Correct Runtime Operation for NoCs through Adaptive-Region Protection

Rawan Abdel-Khalek and Valeria Bertacco
Computer Science and Engineering Department, University of Michigan
(rawanak,valeria)@umich.edu

***Abstract*—Networks-on-chip (NoCs) are increasingly being adopted as the interconnect model for systems-on-chip and chip-multiprocessors. As the only communication medium in these designs, the NoC's functional correctness is critical. In practice, design-time verification of NoCs is always partial, due to their large scale and the challenges that hinder verification efforts. As a result, functional design bugs are bound to escape and potentially manifest at runtime, compromising system functionality.**

We propose REPAIR, a runtime solution to detect and recover from functional design errors that have escaped in NoCs. Existing runtime verification techniques incur significant area and performance overheads to monitor and check the correctness of every packet traversing the network. However, REPAIR relies on a retransmission-based technique that adaptively determines the subset of packets requiring protection by identifying dynamic network regions where the specific runtime execution is likely to expose functional design bugs. We achieve runtime correctness at lower performance and area costs, relative to a traditional solution: on average, we are able to achieve more than 50% better overall performance with 2-3x fewer retransmission buffers.

I. Introduction

Network-on-chip (NoC) interconnects are the central medium of communication in many current and future homogeneous and heterogeneous chip-multiprocessors (CMPs) and systems-on-chip (SoCs). Therefore, the NoC's functional correctness is critical, and it requires extensive verification, both as a stand-alone fabric and as part of the entire system. Mainstream verification efforts initially target the correctness of the router architecture. However, the correctness of the full interconnect cannot be presupposed by verifying the operations of just one router or a small subset of routers connected together. The network's overall execution is the result of system-level interactions that occur as responses to the traffic patterns traversing the network. These interactions could span a small region of the network or the entire network in some cases. Moreover, the correctness of the routing and communication protocols and their implementations can only be observed by considering the network as whole. For example, some errors, such as deadlocks and livelocks, can only manifest at the network-level. Thus, the complete verification of the NoC requires modeling the entire network.

The continuously increasing size and design complexity of NoC interconnects poses significant challenges to design-time verification efforts. Today's CMPs include up to a 100 cores that rely on an equally large NoC interconnect for communication. Moreover, aggressive interconnect designs are adopted to provide enhanced performance and energy efficiency. This growing complexity is evident when considering the expanding repertoire of features that are incorporated into the architecture of routers, including intricate arbitration mechanisms, speculation, and adaptive routing implementations. There is also an additional layer of complexity due to the implementation of advanced routing protocols, power and application-adaptive interconnects and complex overlaying communication protocols. This high design complexity creates a large design state space, which cannot be thoroughly exercised and validated during design-time verification, with the result that interconnect designs may be released into the market with corner-case bugs latent in those unverified executions. Those bugs become problematic if the unverified functionality is exercised at runtime, while the system is being utilized by the user, and may compromise the entire system, the applications running on it and the user's data. Additionally, design houses are then forced to release fixes, or in some cases recall the product alltogether, all of which result in extensive monetary losses. One technique to protect NoC interconnects from the manifestation of latent bugs consists of equipping them with mechanisms to detect and recover from erroneous behavior at runtime.

We propose REPAIR (**R**untime **E**rror **P**rotection over **A**daptively **I**dentified **R**egions), a solution to ensure communication correctness in NoC interconnects at runtime (after a product's release). REPAIR targets functional design bugs that have escaped verification and that manifest at runtime in the NoC. Such bugs tend to be well-hidden and are only exposed when the runtime execution triggers a specific complex set of events or corner-cases that were never tested or observed before the product's release. At runtime, applications running on CMPs and SoCs typically have spatially and temporally varying traffic patterns, such that not all portions of the network encounter the same set of runtime operations, and throughout an application's execution, some regions exhibit runtime conditions that are more likely to expose latent design bugs. REPAIR dynamically identifies those regions and it employs a detection and recovery mechanism to protect packets traversing them. Before a packet enters an error-prone region, it is copied into a retransmission buffer within the network interface of the routers at the periphery of those target regions. This packet is then marked as an acknowledgment_required (*ack_required*) packet and is transferred forward, towards its destination. Upon correct delivery to its final destination, an acknowledgment is sent back to the intermediate node that created the copy and the retransmission buffer is freed. If the intermediate node times-out before receiving an acknowledgment, it initiates REPAIR's recovery scheme. During recovery, routers independently drop all *ack_required* in-flight packets and retransmit a copy of them from the retransmission buffers, in which they are stored. The manifestation of design bugs at runtime is an infrequent occurrence, as bugs are only exposed during specific corner-case execution scenarios that were not verified. By clearing in-flight *ack_required* packets and retransmitting them, the network is extremely unlikely to encounter again those exact same conditions that triggered the original bug. Figure 1

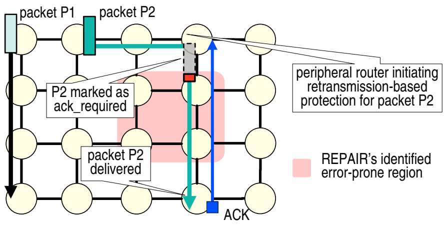
packet P1
packet P2
peripheral router initiating retransmission-based protection for packet P2
P2 marked as ack_required
packet P2 delivered
REPAIR's identified error-prone region
ACK

Fig. 1. **High-level overview of our runtime solution.** REPAIR identifies network regions exhibiting operations prone to exposing latent design bugs, and protects packets traversing these regions from being affected by the bugs.

depicts a high-level overview of our solution. There, packet P2 is traversing an identified error-prone network region, and it is protected by REPAIR's selective-retransmission.

Example. Consider the case of a deadlock occurring at runtime in an error-prone region, due to a bug in the implementation of the routing algorithm. For the deadlock to have happened, it means that the runtime execution encountered a precise sequence of events that allowed the deadlocked packets to arrive to a specific set of congested routers and request specific resources with the precise timing that caused the cyclic dependency to form. This exact scenario is unlikely to be easily replicated, otherwise this bug would probably have been caught earlier during development-time verification. In a network equipped with REPAIR, routers are instrumented with timeout counters. In a deadlock situation, at least one counter will timeout, causing its corresponding router to initiate REPAIR's recovery scheme. During recovery, routers drop in-flight ack_required packets, which clears the deadlock. Then, each router independently retransmits a copy of the packets residing in its retransmission buffers. The network conditions, timing of packet injections and transfers, and the availability of resources are likely to have changed upon retransmission, highly reducing the chances of triggering the same corner-case behavior that caused the deadlock to occur.

REPAIR'S acknowledgment-retransmission is inspired by traditional retransmission solutions, where a copy of every packet is stored at the source node, before the packet is injected. However, there are several key differences in our solution. First, copies are made for only a small subset of packets, and any intermediate node along a packet's path may decide to be the one that initiates protection via a copy. Second, traditional retransmission solutions recover by retransmitting only the timed-out packets, and thus cannot overcome all types of design bugs, such as deadlock-type bugs. In contrast, REPAIR employs a network-level recovery scheme to clear the effect of the bug before retransmitting all ack_required packets.

A. *Contributions*

- We introduce a novel solution to ensure correct communication during runtime execution in the presence of a network-on-chip interconnect with escaped design bugs.
- We adaptively identify dynamic network regions where the runtime conditions are prone to exposing design bugs.
- We deploy a lightweight acknowledgment-retransmission solution to protect packets traversing the identified error-prone network regions. By applying acknowledgment-retransmission to only a small subset of packets injected into the network, we maintain a low area overhead and a lower performance impact, relative to a traditional source-based retransmission solution.

II. Error-prone Network Regions

The design-time verification of NoC interconnects initally focuses on verifying the correctness of the NoC's most fundamental operations, which often occur when only a small number of nodes are injecting traffic into the network, consequently generating simpler network operations and interactions. Once this initial verification is complete, additional randomly-generated traffic patterns and application traffic are run to exercise the more complex execution scenarios that could occur in the network. In practice, the majority of design bugs uncovered during the development of NoC-based designs tend to manifest in complex behavior involving a large number of simultaneous interactions between the design's components [1]. However, as the number of active nodes and the contention among in-flight packets increase, more elaborate interactions begin to occur between packets and within network components. As a result, the design space spanned by all possible NoC operations grows exponentially and becomes intractable. Thus, design-time verification efforts focus on validating the fraction of these executions that represents the most common network behavior. The remaining susbet of complex executions are not excercised nor validated, and it is precisely in those scenarios that design bugs remain hidden.

Once the system is released, a large number of users execute a vast spectrum of different applications, stimulating the NoC much more thoroughly than it was possible during development. Note that, during an application's execution, different regions of the network experience different patterns of activity, depending on the application demands. Lightly loaded network regions have a few packets traversing them, and hence observe limited contention between in-flight traffic. Therefore, the operational conditions in these regions are among those simple execution scenarios that are well-verified. Whereas, network regions with intense activity, involving a larger number of contending packets and multiple injecting nodes exercise some of the more complex operations, which may have not been exhaustively verified. Therefore, the execution conditions in those regions are prone to exposing latent design bugs.

Instead of deploying a costly runtime solution that is active throughout execution and for all regions of the network, we propose an adaptive scheme that identifies error-prone regions and protects only the packets traversing these regions. Since design bugs are likely to be hidden in more complex executions, we seek to identify network regions with high activity and intricate operations and we consider those regions to be error-prone. To this end, we rely on congestion as an indication of operational complexity. Congestion occurs due to an increased traffic load in the network and, although it does not imply a functional error, it directly relates to the complexity of the execution. In particular, highly congested network routers are traversed by a large number of packets that are contending for the available resources. This contention and the number of active input and output channels within these routers create more intricate interactions within the routers and across them. Therefore, while operating under this more demanding stimuli, the congested routers are more likely to encounter a corner-case execution that has not been verified. Whereas, uncongested routers that are exhibiting simpler traffic flows are more likely to be operating within the well-verified subset of executions. Moreover, in some cases, congestion

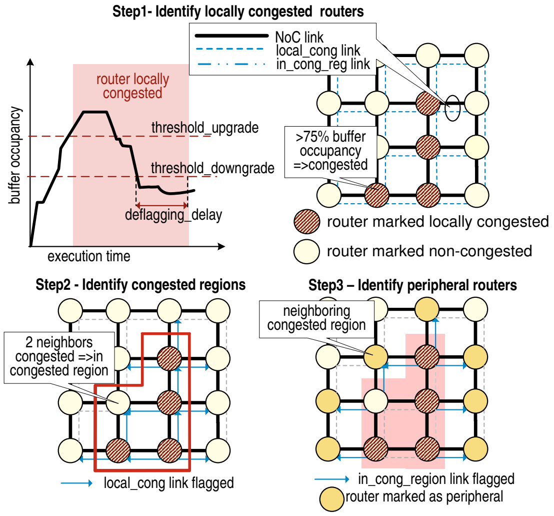
Step1- Identify locally congested routers
NoC link
local_cong link
in_cong_reg link
router locally congested
buffer occupancy
threshold_upgrade
threshold_downgrade
deflagging_delay
execution time
>75% buffer occupancy =>congested
router marked locally congested
router marked non-congested
Step2 - Identify congested regions
2 neighbors congested =>in congested region
local_cong link flagged
Step3 – Identify peripheral routers
neighboring congested region
in_cong_region link flagged
router marked as peripheral

Fig. 2. **Identifying congested regions and peripheral routers.**

introduces an additional layer of complexity to the network's operations. For example, CMPs and SoCs commonly employ power management techniques that utilize various approaches, such as injection throttling and frequency scaling, to limit power dissipation when the network is heavily loaded. Finally, congestion plays an important factor in many routing protocols, adapting the routes followed by packets, as well as triggering protocol-level deadlock recovery and congestion avoidance measures. The interactions between such features and the NoC's regular operations can create unforseen and unverified corner-cases that hide design bugs.

A. Identifying Congested Regions

REPAIR adaptively identifies error-prone network regions by relying on a light-weight algorithm that delineates network areas with high congestion, and hence complex runtime activity. Our 3-step algorithm, illustrated in Figure 2, is distributed and iterative, allowing us to dynamically and adaptively classify congested regions. If the network congestion spreads, our algorithm adapts and includes newly congested routers in the classified congested regions. Similarly, as the network load is throttled and congestion is reduced, the classified congested regions move and change accordingly.

Step 1 - Identify locally congested routers. Routers determine their local congestion status, based on their internal buffer occupancy, similar to the scheme described in [2]. Each router keeps track of the number of input buffer entries in-use across all of its input buffers. If the number of occupied entries exceeds a certain fraction, then the router flags congestion. We refer to this occupancy threshold as *threshold_upgrade*. Once the congestion flag is set, it can only be deflagged when the buffer occupancy falls and stays below another threshold for a certain number of clock cycles. We refer to this second occupancy threshold as *threshold_downgrade* and the number of clock cycles as *deflagging_delay*.

Step 2 - Identify congested regions. Once the local congestion flag is set at each router, it is passed to all of its direct neighbors, through a 1-bit bi-directional link added between all routers (*local_cong* link). Then, each router determines whether it belongs to a congested region, based on two criteria: 1) Is the router itself congested? or 2) Is it neighboring two congested routers? The second criteria allows grouping congested routers into more contiguous regions. The result of this evaluation sets a new flag, *in_cong_region*, which is again transmitted to all the first-hop neighboring routers, requiring another 1-bit bidirectional link between each router.

Step 3 - Identify peripheral routers. Each router receives the *in_cong_region* flags from its neighbors. If at least one of its neighbors belongs to a congested region, the router determines that it is a peripheral router.

III. Achieving Communication Correctness

When REPAIR identifies a congested network region, routers at the periphery of the region, and routers within it, initiate our acknowledgment-retransmission scheme to protect packets traversing the congested region. We refer to these routers as the *initiating routers*. As packets are injected into the network, they are augmented with an error-detection code that can be added to their header flits. At an initiating router, packets requiring protection are copied into the router's retransmission buffers. Such packets are marked as ack_required and upon delivery to their destination node, an acknowledgment is sent back to the initiating router. If the router times-out before receiving an acknowledgment, it assumes that an error occurred and transmits a recovery signal across the network through a 1-bit serial link that connects all routers. Afterwards, each router independently starts the recovery process by first dropping all ack_required packets traversing it and then retransmitting a copy of the packets residing in its retransmission buffers. This recovery process does not halt regular execution, as the network's operations proceed normally in conjunction with the retransmissions.

REPAIR can overcome a wide variety of functional design bugs in the network. Independently of the exact bug, the correctness of communication can only be compromised if the bug corrupts in-flight packets or if it prevents their correct and timely delivery to their destination. The latter case includes deadlocks, livelocks, misroutes, and dropping of packets. In REPAIR, the error-detection code that is added to every packet and the timeout functionality at initiating routers are sufficient to flag bugs causing packet corruption and incorrect packet delivery. REPAIR's recovery scheme can also overcome both of these errors categories, even when the retransmitted packets follow the same paths as the original ones. First, by dropping all ack_required packets, REPAIR ensures that the effects of the detected bug are cleared from the network. Second, design bugs during runtime operation manifest rarely and are exposed only under specific execution conditions. During recovery, when all ack_required packets are retransmitted, they are likely to encounter different operational conditions and timings that will not trigger the same bug to re-manifest, allowing REPAIR to side-step the design bug.

A. Creating Packet Copies at Initiating Routers

REPAIR maintains a congestion status table per router to record which of the router's output directions are congested. We also assume that the retransmission buffers at each node reside in the corresponding network interface. Thus, when an initiating router decides to apply acknowledgment-retransmission

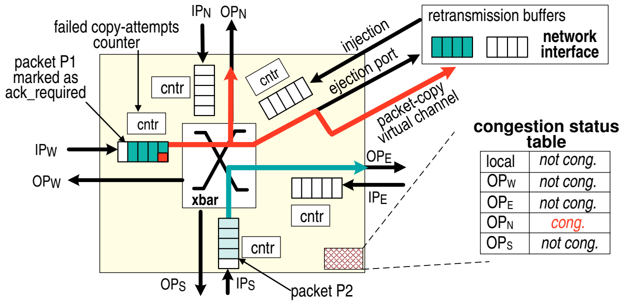

Fig. 3. **Applying acknowledgment-retransmission at a peripheral router.** Since Packet P1 is heading to the North (entering a congested area), it is protected by REPAIR. This peripheral router duplicates P1 and forwards a copy to be stored in one of the retransmission buffers.

to a packet, it must duplicate it and eject a copy to its network interface. The packet is then marked as ack_required and no other router will make another copy of it. If the initiating router is itself in a congested region, it initiates REPAIR's protection for any previously unprotected packet traversing it and for every packet it is injecting into the network. This situation may arise because our congested regions vary during execution, and a router may become marked as congested while packets reside in its buffers. Figure 3 shows a peripheral router applying acknowledgment-retransmission to packet P1.

In a typical wormhole router, packets undergo four main steps: route computation (RC), virtual channel allocation (VCA), switch allocation (SA) and crossbar traversal. Once a packet's output direction is known after the RC step, a look up in the congestion status table allows REPAIR to classify whether this packet requires protection. If the packet requires protection, then, in the VCA step, it must request a virtual channel in its assigned output port and in the ejection port. The packet completes VCA only if both virtual channels have been granted. If all retransmission buffers are full, the packet must stall. REPAIR utilizes a separate virtual channel to eject packet copies to the retransmission buffers, as we discuss in Section III-B. During the SA step, the flits of the packet requiring protection must simultaneously request two crossbar connections, one to its desired output port and another to the ejection port. When both connections are granted, the original flits proceed to their destination output port, and a copy of each flit is ejected to be stored in a retransmission buffer.

B. *Cyclic Dependencies and Deadlock Avoidance*

In REPAIR, we can categorize in-flight traffic as ack_required packets, acknowledgment packets (ACKs), or regular packets. We can further group regular packets into those that are about to be ejected from the network to their destination (ejecting_packets) and packets that are still traversing the network to their destination (traversing_packets). Cyclic dependencies can arise between ack_required packets and the other three types of traffic. In this section, we discuss the situations where these dependencies can lead to deadlocks and we provide mechanisms to overcome them.

ACKs vs. ack_required packets: To prevent the blockage of in-flight ACKs behind stalled packets waiting for available retransmission buffers, we utilize a separate virtual channel for ACKs. This virtual channel can be designed with fewer buffering resources than the regular virtual channels of the network, as ACKs are only 1-flit long. The separation of acknowledgment packets from the remaining traffic is common in acknowledgment-based solutions and it is also typical for NoCs to have separate virtual networks for different types of traffic to eliminate message-dependent deadlocks.

Ejecting_packets vs. ack_required packets: At an initiating router, an ack_required packet traverses the VCA and SA steps only when the ejection and its desired output ports are both available. Such a packet can acquire a virtual channel in both ports, but then stall during SA, impeding other packets from being ejected. Thus, we add another ejection virtual channel to be used only for ejecting packet copies to retransmission buffers. The separation of ejection resources ensures that all packets can eventually be drained at their destination.

Traversing_packets vs. ack_required packets: Packets stalled waiting for a free retransmission buffer can block other in-flight packets, including those packets whose copies are occupying the retransmission buffers. If those packets cannot reach their destination, ACK packets cannot be sent back to release the retransmission buffers and resolve the blockage. To prevent the occurrence of a deadlock, we allow a fixed number of attempts in acquiring a retransmission buffer. If the packet fails all the attempts, we override the decision to copy the packet and we allow it to proceed to the downstream router. In our experimental setup, we set the maximum stalling duration to 256 cycles and found that this countermeasure affects very few packets. In all 15 workloads, REPAIR successfully protects >99% of packets traversing congested regions.

IV. Experimental Evaluation

We modeled an 8x8 mesh interconnect using the cycle accurate Booksim simulator [3] and implemented REPAIR. We also generated directed random traffic workloads to model communication patterns of applications running on CMPs and SoCs where, depending on the type and scheduling of applications, some nodes communicate more than others. Moreover, the rates and the frequency of communication, and the nodes involved might change throughout execution, creating different execution phases. In this section, we show results for 15 distinct traffic workloads, each consisting of three execution phases. The first and third phases are periods of low network activity and the second phase models a more active communication period, where a certain number of randomly chosen node-pairs communicate more frequently than others. The length of the high activity phase is set to 400,000 cycles and that of the low activity phases to 100,000 cycles each. In the workloads labeled *mc*, we set 6 high-communication node-pairs, chosen at random, to send traffic at an injection rate of 0.5 flits/cycle/node, while other nodes inject uniform traffic at a much lower rate of 0.1 flits/cycle/node. The set of chosen node-pairs remains the same throughout each workload. As such, the generated *mc* workloads model traffic patterns with small to medium sized congested regions, and we observe regions spanning 7 to 20 routers, on average. In the remaining workloads, labeled *hc*, we assume 10 randomly chosen high-communication node-pairs that inject traffic at a rate of 0.3 flits/cycle/node, while other nodes inject uniform traffic at a rate of 0.2 flits/node/cycle. We utilize different random seeds to generate 5 variations of this type of workload. The *hc* workloads create larger congested regions, consisting of 16 up to 35 routers, on average. In the low activity phases, the injection rate of each node is 0.05 flits/cycle/node.

A. Error Detection and Recovery

Our solution targets functional bugs in the interconnect design or implementation, and it does not address functional errors in the overlaying communication protocols, such as the cache coherence protocol. Moreover, REPAIR is independent of the exact source of the bug and its location within the network logic. Instead, REPAIR aims to detect and recover whenever these bugs affect the correctness of packet transfer. To evaluate REPAIR's error detection and recovery, we modeled 5 bugs, each triggered when routers encounter a different set of conditions, due to several contending packets traversing it (Table I). As soon as the conditions are met, the corresponding router drops one of its packets. Our injected bugs serve to model any incorrect packet delivery, and not a particular bug in the interconnect design. Note that we are not restricted to packet-drop type errors, indeed our model can also emulate the effects of incorrect packet delivery and incorrect destination delivery, which spans a wide range of errors such as livelocks, deadlocks, packet corruptions, *etc.* We ran our 15 workloads and observed the total number of times each bug manifested across all workloads and the number of workloads that were affected by the bug. Being representatives of design bugs that occur at runtime, our bugs are only triggered in corner-case executions and do not manifest very frequently or in all workloads, as shown in Table II. In all cases, REPAIR detects and recovers from the bug, as these complex corner-case conditions arise, as expected, in high-traffic regions, where packets are protected by REPAIR's acknowledgment-retransmission. Thus, a copy of each erroneous packets is retransmitted and delivered correctly to its destination.

B. Network Performance

We also evaluated the network performance when using REPAIR. Since source-based retransmission is traditionally adopted to ensure correct communication in the presence of serveral types of errors, such as dropped packets, packet corruption and misrouting, we also compared REPAIR against source-based retransmission. Figure 4 shows the execution time of our workloads when using REPAIR and source-based retransmission, normalized to the execution time of a baseline system that is not equipped with any retransmission capabilities. In both source-based and REPAIR, we assume two retransmission buffers per node. A network equipped with REPAIR can achieve communication correctness with 58% faster execution times than source-based retransmission. Although REPAIR is operating under the same network load as the source-based retransmission solution, REPAIR subjects only 25% of packets, on average, to acknowledgment retransmission. Thus, in REPAIR, there is less contention for the retransmission buffers, which accounts for the improved performance. Whereas, in source-based retransmission, there are many packets stalled at injection waiting for retransmission buffers, which causes, on average, a 75% slowdown. We further evaluated the performance trade-offs by varying the number of retransmission buffers utilized in source-based retransmission, and compared it against the execution time of a REPAIR implementation that utilizes only 2 retransmission buffers per node (Figure 5). The results are normalized to the execution time of a source-based retransmission solution using 8 buffers. As shown in this figure, REPAIR achieves better performance with 2-3x fewer buffer resources than traditionl source-based solutions.

Bug	**Description**
Bug_A	$AB = 6$, $AI = 3$, $\geq$ 10 flits in E,W,L ports $sw\_req = W - S$, $vc\_req = E.0 - N.0, E.1 - N.1, W.0 - E.0$
Bug_B	$AB = 7$, $AI = 4$, $\geq$ 16 flits in E,S ports, $\geq$ 8 flits in W,L prorts $sw\_req = W - S$,$vc\_req = E.0 - N.0, E.1 - N.1, W.0 - E.0$
Bug_C	$AB = 7$, $AI = 4$, $\geq$ 10 flits in E,L ports $sw\_req = L - S, E - W$, $vc\_req = E.0 - W.0, N.0 - L.0$
Bug_D	$AB = 6$, $\geq$ 10 flits in L ports $sw\_req = N - S, W - E, E - L$, $vc\_req = S.1 - L.0$
Bug_E	$AB = 7$, $\geq$ 16 flits in E, S ports, $\geq$ 10 flits in L port. $sw\_req = E - W, W - E, N - L$ $vc\_req = S.0 - N.0, S.1 - N.1, E.1 - W.0$

AB: # active buffers. AI: # active input ports.
N,S,E,W,L: North, South, East, West and Local ports.
sw_reqs: switch allocator requests. vc_reqs: virtual channel allocator requests.
e.g. W-S: packet in the West port is requesting the South port.
E.0-N.0: packet in VC=0 of the East port is requesting VC=0 of the North port.

TABLE I. **DESCRIPTION OF MODELED BUGS.**

Bug	**# times manifested**	**# affected workloads**	**# times recovered**
Bug_A	6	3 out of 15	6
Bug_B	2	1 out of 15	2
Bug_C	7	5 out of 15	7
Bug_D	3	3 out of 15	3
Bug_E	3	2 out of 15	3

TABLE II. **REPAIR'S BUG RECOVERY.**

C. Limitations and Discussion

1) Packet latency: Despite providing better network throughput, packet latencies may increase when using REPAIR. In the absence of free retransmission buffers at initiating routers, packets needing to be copied will stall, contributing to the increased average packet latency. The average latency of packet traversal through the network when using REPAIR, with 2 retransmission buffers, is up to twice that of a throughput-equivalent source-based retransmission solution. In contrast, in source-based retransmission, the lack of free buffers causes packets to stall at injection, leading to a reduction in throughput, but once packets are in-flight, no additional stalling occurs.

2) Implementation overhead: Assuming routers with 5 ports, 2 virtual channels per port and 8-flit buffers, REPAIR's congestion detection requires three 7-bit registers per router to hold each of the occupancy counter, the *upgrade_threshold* and the *downgrade_threshold*. We set the *downgrade_delay* to 10 bits, allowing a maximum delay of 1,024 cycles before toggling the local congestion flag. The congestion status table consists of 1 entry to store the router's local congestion and 4 entries to store the congestion statuses of its neighbors. The congestion status itself is encoded with 1 bit. To implement REPAIR's protection scheme, a *failed_copy_attempts* counter is needed per input buffer to monitor the number of times a packet fails to acquire a retransmission buffer at an initiating router. We set this counter to 8 bits, capping the waiting-time to 256 cycles. Thus, the total storage overhead of REPAIR is 116 bits per router, less than a one-flit entry of a router's input buffer, which is typically at least 128 bits. As for the control logic to update these counters, its area overhead is minimal (<0.1%). Lastly, to instrument routers to create copies of in-flight packets, we require an additional output virtual channel in the ejection port and modifications to the virtual

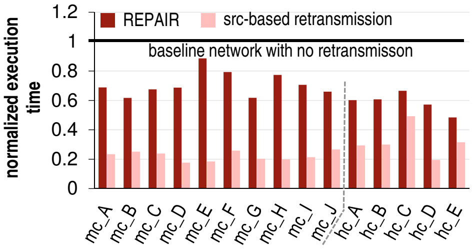

Fig. 4. **Execution time of each workload, comparing REPAIR to a traditional source-based retransmission.** Results are normalized to the execution time of a baseline NoC, without any retransmission capabilities. REPAIR's performance is 58% better, on average, than source-based retransmission.

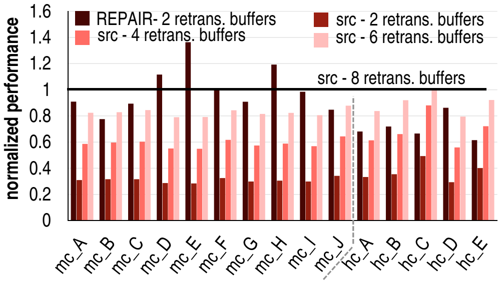

Fig. 5. **Buffering requirements of REPAIR vs. source-based retransmission.** Execution time is normalized to source-based retransmission utilizing 8 retransmission buffers per node. On average, REPAIR achieves better performance, even with 2-3x fewer retransmission buffers.

channel allocator to account for it. This additional virtual channel does not require adding another port to the router and has minimal effects on the router logic. We modeled and synthesized a baseline router using Synopsys Design Compiler with a 45nm target library and found that the area overhead for implementing this functionality is 0.86%. The total area overhead of REPAIR is $<$1% (0.86% + 116 bits).

V. Related Work

Ensuring the runtime correctness of communication in NoCs is a widely explored topic. However, many solutions focus on correctness in the face of transient or permanent faults in the NoC hardware. [4] is a recent survey of common fault-tolerance techniques. Source-based acknowledgment-retransmission mechanisms are among those that are most relevant to our work [5]. However, we show that relying on source-based retransmission for detecting and recovering from escaped design bugs is prohibitively costly and ineffective to resolve design bugs. We propose a different acknowledgment-retransmission approach that successfully protects only the packets that are susceptible to design bugs.

Several works have proposed runtime approaches that target functional design errors in processors and memory systems [6]–[9], whereas a few focus on NoCs, [10], [11]. The authors of [10] use a combination of formal verification and runtime checkers to ensure correct NoC operations. [11] is solely based on a runtime solution, but fails to protect the network from some of types of errors. Both approaches rely on augmenting the baseline network with a checker network, introducing significant area overheads ($>$9%). In contrast, REPAIR uses an extremely lightweight technique that can successfully protect network communication, independently of the types of errors encountered, while incurring ($<$1%) area overhead.

Many previous works, [2], [12]–[15], have explored congestion detection in NoCs and they rely on a variety of metrics including crossbar demand, free virtual channels, free buffer space, output port contention, *etc.* Local congestion is then propagated through the network, by aggregation, [12], or through a separate subnetwork [13], [14]. In these solutions, congestion detection is used for providing better performance, routing algorithms, or congestion control mechanisms [15], [16], often requiring an accurate global view of congestion. However, in REPAIR, we aim to detect congested regions for the purpose of identifying areas with complex activity, thus we resort to a simpler, yet effective, estimate of congestion.

VI. Conclusions

We introduced REPAIR, a runtime solution that protects on-chip networks from the manifestation of design bugs. Complex execution scenarios are often encountered in high-traffic network regions, making these regions susceptible to the manifestation of latent design bugs. REPAIR identifies congested regions and protects packets traversing them with an acknowledgment-retransmission scheme. Unlike common source-based retransmission techniques, acknowledgment-retransmission is selectively utilized only for a small subset of packets, while allowing REPAIR to successfully detect and recover from errors that manifest in congested regions.

Acknowledgments. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, and NSF grant #1217764.

References

[1] S. Chatterjee, M. Kishinevsky, and U. Ogras, "xMAS: Quick formal modeling of communication fabrics to enable verification," *Design Test of Computers, IEEE*, vol. 29, no. 3, 2012.

[2] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *Proc. ISCA*, 2013.

[3] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[4] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch, "Methods for fault tolerance in networks-on-chip," *ACM Computer Survey*, 2013.

[5] S. Murali, T. Theocharides, L. Benini, G. D. Micheli, N. Vijaykrishnan, and M. J. Irwin, "Analysis of error recovery schemes for networks on chips," *IEEE Design & Test*, 2005.

[6] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *Proc. MICRO*, 1999.

[7] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proc. MICRO*, 2007.

[8] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in *Proc. DATE*, 2007.

[9] A. Meixner and D. Sorin, "Error detection via online checking of cache coherence with token coherence signatures," in *Proc. HPCA*, 2007.

[10] R. Parikh and V. Bertacco, "Formally enhanced runtime verification to ensure NoC functional correctness," in *Proc. MICRO*, 2011.

[11] R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco, "Functional correctness for CMP interconnects," in *Proc. ICCD*, 2011.

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

[13] X. Chang, M. Ebrahimi, M. Daneshtalab, T. Westerlund, and J. Plosila, "PARS; an efficient congestion-aware routing method for networks-on-chip," in *Proc. CADS*, 2012.

[14] J. Escamilla, J. Flich, and P. Garcia, "ICARO: Congestion isolation in networks-on-chip," in *Proc. NOCS*, 2014.

[15] E. Baydal, P. Lopez, and J. Duato, "A family of mechanisms for congestion control in wormhole networks," *IEEE Trans. Parallel and Distributed Systems*, 2005.

[16] M. Thottethodi, A. Lebeck, and S. Mukherjee, "Self-tuned congestion control for multiprocessor networks," in *Proc. HPCA*, 2001.