computation overhead for FATE and APSRA**

	average time (sec)		average number of placements attempted		% runs reaching 200k cap	
	APSRA	FATE	APSRA	FATE	APSRA	FATE
fault-free	>500	3.61	71,321	117	19%	0%
1 fault		3.27	94,639	107	31%	0%
3% faults		3.29	107,956	107	41%	0%
5% faults		3.21	120,877	105	48%	0%
10% faults		3.62	151,802	118	69%	0%
15% faults		2.93	159,667	96	74%	0%

Although FATE requires much less computation than existing application-aware routing, it may still not be sufficiently fast because of its software-based computation. In those situations where recovery performance is of essence, alternative hardware-only solutions (*e.g.* retransmission [18] and BFS-based routing [1]) can be deployed concurrently with FATE, while it executes in software. Upon FATE's completion, its solution can replace the interim recovery solution.

Area and power overhead. FATE requires a reconfigurable routing infrastructure (*e.g.*, routing tables) to recompute the routing function at runtime. We deploy routing tables as shown in [1,16], one for each router. Each table contains N entries where N is the number of nodes, and each entry contains four directional 2-bit fields (8 bits per entry). The 2-bit fields are used to prioritize valid output directions by using the number of hops to the destination. In addition, we utilize four routing-restriction bits (similar to [7]) to avoid making invalid decisions at each router. Thus, to store the computed routing function in memory, we require $N \times (N \times 8+4)$ bits per application of the FATE algorithm. Moreover, we use the number of used credits as our congestion metric: this is often already available in routers and comes at no extra cost.

We evaluated the area overhead of routing tables and route-computation logic, targeting the Nangate 45nm library using Synopsys DC. The micro-architecture of a baseline router is configured as specified in Table 1 with an operating frequency of 400MHz. With this configuration, our routing computation adds approximately 6% area overhead, mostly for the routing table. Note that the other fault-tolerant, adaptive routing solutions we compared against (BFS, DFS and APSRA) entail similar overhead, as they utilize similar routing intrastructures as FATE.

While we have not evaluated FATE's power overhead, we provide here a qualitative comparison. Our solution entails significantly less computation than APSRA while, at the same time, producing routing functions that lead to lower packet latency. Thus we believe FATE would consume less power than APSRA. When comparing to BSOR, BFS and DFS, the two trends are in opposition: we do attain lower packet latency, at the cost of higher computation time for the routing function. Thus we only provide an overall power gain if we can absorb the additional power cost of the computation over the benefits in packet latency.

7. CONCLUSIONS

Maintaining high-throughput and low-latency after faults have occurred is critical to the continuous deployment of NoCs on unreliable silicon substrates. We proposed FATE, a high-performing, adaptive routing solution for faulty networks-on-chip, that leverages the knowledge of an application's communication patterns. We developed turn-enabling rules to quickly determine the optimal set of turns that should be disabled to break all deadlocks, while still providing connectivity in faulty 2D meshes. Our application-aware heuristic balances load evenly among network resources using our novel load-estimation metrics, and chooses the most promising turn-restriction locations. Experimental results show improvements in throughput and latency for synthetic traffic patterns and SPLASH-2 benchmark traces over state-of-the-art fault-tolerant and application-aware routing solutions. Finally, FATE keeps a low cost profile (6% area overhead), while providing a better routing performance than prior application-aware routing. We also showed that our solution is a viable, low-cost, adaptive-routing alternative even for fault-free networks when compared to fully-adaptive routing solutions.

Acknowledgements. This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

8. REFERENCES

[1] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco. Ariadne: agnostic reconfiguration in a disconnected network environment. In *Proc. PACT*, 2011.

[2] G. Ascia, V. Catania, M. Palesi, and D. Patti. Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. *IEEE Trans. Computers*, 57(6), 2008.

[3] M. Badr and N. Jerger. SynFull: synthetic traffic models capturing cache coherent behavior. In *Proc. ISCA*, 2014.

[4] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of Splash-2 and Parsec. In *Proc. IISWC*, 2009.

[5] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. PACT*, 2008.

[6] N. Binkert *et al*. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[7] J. Cano *et al*. Efficient routing in heterogeneous SoC designs with small implementation overhead. *IEEE Trans. Computers*, 63(2), 2014.

[8] J. Cong, C. Liu, and G. Reinman. ACES: application-specific cycle elimination and splitting for deadlock-free routing on irregular network-on-chip. In *Proc. DAC*, 2010.

[9] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers*, C-36(5), 1987.

[10] J. Flich *et al*. A survey and evaluation of topology-agnostic deterministic routing algorithms. *IEEE Trans. PDS*, 23(3), 2012.

[11] C. Glass and L. Ni. The turn model for adaptive routing. In *Proc. ISCA*, 1992.

[12] P. Gratz, B. Grot, and S. Keckler. Regional congestion awareness for load balance in networks-on-chip. In *Proc. HPCA*, 2008.

[13] J. Henkel *et al*. Reliable on-chip systems in the nano-era: lessons learnt and future trends. In *Proc. DAC*, 2013.

[14] C. Iordanou, V. Soteriou, and K. Aisopos. Hermes: architecting a top-performing fault-tolerant routing algorithm for networks-on-chips. In *Proc. ICCD*, 2014.

[15] N. Jiang *et al*. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proc. ISPASS*, 2013.

[16] M. Kinsy *et al*. Optimal and heuristic application-aware oblivious routing. In *IEEE Trans. Computers*, 2013.

[17] M. Li, Q.-A. Zeng, and W.-B. Jone. DyXY: a proximity congestion-aware deadlock-free dynamic routing method for network on chip. In *Proc. DAC*, 2006.

[18] S. Murali *et al*. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 22(5), 2005.

[19] M. Palesi *et al*. Design of bandwidth aware and congestion avoiding efficient routing algorithms for networks-on-chip platforms. In *Proc. NOCS*, 2008.

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

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

[22] J. Sancho, A. Robles, and J. Duato. An effective methodology to improve the performance of the Up*/Down* routing algorithm. *IEEE Trans. PDS*, 15(8), 2004.

[23] M. Schroeder *et al*. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal of Selected Areas in Communications*, 9(8), 1991.

[24] A. Shafiee *et al*. Application-aware deadlock-free oblivious routing based on extended turn-model. In *Proc. ICCAD*, 2011.

[25] S. Woo *et al*. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, 1995.