DiAMOND:Distributed Alteration of Messages for On-Chip Network Debug

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

Abstract—**During emulation and post-silicon validation of networks-on-chip (NoCs), lack of observability of internal operations hinders the detection and debugging of functional bugs. Verifying the correctness of the control-flow portion of the NoC requires tests that exercise its functionality, while abstracting the data content of traffic. We propose a methodology where network packets are repurposed for the storage of debug information collected during execution. Debug data pertaining to each packet is collected at routers along its path and stored by replacing the packet's original data content. Our solution is coupled with a detection scheme consisting of small checkers that monitor execution and flag bugs. Upon bug detection, we analyze the debug information to reconstruct network traffic. We also provide relevant statistics for debugging, such as packet interactions and packet latencies, per router. In our experiments, this approach allows us to reconstruct over 80% of the packets' routes. Moreover, the obtained statistics facilitate debugging erroneous network behavior and identifying performance bottlenecks.**

I. Introduction

Networks-on-chip (NoCs) have become the prevalent communication paradigm for current and future chip-multiprocessor (CMP) and system-on-chip (SoC) architectures. In today's market, large chip-multiprocessors, incorporating hundreds of processing elements, are developed to provide the computational power needed to run parallel and high performance computing applications. Concurrently, SoC design is experiencing an increasing trend of high integration, where a number of IP blocks, commonly obtained from third party vendors, are integrated on to a single chip through a communication substrate. The network-on-chip model constitutes a distributed and generalized architecture that can meet the growing communication needs of SoCs and CMPs.

A NoC consists of a set of routers connected according to a chosen topology. In the mainstream approach of wormhole routing, messages sent over the NoC are divided into packets, which in turn are divided into equal size segments of data called 'flits'. A generic router design includes several storage buffers and arbitration and allocation units to assign resources to packets-in-flight. Router designs are often complex, as router architectures may include a number of advanced features, such as virtual channels and intricate arbitration units. Moreover, a number of regular and irregular topologies can render the packet flow and the overall network subsystem extremely complex. In addition, network complexity is further increased when deploying elaborate routing protocols that utilize network state to guide routing decisions.

For these large CMP and SoC architectures, a great deal of effort is spent in the functional verification process of the individual cores and IP blocks, the NoC interconnect itself, as well as the entire system. With the increase in the size and complexity of these systems, along with shrinking time-to-market windows, a lot of this effort is shifting towards the heavy use of emulation and post-silicon validation to ensure functional correctness. A key underlying reason for this trend is the vast complexity of modern CMP and SoC designs and their communication subsystem, which are often too large for formal and software-based validation solutions. In emulation, the design under test is mapped onto configurable hardware units, such as FPGA-based platforms. Tests are run on the emulation platform, which provides orders of magnitude speedups relative to software-based simulations of the design's RTL description. On the other hand, post-silicon validation comes at a later stage, when the first few silicon prototypes of the chip become available. During this phase, tests run directly on the hardware and at-chip speed, enabling a faster and hence more thorough validation of the system's functionality. In the context of validating the NoC design, emulation and post-silicon validation provide great advantages in speed but suffer from limited observability of the design under test. Lack of visibility of internal operations makes the detection, diagnosis and debug of errors an extremely challenging process. Moreover, general solutions that enhance observability, such as scan chains, do not always provide sufficient debug data to permit a fast and efficient functional validation of the NoC.

In our work, we aim to address the challenge of validating the complete network subsystem on these fast platforms (emulation and post-silicon). In performing the functional validation of NoCs, we can consider any NoC design to consist of two components: data flow and control flow. Validating the data flow correctness means ensuring that data sent over the network is not corrupted in transfer. Verifying the correctness of the control flow portion essentially means validating all functionality, and hence control decisions made in the network. The NoC functionality is entirely dependent on the traffic patterns observed and it is agnostic to the data content of the messages. Therefore, specialized test cases can be run with the goal of exercising as much of the network's functionality as possible. When running such tests, the packets' data contents are effectively irrelevant. In this context, we propose a methodology to greatly enhance the observability of the network traffic and its internal state to facilitate the detection and debug of control-flow functional errors. Our solution, called DiAMOND, proposes to replace packets' data contents with debug information collected during the network's execution. At every router along a packet's path, we gather debug data that encapsulate the packet's current state and we systematically substitute the data flits of the packet with this information. Once packets arrive to their destinations, they are stored at the local cache or memory associated with those nodes. Along with this data collection mechanism, we instrument routers with small checkers that can detect various functional errors. Upon error detection, the collected debug data accumulating throughout the network is analyzed by software-based algorithms running on the CMP/SoC cores or off-chip. The information that can be extracted from this data includes a detailed overview of the packets' paths, analysis of performance metrics at internal routers, as well as the sequence of events observed at a given router during a given interval. Armed with this enhanced visibility into the network behavior, verification engineers can more promptly localize and debug functional (and in some cases performance) bugs.

A. Contributions

• We introduce a novel solution to gather debug data during the emulation or post-silicon validation of NoC interconnects and, specifically, while running tests targeting the validation of the NoC's control-flow.
• We present a complete framework that couples our debug data collection mechanism with a bug detection scheme. We also present a debug data processing methodology to extract relevant information that facilitates the diagnosis and debug of functional errors in the NoC design.
• Our solution introduces minimal perturbations and requires minor hardware additions. We also provide three modes of operations to configure our data collection mechanism, trading-off the exhaustiveness of the debug data gathered with the degree of perturbation to the original system.
• In addition to detecting and debugging functional errors, our solution can also detect performance bugs, such as starvation and misroutes. It can also provide performance statistics at internal network routers, which in turn aids in analyzing the NoC's overall performance.

II. Related Work

Previous work on post-silicon validation of NoCs have proposed various approaches to increase NoC observability. In [1], authors instrument routers and network interfaces with monitors. These monitors filter network traffic to identify transactions of interest, as well as analyze performance and validate data flow errors. Similarly, approaches proposed by [2], [3] add monitors to network routers that observe traffic and abstract it into events or transactions. The extracted events and transactions are then transferred over the network for further analysis. Our solution differs from these approaches by focusing on the validation of functional bugs in the control-flow portion of NoC designs. It can also detect and debug some types of performance bugs. In contrast to the above works, we provide a complete framework that can collect debug data, detect functional bugs, and then analyze the data for diagnosis and debugging. The collected debug data is transferred by substituting the original data content of packets. Moreover, the type of debug data collected and how it is analyzed is independent of network topology and router architectures.

Other recent work proposes a post-silicon solution that relies on taking periodic snapshots of traffic to reconstruct packet paths and identify functional errors related to forward progress [4]. By periodically sampling traffic, this solution can only provide low error detection probability for those bugs that are transient in nature, such as misroutes or starvations. It also fails to detect other types of bugs, such as dropped packets. On the other hand, our approach achieves a much better observability of network traffic, as all data packets can be observed, for most or all of their paths. We can also reconstruct longer and more uniform routes for each packets, achieving a reconstruction rate of more than 83%. In addition, since debug data is collected per packet and then stored within each packet's body flits, the amount of debug data logged is proportional to the amount of traffic and not the test execution length.

A number of works have targeted emulation of networks-on-chip [5]–[9], where authors proposed different ways of implementing an emulation platform that allows the modeling and exploration of various NoC designs. The emulated NoC is evaluated by relying on traffic generators and receptors that can be configured to inject different traffic patterns and analyze received packets. These works focus mostly on designing an emulation platform and a methodology that allows modeling different NoCs to speed up design exploration and validation. Our work is complementary to these approaches. Independently of how the NoC is emulated, we provide a methodology to collect debug data from network routers to facilitate the debugging of functional errors in the control flow portion of the design. Moreover, the traffic receptors proposed are limited to analyzing end-to-end correctness and performance metrics and can not provide insights regarding the internal events that occurred in the network. In contrast, our debug data collection mechanism provides a detailed view of internal network behavior, generating results about packet interactions within routers, packet latencies observed per router, as well as the routes followed by the packets.

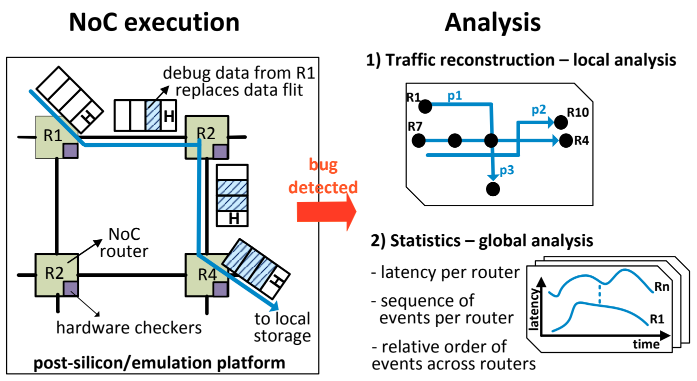
NoC execution
Analysis
debug data from R1 replaces data flit
H
R1
R2
R2
R4
NoC router
hardware checkers
to local storage
post-silicon/emulation platform
bug detected
1) Traffic reconstruction – local analysis
R1
p1
R7
p2
R10
R4
p3
2) Statistics – global analysis
- latency per router
- sequence of events per router
- relative order of events across routers
latency
time
Rn
R1

Fig. 1. **Overview of our solution.** During NoC execution, debug data is collected at every hop and stored in the packet, overwriting data flits. Routers are instrumented with hardware checkers that monitor execution and flag functional errors. Upon error detection, the debug data is analyzed to reconstruct traffic, as well as provide a number of relevant statistics.

III. The DiAMOND Solution

A. Overview

The aim of DiAMOND is to provide observability of the network's operation to facilitate the diagnosis and debug of functional errors in the NoC's control flow. Tests used to validate the NoC functionality aim at creating various network traffic scenarios, while abstracting away the data content of messages. When running such tests, our solution relies on using the contents of packets to store debug data collected during execution. Network execution is partitioned into epochs, during which the network is instrumented for bug detection, as well as debug data collection. As packets traverse the network, their data content is substituted with debug information collected at every hop, as illustrated in Figure 1. Once a packet is delivered to its destination node, it is stored in the local cache or memory associated with that node. In parallel with debug data collection, small hardware checkers monitor the network's execution and detect functional bugs. Upon flagging an error, execution is halted and the debug data that has been collected in the caches is analyzed. This analysis process is first carried out locally, where each processor core examines the debug data of packets that were destined to itself, and then globally, where debug data from all nodes are aggregated at a central location for a global overview of the network's behavior. On the other hand, if the epoch ends without the detection of any bugs, the collected debug information in the caches is simply overwritten in the following epoch.

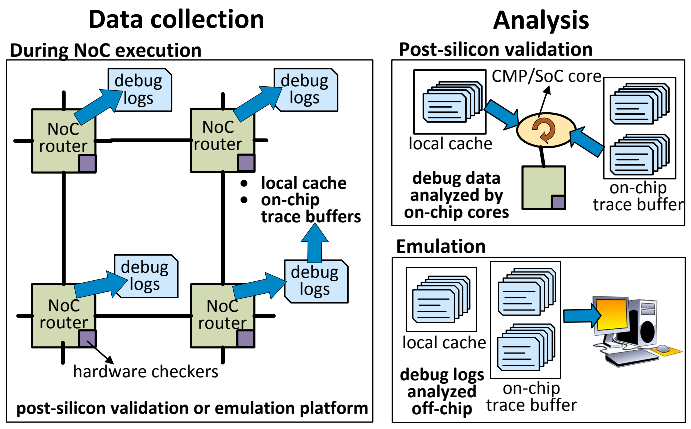

Fig. 2. **Functional validation flow.** During the emulation or post-silicon validation of NoC designs, debug data is collected during testbench execution. The debug logs can be stored in the local cache or memory associated with each node or in on-chip trace buffers. Upon the detection of an error, execution is halted and the collected debug data is analyzed by the on-chip processor cores or can be transferred off-chip and analyzed separately.

Since this approach relies on utilizing each packet's flits to carry debug information, the amount of debug data that can be collected is limited by the packet's size and the length of its path through network. However, in practice, this limitation does not always cause a lack of scalability. With low packet latency being a primary concern, NoC designs, even if large in topology, commonly incur a low average hop count for packets in flight. Moreover, in this paper, we consider a baseline CMP system, where each node consists of a processing element and a local cache, but our solution is also adaptable to SoC designs, where the system consists of general purpose processors, as well as other IP modules. In this context, only nodes with memory/cache modules store debug data. It is also common during post-silicon validation and emulation to have additional on-chip trace buffers, which can be utilized to store debug data for nodes without caches. The debug data analysis process can be carried out by running a software program on the on-chip cores or by performing the analysis off-chip. Figure 2 shows the complete validation flow of our proposed solution.

B. Debug Data Collection

Debug data is collected for every packet injected into the network and at every hop during its flight. At each hop, the debug data associated with each packet is stored in one of the packet's flits replacing the original data content.

For every input buffer within the router, we add a register, called *log_buffer*, to store the collected debug information. In addition, we require each router to include a packet counter (*pckt_cntr*) that is incremented upon receiving a packet. The *log_buffer* is updated everytime a new packet is at the head of its corresponding input buffer. The information collected and stored in the *log_buffer* consists of:

1) The router ID
2) Arrival timestamp (*timestampA*) that indicates the value of the *pckt_cntr* when the header flit of the packet was received by the router.
3) Departure timestamp (*timestampD*) that indicates the value of *pckt_cntr* when the header flit was sent from the router. Logging timestampA and timestampD allows us to order packets passing through each router, as well as reason about packet interactions within a router.
4) A third timestamp (*pckt_latency*) that indicates the amount of time (in cycles) the packet's header flit remained in the router. This timestamp allow us to analyze packet latencies observed at interval routers.
5) The packet's input port and input virtual channel.
6) The output port and virtual channel the packet requests.

Once the *log_buffer* is complete, the debug data is written to one of the packet's body flits. The index of the flit to be written is maintained by a counter that is added to the packet's header flit. When a packet arrives to a router and reaches the head of one of its input buffers, we first extract the flit index where the debug data will be written. Then, *timestampA*, the input port and the input virtual channel are logged in the *log_buffer*. When the header flit completes its route computation and virtual channel allocation, the requested output port and requested virtual channel are logged. Finally, when the header flit is sent to the next router (or ejected if it is at a destination router), *timestampD* and *pckt_latency* are logged. Once the packet's header has been routed to the next hop, the packet's body flits follow. Based on the flit's write index, the *log_buffer* is simply written in the appropriate flit.

A typical flit width in NoCs is between 128 and 256 bits [10]. In our evaluation, we assume a flit width of 128 bits and a *log_buffer* size of 64 bits. Therefore, the debug data collected at every hop occupies only half a flit, with the remaining half written at the next hop. In order to implement this functionality, the flit write-index field in packet headers is extended by 1 bit, which indicates whether the debug data will replace the first or second half of a body flit. Moreover, the flit write-index field is incremented once every two hops. Figure 4 shows the various fields of packets' header flits.

In the case of packets that do not have enough flits to store the collected debug information, we provide three solutions for our approach, depending on the needs of the verification methodology in use: *drop remaining*, *drop at alternate hops*, and *append*. Figure 3 illustrates the behavior of each mode. In these three modes, the verification process can be tuned to trade-off debug capabilities with the degree of perturbation introduced to the original system.

1) Drop Remaining: During drop remaining, when the number of hops in a packet's path exceeds the available flits

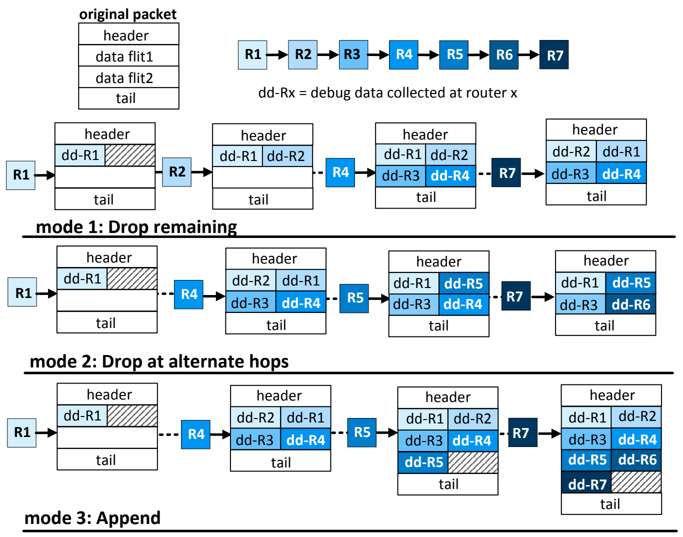

Fig. 3. **Debug data collection.** In *drop remaining*, if the number of hops exceeds the available data flits, additional debug data is simply dropped. During *drop at alternate hops*, additional debug data replaces alternating entries providing a more uniform overview of the packet's path. The last mode, *append*, creates new flits as needed and appends them to the packet.

in the packet, additional debug data is dropped. This mode of operation is simple to implement at the expense of low observability for packets with long routing paths.

2) Drop at Alternate Hops: In this mode of operation, debug data collection is implemented as before. However, when the space in the packet is exhausted, new debug data overwrites older debug data, creating an every-other-hop scheme. For example, as shown in Figure 3, the debug data collected at router R5 replaces older debug data collected at router R2. As opposed to drop remaining, this mode provides a longer and more uniform overview even of long routing paths. Moreover, data belonging to the alternating missing hops can be partially reconstructed or extrapolated from the debug data that remains. For example, the output port requested along with the router ID can be used to determine the downstream router, for which we have no log. Similarly, the input port and router ID can be used to determine the missing upstream router.

3) Append: We also provide a mode of operation that allows routers to append new flits to the packet. New flits are appended before the tail flit, as illustrated in Figure 3. While this mode provides the complete path of a packet, it requires additional hardware to add the new flits. It also alters the network's original execution by creating longer packets, potentially masking bugs. It is also possible for the perturbation created by increasing the length of some packets, to expose bugs that would have not been observed otherwise.

C. Error Detection

Our debug data collection methodology is orthogonal to the mechanism by which functional bugs are detected in the NoC. We propose the use of a fine-grain detection approach that relies on adding small checkers to the NoC routers. These checkers monitor runtime execution for signs of erroneous behavior, while targeting a wide range of functional bug manifestations detailed below. Instead of localizing the root cause of a functional bug, our detection mechanism targets the bug's manifestation on the traffic in-flight. Whereas, our debug collection mechanism stores low level debug data pertaining to each stage of a router's control path (route computation, virtual channel allocation and switch allocation), which later permits a more detailed diagnosis and debugging analysis. In terms of detection, irrespective of its exact location, a functional bug in a NoC can manifest by affecting the traffic in-flight in a finite number of ways. First, a functional bug can lead to bit corruptions in the transferred data, which can be detected by including an error correction code (ECC) in each flit. However, in this work, we target the validation of the control flow portion of the NoC design and therefore only focus on that subset of functional bug manifestations. In terms of control flow and at the level of packets, a functional bug can hinder the forward progress of packets through the network, such as in the cases of deadlocks, livelocks, starvations, and misroutes. It could also lead to entire packets being dropped or duplicated. At the level of flits, a functional bug in the control flow can manifest as dropped or spurious flits.

Deadlock and starvation: Packets involved in a deadlock are permanently blocked waiting on each other to free needed resources. On the other hand, starvation occurs when a packet is temporary blocked waiting for resources that are allocated to other packets due to unfair arbitration and allocation schemes. Note that unless bounded packet delivery is a system requirement, starvation does not always affect the correctness of execution and it is often considered a performance bug. A common technique to detect blocked packets adds counters to routers, one associated with each input buffer. After a header flit, marking the beginning of a new packet, reaches the head of an input buffer, its corresponding counter is incremented in every cycle. The counter is reset when the tail flit is observed. If the counter exceeds a user-defined threshold, it flags an error [11]. In our work, we differentiate between deadlock and starvation by allowing the network to drain after the error is flagged. In contrast to deadlock, starved packets will eventually acquire the resources they need to move forward and their corresponding counters reset to zero.

Livelock: Packets are in livelock if they are continuously transferred between routers without making forward progress to their destinations. A common approach to detecting a livelock adds a hop counter to the header flit of every packet. The counter is incremented at every hop and a livelock is flagged if the counter exceeds a pre-defined threshold [12].

Dropped and duplicated packets: To detect dropped packets, we utilize the approach proposed by [13], where a packet counter is maintained per router. The counter is incremented upon receiving a tail flit and decremented upon sending one. If packets are not dropped within the router, then the counter should reach a value of zero at some point within a checking window. Similarly, a packet counter reaching a negative value can be used to identify packet duplication or spurious packet creation. In the case of dropped packets, it is possible for this approach to exhibit false positives, particularly under high congestion traffic. High congestion can also mask packet duplications, causing this technique to exhibit false negatives. However, [13] shows that choosing a suitable checking window size can practically eliminate false positives. Moreover, false negatives are rare and duplications will eventually be detected.

Dropped and duplicated flits: In order to identify dropped and duplicated flits within packets, we require the addition of a *size* field to the header flit. A simple counter and comparator added to every input buffer are then used to keep track of the number of flits observed. If the tail flit is reached and the counter does not match the size field, then a flit must have been duplicated, created or dropped.

Misrouting: A packet is misrouted if it is sent to the wrong destination. To detect such errors, we perform a check upon packet delivery to ensure that the destination field in the header flit matches. Misrouting could also occur if a packet is delivered to the correct destination, but takes incorrect routes along its path. In such cases, misroutes can be detected by adding simple checkers to internal routers. The exact checker implementation is dependent on the routing protocol. For example, for deterministic or minimal routing algorithms, a simple lookup table or assertion can detect such errors [13].

D. Debug Data Analysis

Once an error has been flagged, execution is halted and the network is allowed to drain. Packets blocked, due to deadlocks or livelocks, are permitted to drain to the closest node. At this point, all the debug data that was collected during execution is residing in the content of packets, which are stored in the local caches or trace buffers across the network. This data is processed in two steps: local and global, each providing a different overview of the network's execution.

1) Local Processing: The content of each local cache is individually analyzed by a software application running on the corresponding core. The data can also be loaded off-chip for a similar analysis. By examining the contents of every packet, its path through the network can be reconstructed. The path overview allows the identification of any livelock cycles, as well as any misrouted segments along its route. In the case of adaptive routing algorithms, the reconstructed paths provide insights regarding the performance and effectiveness of the routing protocol. In addition, by examining the recorded *pckt_latency* timestamps, network performance can be analyzed. Periods of high packet latency can be identified along with the routers where this high latency was recorded. Finally, by comparing a packet's requested output port and output virtual channel within a router relative to the input port and input virtual channel of the downstream router, functional bugs in switch arbitration logic can be flagged.

2) Global Processing: Through the local processing step, execution intervals or routers of interest are identified. Then, data from all local caches are aggregated at a central location, where another software algorithm, running on one of the cores or running off-chip, groups this data on a per router basis. Then, using the *timestampA* and *timestampD* counters, each router's data is sorted by increasing time. The sorted information basically encapsulates the series of packets and events witnessed by each router during execution. This, in turn, gives insights regarding packet interactions within routers, allowing us to reason about the source of the error observed. Since each router's *timestampA* and *timestampD* represent the value of the router's packet counter, these timestamps do not have a notion of physical time. Therefore, the arrival and departure of packets from different routers can not be correlated. However, by leveraging techniques similar to those used in ordering events for distributed systems [14], we can still construct a partial order of events by using packets as points of reference. A packet transferred from routerA to routerB serves as a synchronization point, where events observed in routerA before the packet was sent can be classified to have happened before the events occurring in routerB after the arrival of the packet.

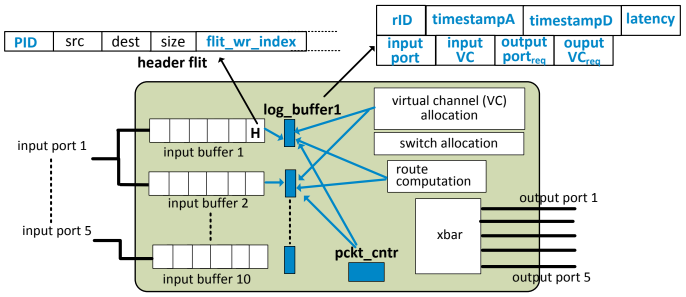

Fig. 4. **Debug data collection - hardware implementation.** Additional fields are added to the header flits of packets. A register, log_buffer, is added to each input buffer to store debug data. A packet counter is required per router to provide the *timestampA* and *timestampD* values.

IV. Implementation of Debug Data Collection

In order to implement DiAMOND's debug data collection solution, we include additional fields in the header flits of packets, as shown in Figure 4. A header flit commonly carries the router IDs of the packet's source and destination nodes. It also commonly has unused bits, which we can utilize for our solution. Therefore, we include a small counter, which along with the source and destination, serves as an ID that can uniquely identify the packet in the network. Moreover, to keep track of the flit ID where the debug data will be stored at each router, we require an additional *flit_write_index* field. The *flit_write_index* is a counter that is incremented every two hops. It also has an extra bit, which indicates whether the debug data will replace the first or second half of the flit, as explained in Section III-B. Note that the length of the *flit_write_index* is determined by the number of body flits in a packet and would typically be 3-4 bits. Table II shows the various fields added to header flits and their length.

We also require some minor additions to the NoC routers. Debug data collected at each router is stored in a register, the *log_buffer*, before its written to the appropriate data flit. We require one *log_buffer* for every input buffer. The size of the *log_buffer* register depends on the network size and router architecture. Table I lists the various entries of a *log_buffer* and their lengths. In addition, we add a packet counter per router, which tracks the number of packets received by the router and which provides the *timestampA* and *timestampD* values. In order to record the *pckt_latency*, we make use of the same deadlock/starvation counter needed for bug detection (Section III-C). As for the remaining entries in the *log_buffer*, they can be recorded directly from the original router implementation and do not require any additions. Once a packet is received by a router, the *flit_write_index* field of the header flit is extracted. Then, a simple combinational logic counts the number of data flits observed and when it matches the *flit_write_index*, it copies the *log_buffer* to the appropriate half of the data flit. We implemented these additions and the detection checkers in the Verilog model of the baseline router architecture that is described in Section V. Synthesis results show an area overhead of 2%. Moreover, the incurred power overhead is minor and is in itself not a significant concern during the emulation and post-silicon validation of the NoC. These hardware modifications are also decoupled from the router's functionality and can be disabled when the chip is released.

V. Experimental Evaluation

To evaluate our solution, we modeled an 8x8 mesh network using the cycle-accurate Booksim simulator [12]. Our baseline router architecture consisted of a general input-queued virtual channel router, with 5 input ports and 2 virtual channels per port. We ran both random directed traffic, as well as network flow traces from the PARSEC benchmark suite [15]. For uniform random traffic we varied the packet size between 5 flits/packet (a header, a tail and 3 body flits) and 7 flits/packet. As for the PARSEC network flow, traffic consisted of both control packets and data packets. While data packets consisted of 5 flits, control packets were only 1-flit long.

log_buffer entries	**number of bits**
routerID	6 bits
timestampA	15 bits
timesatampD	15 bits
pckt_latency	10 bits
input port	3 bits
input virtual channel	1 bit
output port requested	3 bits
output virtual channel requested	1 bit
total	**64 bits**

TABLE I. Log_buffer.

fields	**number of bits**
PID	8 bits
flit_write_index	3-4 bits
size	4 bits
total additions	**14-15 bits**

TABLE II. Additions to header flits.

We modified Booksim to implement the three modes of the data collection mechanism: drop remaining, drop at alternate hops and append. Based on the network size and router architecture, we determined the length of the *log_buffers* required at every router, as shown in Table I. The lengths of *timestampA* and *timestampD* (and hence the *pckt_cntr*) were chosen to be 15 bits, to ensure that the packet counter does not wrap around too frequently. In the event that a wrap-around occurs at any router, we force the epoch to end early, which permits clearing the previously collected debug data from the caches and restarting the counters. We chose a length of 10 bits for the *pckt_latency* field, limiting the maximum latency value that can be logged to 1,024 cycles. Finally, we assume a flit size of 128 bits, which is a common flit length [10]. Therefore, we are able to store the *log_buffers* collected along two hops in each flit, as explained in Section III-B.

Our implementation also requires adding several fields to each packet's header flit, which are listed in Table II. We chose the packet ID to be an 8 bit counter, which along with the packet's source and destination node IDs forms a unique identifier of each packet. The length of *size*, which is used for the detection of dropped and duplicated flits (Section III-C), depends on the number of flits in a packet. In our evaluation, we consider packets of size 5 and 7 flits, making the *size* field 4 bits long. Similarly, *flit_wr_index* depends on the number of body flits in a packet, with an additional bit to indicate which half of the flit is to be written.

A. Path Reconstruction Results

We first examined the observability gained from utilizing our debug data collection solution by evaluating the fraction of the path that can be observed for each packet. In our validation platform, the path reconstruction process is completed during the local processing phase, where body flits of packets are examined and the sequence of routers, through which each packet passed, is reconstructed. Table III shows the average percentage of each path that can be reconstructed under all three modes of operation. For the PARSEC network flow, data packets consist of 3 body flits and could carry complete debug data from 6 routers along their path. Therefore, under drop remaining, we are able to achieve full observability (100% path reconstruction) over packets whose path traverses 6 or fewer routers. Remaining packets have smaller path reconstruction fractions depending on their path length. For all PARSEC benchmarks, the percentage of path reconstruction is 83.76% on average. During the drop at alternate hops mode, when all body flits have been utilized, new debug data replaces older data by over-writing only the second half of each body flit, as illustrated in Figure 3. Moreover, routers pertaining to the alternate missing hops can be extrapolated. For the PARSEC network flow, in addition to the 6 routers that can be extracted directly from the debug data, the 3 alternate routers that were overwritten can be deduced from the recorded router IDs and input ports. Therefore, paths consisting of up to 9 routers can be fully observed. This mode provides a higher path reconstruction of 96.54%, on average. Note that, the PARSEC network flow also consists of 1-flit control packets that do not have any body flits. For such packets, we are not able to collect any debug data during these two modes. Finally, for the append mode, we achieve 100% path reconstruction for both control and data packet, as expected, since packets can append as many new flits as needed to store debug information. Similar results are also observed for uniform random traffic. Results are averaged over a sweeping injection rate from low injection, 0.04 flits/cycle/node, to high injection, 0.24 flits/cycle/node.

PARSEC network flow	**drop remaining**	**drop at alternate hops**	**append**
blackscholes	83.2%	96.3%	100%
bodytrack	85.0%	97.1%	100%
dedup	84.4%	96.8%	100%
ferret	84.5%	96.9%	100%
freqmine	83.8%	96.6%	100%
streamcluster	84.3%	96.8%	100%
swaptions	84.2%	96.8%	100%
vips	81.4%	95.4%	100%
x264	83.0%	96.2%	100%
average	**83.76%**	**96.54%**	**100%**
uniform traffic packet size = 5 flits	87.1%	97.8%	100%
uniform traffic packet size = 7 flits	98%	100%	100%

TABLE III. Average path reconstruction

B. Performance Analysis

Under drop remaining and drop at alternate hops, our solution does not introduce any additional performance overhead. However, the append mode can increase the length of packets to create space for storing new debug data. Longer packets increase network congestion and can slow down execution. Figure 5 shows the average network latency for the PARSEC network flow during the 3 modes of operation of DiAMOND, as compared to a baseline system without our solution. As expected, the network does not incur a performance impact when operating during drop remaining and drop at alternate hops. However, when operating with the append mode, average

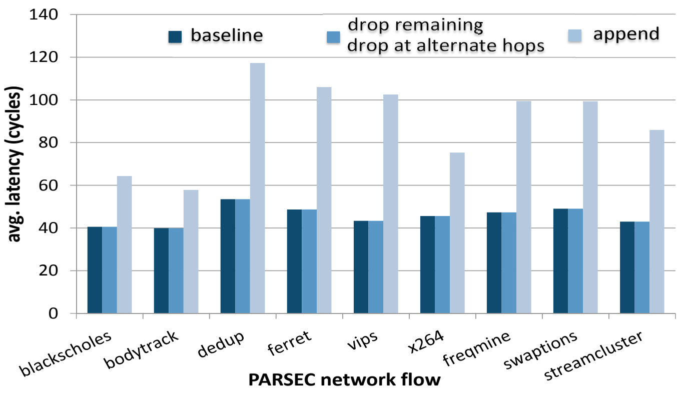

Fig. 5. **Average network latency for PARSEC network flow traffic.** The append mode of operation provides complete observability of all packets at the expense of increased latency.

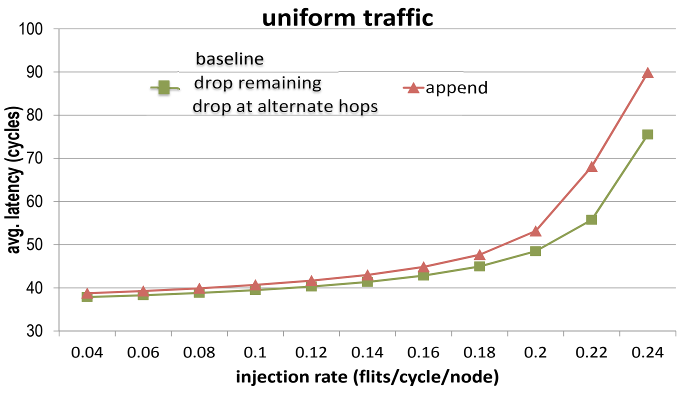

Fig. 6. **Average network latency for uniform traffic.** The append mode of operation increases the network's average latency.

network latency increases by a factor of 1.95, on average. A similar trend can be observed for uniform traffic with packets consisting of 5 flits, as shown in Figure 6. At low to medium injection rates, the impact of the append mode on network latency is minimal. As injection rate increases, the increase in average network latency is more pronounced. In those cases, the network is already congested and the append mode increases the length of some packets resulting in even more congestion. On average, the append mode increases the average network latency by a factor of 1.07. Note that the PARSEC network flow exhibits a greater performance impact than uniform traffic. This is due to the fact that PARSEC benchmarks consist of both data and control packets. With control packets being only 1-flit long, all of them require appending additional body flits to carry the collected debug data. Thus, more packets are affected by extensions than when running uniform traffic, resulting in a greater increase in average network latency.

C. Case Study: Analysis Results

Packet interactions. Our solution also allows examining packet interactions within routers, as well as constructing a partial overview of global network behavior. During the global analysis phase, debug data from all nodes is aggregated and grouped per router. Each router's data is then sorted by increasing *timestampA* values, allowing us to order packet arrival and departure to and from each router. As an example, we ran uniform traffic over an 8x8 mesh at an injection rate of 0.19 flits/node/cycle. Figure 7 shows the packets traversing

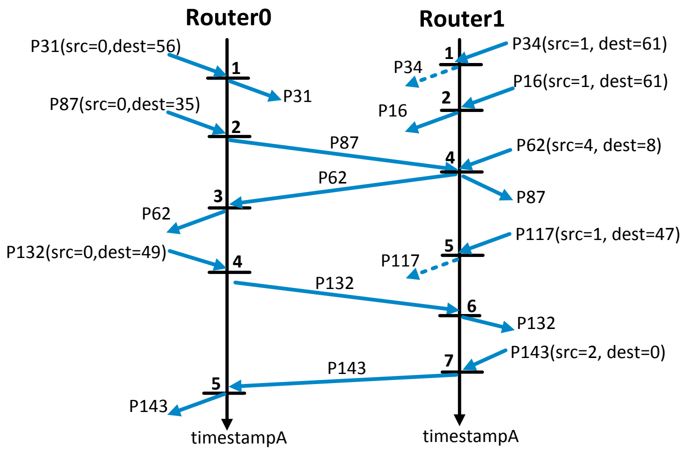

Fig. 7. **Example of reconstruction of packet interactions.** TimestampA values are used to construct the sequence of events observed in each router. For example, the packet with ID 31 (P31) is received and sent from router 0 at timestampA=1. Dashed lines indicate packets sent from router0 or router1 to other routers in the network.

router0 and router1 in the first 100 cycles of the simulation, as obtained from our solution. Note that since each router's *pckt_cntr* operates independently, the *timestampA* values are not synchronized across routers, preventing the establishment of a complete global order of events across the network. However, by leveraging common packets as points of reference, we can establish a partial order. For example, in Figure 7, packet 132 is a common packet between routers 0 and 1. Based on that, events relating to packets 31, 87 and 62 in router0 (*i.e.*, the events that occurred before sending packet 132) happened before events associated with packet 143 in router1.

Latency at internal routers. Typically, during the performance validation of NoCs, the debug information that can be collected relies on end-to-end analysis of latencies and throughput. However, our solution has the benefit of providing performance statistics at internal routers, since debug data is collected at every hop along a packet's path. By examining the *pckt_latency* field recorded in the debug data, we can study the average and maximum packet latencies observed at every router and throughout the network's execution. This information facilitates debugging performance bugs, where the execution is functionally correct but does not meet the performance specifications of the design, such as in the case of starvation errors. As an example of the type of results that can be generated, Figure 8 shows packet latencies, averaged over all routers for the various PARSEC benchmarks. Some benchmarks, such as *dedup* and *ferret*, exhibit larger average packet latencies within routers, as compared to others.

We can also plot average packet latencies observed within each router during a specific testbench execution. For example, Figure 9 shows this information for the *dedup* benchmark, where we can observe that router 49 exhibits the highest average packet latency compared to other routers. Using such results, we can identify potential performance bottlenecks in the network. It is also possible to use our scheme to examine packet latency values over time and per router. For example, Figure 10 shows the variation in packet latencies observed at router 49 throughout the execution of *dedup*. Based on that, execution periods of interest can be identified for further analysis, such as the period highlighted in Figure 10, where we record the first significant increase in latency. By examining packet interactions and path reconstruction results for the *dedup* benchmark, we identify the packet associated with this latency and find that it is blocked in router 49 due to congestion in the downstream router 41. Router 41, in turn, has several packets that are also waiting for busy virtual channels. By means of inspection, we realized that the simulated router design was setup to utilize a basic credit-flow mechanism that releases the output virtual channel only after the entire packet is transferred, which amplifies the packet latencies in the presence of congestion. This example highlights how DiAMOND provides high quality diagnosis and traffic-inspection capabilities in post-silicon or emulation environments, inculding the ability to investigate performance flaws in the network. This technique can also be utilized for non-verification purposes early on in design process, particulary during NoC design exploration and performance profiling.

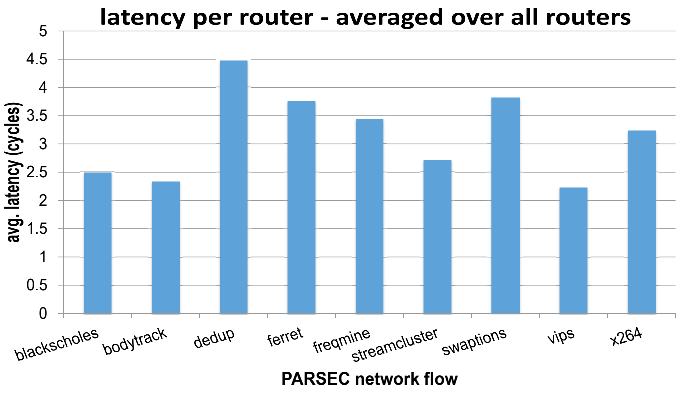

Fig. 8. **Average packet latency at internal routers.** Results are shown for the PARSEC network flow, averaged over all routers

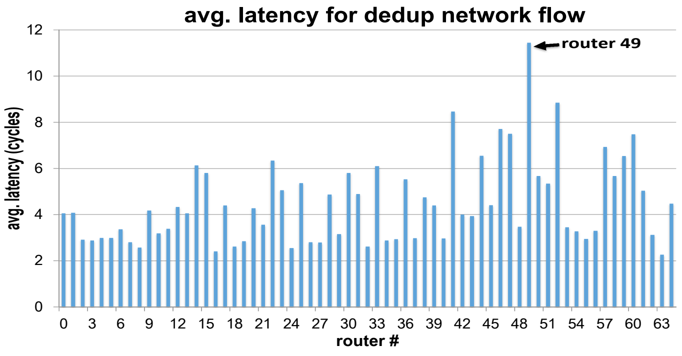

Fig. 9. **Average packet latency for a sample benchmark.** For the dedup network flow, we show the average packet latency observed at each router.

VI. Conclusion

We presented DiAMOND, a debug solution for the post-silicon validation and emulation of networks-on-chip. Targeting the functional validation of the control-flow portion of NoCs, we log debug data during network execution and store it by replacing the data content of packets. Debug information is collected for every packet, at each router along its path, and then systematically written in one of its body flits. In addition, simple hardware checkers are added to routers to monitor execution and flag functional bugs. Upon bug detection, the

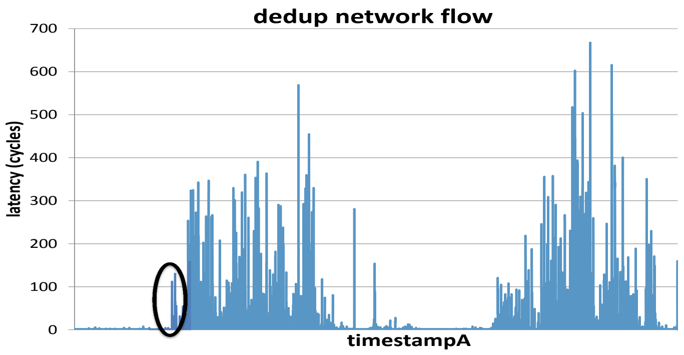

Fig. 10. **Packet latency at router 49 for dedup benchmark.**

collected debug data provides increased observability of network traffic. The analysis process reconstructs the packets' paths, achieving, in most cases, over 80% reconstruction. We also provide several functional, as well as performance, statistics regarding the network's operation, including the sequence of events that occurred within routers and packet latencies observed per router and over time.

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] B. Vermeulen and K. Goossens, "A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs," in *Proc. VLSI-DAT*, 2009.

[2] C. Ciordas, K. Goossens, T. Basten, A. Radulescu, and A. Boon, "Transaction monitoring in networks on chip: the on-chip run-time perspective," in *IES*, 2006.

[3] C. Ciordas, T. Basten, A. Radulescu, K. Goossens, and J. Meerbergen, "An event-based network-on-chip monitoring service," in *HLDVT*, 2004.

[4] R. Abdel-Khalek and V. Bertacco, "Functional post-silicon diagnosis and debug for networks-on-chip," in *Proc. ICCAD*, 2012.

[5] L. Yangfan, L. Peng, J. Yingtao, Y. Mei, W. Kejun, W. Weidong, and Y. Qingdong, "Building a multi-FPGA-based emulation framework to support networks-on-chip design and verification," *International Journal of Electronics*, vol. 97, 2010.

[6] L. Peng, X. Chunchang, W. Xiaohang, X. Binjie, L. Yangfan, W. Weidong, and Y. Qingdong, "A NoC emulation/verification framework," in *Proc. ITNG*, 2009.

[7] N. Genko, D. Atienza, G. De Micheli, J. Mendias, R. Hermida, and F. Catthoor, "A complete network-on-chip emulation framework," in *Proc. DATE*, 2005.

[8] N. Genko, D. Atienza, G. De Micheli, and L. Benini, "Feature - NoC emulation: a tool and design flow for MPSoC," *Circuits and Systems, IEEE*, vol. 7, 2007.

[9] M. Kouadri, M. Abdellah, B. Senouci, and F. Petrot, "Large scale on-chip networks:an accurate multi-FPGA emulation platform," in *Proc. DSD*, Sept 2008.

[10] S. Ma, N. Enright Jerger, and Z. Wang, "Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip," in *Proc. HPCA*, 2012.

[11] K. Anjan and T. Pinkston, "DISHA: A deadlock recovery scheme for fully adaptive routing," in *Proc. IPPS*, 1995.

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

[13] A. Ghofrani, R. Parikh, S. Shamshiri, A. DeOrio, K.-T. Cheng, and V. Bertacco, "Comprehensive online defect diagnosis in on-chip networks," in *VTS*, April 2012.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, 1978.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," 2008.