Functional Post-Silicon Diagnosis and Debug for Networks-on-Chip

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

ABSTRACT

Networks-on-chip (NoCs) have emerged as a favorable solution to provide higher bandwidth interconnects for large chip multiprocessors (CMPs). In order to enhance the interconnect's performance, the NoC is often designed to include complex components and advanced features. Along with the increase in complexity and size, ensuring the functional correctness of the NoC can be particularly challenging This challenge pervades the entire verification effort, and particularly post-silicon validation, due to the lack of observability of the networks complex internal operation.

We propose a post-silicon validation platform that enhances observability of network activity by periodically taking snapshots of the packets in flight. Each node's local cache is configured to store the snapshot logs in a temporary space allocated for post-silicon validation and released at deployment. Each snapshot log is periodically and locally analyzed by a software algorithm, running on the processor's core, in order to detect functional errors. If an error is detected, the snapshot logs are aggregated and additional debug data is extracted. This includes an overview of the traffic in the network at the time surrounding the manifestation of the error, as well as a partial reconstruction of the routes followed by the packets in flight. In our experiments, we found that this approach allows us to detect several types of functional errors, as well as observe over 50% of the network's traffic on average and reconstruct at least half of each of their routes through the network.

1. INTRODUCTION

As silicon technology continues to scale, large chip multiprocessor (CMP) architectures are an emerging solution targeting parallel applications and high performance computing. In today's market, designs such as Tilera CMPs and Intel's SCC have as many as 48-100 cores on chip. To provide the high communication bandwidth required for these cores to communicate among each other, networks-on-chips (NoCs) are utilized. In a typical NoC architecture, processor cores connect to the interconnect through a network interface. Data sent over the network is divided into packets and transmitted between cores through a series of routers following a path determined by the network's routing protocol. In order to provide more bandwidth, better resource utilization, and higher performance, the NoC designs are becoming increasingly complex. Today, networks often have irregular topologies and advanced routing algorithms and router architectures are often designed with advanced features to boost performance. This growth in complexity and size translates to an increase in the difficulty of verifying the functionality of the NoC. Post-silicon functional validation is an important phase in the verification process of hardware designs. Tests run at high speeds directly on the first few silcon prototypes, which allows a deeper and more thorough validation of the state space. However, this comes at the expense of extremely limited observability and controllability of the design's internals, making the detection and debugging of errors very challenging.

In this paper, we address the limitations of post-silicon validation for the NoC subsystem by introducing a novel debug platform that greatly boosts the observability of the network activity and facilitates the detection and debug of functional errors. While, pre-silicon verification, which validates the RTL description of a design, is effective for individual components [6], system-level validation is spotty at best, given the limited scalability and performance of the tools available. In contrast, post-silicon validation offers the high performance necessary to investigate the correctness of system-level functionality in depth and expose complex bugs. Such functional bugs manifest as incorrect traffic behavior and network resource utilization or by preventing the network from making correct forward progress, such as deadlocks and livelocks.

2. CONTRIBUTIONS

To address the challenges outlined above, we present a post-silicon verification platform that aids in detecting and diagnosing functional errors in network-on-chip interconnects in CMPs. We collect information about traffic in-flight during network operation, by instrumenting each router to take period snapshots of the packets traversing it at the time. The snapshots are stored in a designated portion of the L2 local cache corresponding to that CMP node. This space is temporarily reserved for post-silicon debug and released afterwards. The logs of each router effectively provide samples of the traffic observed within the router throughout a test execution. If a functional error manifests, then it af-

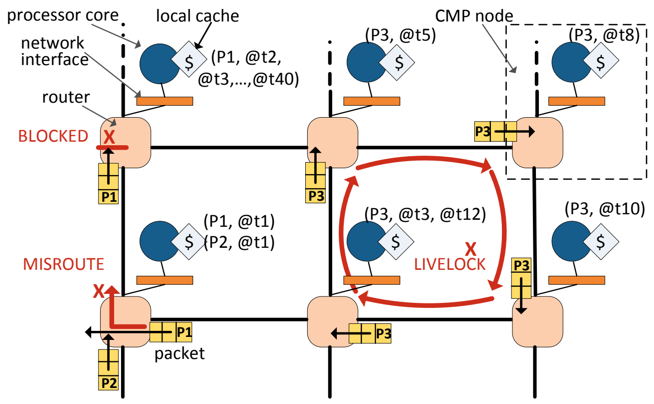

Figure 1: **Overview of our NoC post-silicon validation platform.** We boost observability during post-silicon validation by instrumenting routers to periodically monitor network traffic. We show three possible bugs that our solution can detect and the type of information collected.

fects the behavior of at least one packet. Therefore, we run a software checking algorithm runs on each core to examine the local logs and identify erroneous behavior. If an error is detected, the logs from all nodes are aggregated and used to reconstruct the paths followed by packets in the network. The reconstructed paths provide an overview of the traffic in transit during the time preceding the manifestation of the error (see also Figure 1). Our proposed solution is independent of the specific NoC design, its topology, routing algorithm and router architecture. It is also applicable to a NoC that uses multiple clock domains.

Our solution provides the following main contributions:

- A framework that provides observability of the network operations during post-silicon validation by periodically taking snapshots of routers' contents. The collected data is used to track packets through the network, providing a global overview of the network traffic at the time of the error manifestation. Our experimental evaluation shows that our solution can provide observability of over 50% of packets in most cases and reconstruct at least half of their paths.

- A post-silicon solution that detects and diagnoses functional errors that prevent the network from making forward progress, including deadlocks, livelocks, starvation and misrouting errors.

3. RELATED WORK

Conventional approaches to post-silicon debug augments the design with boundary scan registers (BSRs). Test data can be serially shifted through these BSRs and applied to the component being tested. Test results and traces can be serially read out and transferred off-chip to be analyzed for debugging. Another common approach uses on-chip trace buffers to collect execution traces and then off-loads them for analysis when they are full; however, this comes at the expense of a large area overhead. In contrast, we propose a post-silicon debug platform targeting specifically NoCs, enabling us to tailor our solution for effective functional debugging of the interconnect. Similar to the idea of using on-chip buffers, we collect information about packets in flight, but we store them in the local caches. Data is collected at a central location only if an error is detected in the local logs. Therefore, our approach also has the benefit of eliminating the need to regularly off-load data from all buffers.

Some solutions for post-silicon debugging of NoC designs were proposed in [8], [3] [2]. Vermeulen, *et al.* describes a transaction-based NoC monitoring framework for systems-on-chip, where monitors are attached to master/slave interfaces or to routers. These monitors can filter traffic to observe transactions of interest as well as analyze network performance. [2] proposes adding configurable monitors to NoC routers to observe router signals and generate timestamped events. The events are then transferred through the NoC for off-chip analysis or at dedicated processing nodes. This work was later extended in [3] by replacing the event generator with a transaction monitor, which can be configured to monitor the raw data of the observed signals or to abstract the data to the connection or transaction level. These works propose solutions for increasing NoC observability, but do not demonstrate their use in functional verification. [8] focuses on using the monitors for performance analysis of the network operations. [3], [2] provide a high-level description of the types of events and transactions that can be observed, but do not address their use in detecting and debugging errors. We share with them the idea of monitoring traffic at the routers to provide observability of the network's internal operations. However, we propose a complete framework that selects the exact data to be monitored and uses it to detect functional errors and help in debugging. An additional drawback of the above approaches is that, in order to continuously monitor execution, monitored data either needs to be stored in large buffers or regularly transmitted over the network for analysis. The former increases area and power overheads and the latter regularly perturbs the network's normal execution. The authors of [2, 3] also report a high area overhead (17%-24%) for the monitors that will be no longer needed after system deployment. On the other hand, our framework stores the monitored data in the local cache, analyzes it locally, and transmits it over the network only when an error is detected. Finally, our monitoring hardware introduces a much smaller area overhead (9%).

Other debugging solutions for NoC-based multi-cores and systems-on-chip (SoCs) include [7]. Debug probes are added between each core under debug (CUD) and its network-interface. The probes monitor communication transactions, generate signals to control the CUD, and read the CUD's trace buffers. Control and debug data are transferred between the probes and an off-chip debug controller through the NoC. Similarly in [9,10], probes are added to monitor incoming and outgoing packets of master IPs in an SoC, which are then used to analyze the initiation and completion of transactions. These proposed platforms do not address debugging functional errors in the interconnect itself, which is the target of our work.

4. POST-SILICON VALIDATION PLATFORM

In a typical NoC-based CMP, each processor core and its local cache are connected through a network interface to a router (Figure 1). We assume a general virtual channel worm-hole router architecture, where packets are partitioned into flits, with a header flit marking the beginning of the packet. An incoming packet is first queued in one of the router's input buffer and then it is processed in three

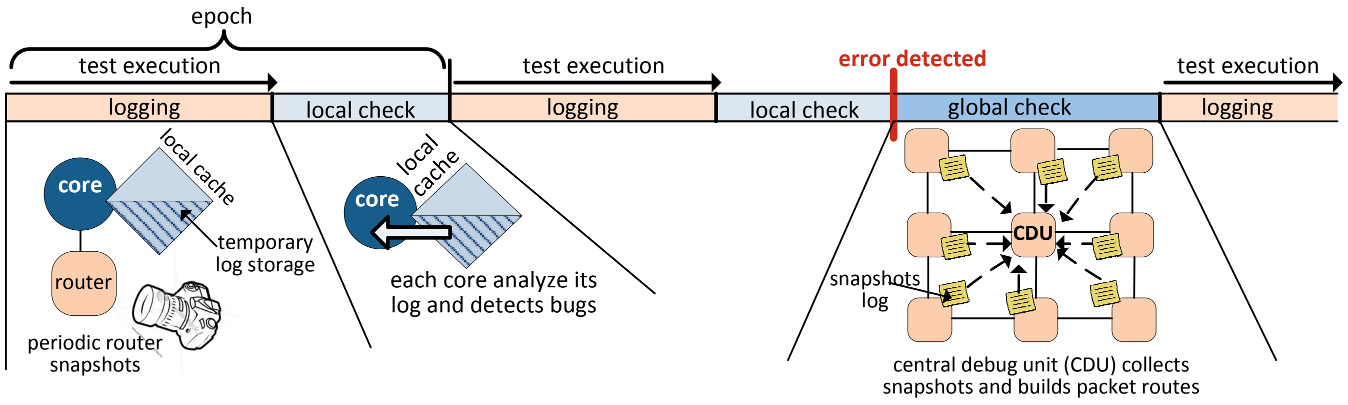

Figure 2: **Execution flow of the NoC debug platform.** Execution is partitioned into epochs, each consisting of a logging phase and a local check phase. During the logging phase, periodic snapshots of each router's contents are logged in the local cache. During the local check phase, the logs are locally analyzed by a software algorithm running on the cores. If an error is detected, the global check phase reconstructs the route of packets in-flight.

stages. First, *route computation* determines the output port for the packet. Then, *virtual channel allocation* determines the output virtual channel. Finally, in the *switch allocation* stage, flits arbitrate for the use of the crossbar. Once the output port and virtual channel are selected for the header flit, the rest of the packet follows.

In our NoC debug platform, execution is partitioned into a series of epochs, each comprising a logging phase and a checking phase, as shown in Figure 2. During the logging phase, routers take snapshots of their internal state. These snapshots are taken at regular intervals and stored in a reserved portion of the local cache attached to that router. When the available space in the cache is exhausted, the logging phase terminates and the network execution is stopped. Then, each core runs a series of software-based checks to analyze the snapshots collected in its local cache and to detect situations where packets are not making forward progress. If such a situation is suspected, in-flight network packets are dropped and the local logs are aggregated at one CMP node. There, a global checking algorithm processes the data to provide debug information, including an overview of the network traffic at the time of the bug occurrence and the reconstruction of the paths followed by packets in flight.

4.1 Logging

During the logging phase, each router is instrumented to take snapshots of packets traversing it at the time, as illustrated in Figure 3. Snapshots are taken periodically at fixed time intervals and the physical clock of the router is used to track time. This interval is set by the user, based on the characteristics of the NoC and the traffic density. To identify the packets to be logged, a router looks for header flits stored in its input buffers. This is accomplished by augmenting the router with one *header buffer* for each input buffer. When a router receives a packet, it stores the packet in its input buffer and stores a copy of the header flit in the corresponding header buffer. The size of the header buffer depends on how many packets can be in a router's input buffer at any point in time, which in turn depends on the minimum number of flits in a packet. When a header flit is identified, the snapshot hardware can extract its source and destination nodes. In addition, we require that the header flit also includes a packet ID (unused space in the header is often available). This ID is provided by a sequential number

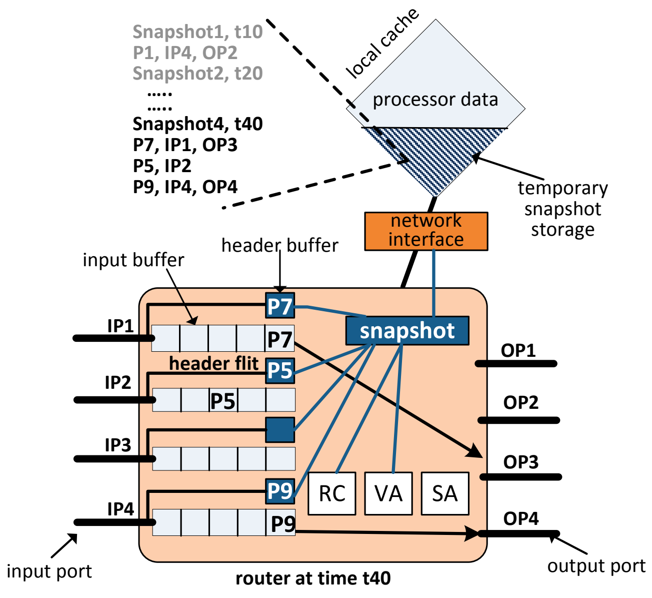

Figure 3: **Logging.** Routers take periodic snapshots of packets. A header buffer is added to every input buffer to keep track of the header flits. The snapshot hardware captures header data and information from the route computation (RC) and virtual channel allocation (VA) modules.

generated at the source node, which, along with the source and destination, forms a unique identifier of that packet. It is also useful to log additional information about the packets, depending on their status within the router. For example, packets that have completed the route computation phase are assigned an output port, so logging the output port allows us to determine the downstream router.

Finally, snapshots also store the physical time at which they are taken. However, if the network uses multiple clock domains, we also rely on the notion of logical time implemented through Lamport clocks [5], where every router has a logical clock that is advanced when a new packet is received. In this setup, packet headers also include a timestamp that monotonically increases with every hop. When a router receives packets, it sets its logical clock to the maximum timestamp of the received packets and then increments it. When a packet leaves the router, its timestamp is updated to the logical time of the router. Thus, in the case of multiple clock domains, a snapshot entry also includes the logical timestamp of the packet. Note, all entries in the same snapshot have the same physical timestamp, we therefore store it only once for the entire snapshot.

Overall, every snapshot consists of several entries, one for each packet. Every entry contains a packet ID (counter, source, destination), input port, input virtual channel, output port (if allocated), output virtual channel (if allocated), and its logical timestamp. Figure 4 shows the information logged in each snapshot. In our experimental platform described in Section 5, the size of a snapshot was at most 57B (assuming 20 bits for the physical timestamp, 15 bits for the logical timestamp and 20 bits for packet ID). The collected snapshot entries are sent through a dedicated link to the network interface and stored in a designated portion of the local cache. When available storage is full, the corresponding node transmits a flag through a dedicated link, halting the network temporarily and initiating the local check phase.

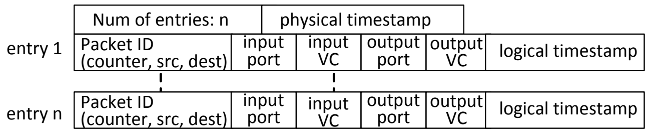

Figure 4: **Snapshot format.** A snapshot consists of several entries, one for each packet in the router.

4.2 Local Checks

Our debug platform targets errors that prevent the network from making correct forward progress. During the local check phase, each core analyzes the snapshot log from its local cache to detect signs of such errors. The algorithm first iterates through the snapshots and groups snapshot entries according to the packet ID. Thus, each packet becomes associated with a list of entries that reflect how the status of the packet changed within that router.

Forward progress can be hindered if a packet is blocked in a router, in the case of a starvation or deadlock bug, or is not advancing correctly towards its destination, in the case of a livelock and a misroute bug. Therefore, we consider each of these cases separately.

Livelock. A network livelock exists if a packet is being transferred through routers but not advancing to its destination. Since the checking algorithm running on each core has only access to its local snapshot log, livelocks must be detected locally at the router. For a network with a finite number of nodes, a livelocked packet will eventually traverse the same router twice. Provided that the epoch length is long enough for the livelock cycle to form, such errors can be detected locally. In Figure 5 (lines 3-7), the algorithm retrieves the physical timestamp of the packet's snapshot entries. If the difference in time between two successive snapshot entries is greater than the snapshot interval, then they were captured in non-consecutive snapshots. This is an indication that the packet traversed the router at different non-consecutive times and the algorithm flags a livelock.

Starvation. A starvation error exists if a packet is temporarily blocked waiting to acquire resources that are given to other packets. Packets traversing the network can only be blocked in a router's input buffers, as this is the only storage available in the network. Therefore, a starved packet must appear in several consecutive snapshots in a router. Hence, the checking algorithm determines the number of consecutive snapshots in which each packet appears (Figure 5, line 9), and based on the snapshot rate, it deduces how long the packet has been waiting and flags a starvation error when this value exceeds a user-set threshold.

Deadlock. A network deadlock exists if packets are blocked waiting on each other to free resources in a way that none of them can advance forward. At the network-level, a deadlock can be identified by the existence of a cyclic dependency of resources, but identifying a deadlock at the router-level by examining only the local snapshot log reduces to the problem of identifying a blocked packet. Similar to the starvation bug, a packet is blocked if it appears in consecutive snapshots. However, a deadlocked packet is permanently blocked, which means it must also be seen in the latest snapshot. Thus, the local check algorithm checks if any of the packets in the snapshots satisfy both these conditions and flags them as deadlocked (lines 10-11 in Figure 5). Note that long starved packets could be misclassified as deadlocks.

Misroute. With deterministic routing, a packet traveling between a source and destination pair should always go through the same route, based on the routing algorithm. A misroute occurs if a packet is routed to a node that is not on this path (irrespective of whether the packet eventually reaches its final destination). To detect such errors, we first assume that all valid paths between each source-destination pair are known, since they can easily be collected theoretically or experimentally beforehand. This information is stored in each local cache in the form of a bit vector that indicates, for each source-destination pair, whether the local router is part of the valid route. Therefore, to detect misroutes, the local check algorithm iterates through the snapshot entries, obtains the source and destination of each entry, and checks it against the valid paths information (lines 13-16 in Figure 5).

In the absence of errors, the snapshot data is cleared and the NoC resumes execution. However, if an error is detected, the logs are aggregated at the central debug unit (CDU), which can be any of the network nodes. In-flight packet are dropped and the logs are sent from one cache at a time to the CDU. Sending from one node at a time reduces the complexity of network operations during the transmission of the logs, boosting the likelihood of error-free transmission, since it only uses basic operations.

Sampling. To reduce the time spent in the local check phase, we also provide an optimization that trades-off the ability to detect errors with the execution time of the local check algorithm. Instead of analyzing the entire snapshot log, each core can downsample the information, such that it only looks at a uniformly distributed fraction of the snapshot entries. This *sampling rate* is another user-defined value, and we evaluate its trade-offs in Section 5.1.

4.3 Global Check

The goal of the global check phase is to provide useful information that can facilitate debugging the detected error. It combines the collected snapshots to reconstruct the paths of observed packets, and it gives an overview of the traffic that passed through the network during the logging phase. Snapshots entries pertaining to the same packet are grouped together. Packet routes are then reconstructed by sorting these entries in increasing order of snapshots' physical timstamps, when a global clock is present, or the logical timestamps when the network uses multiple clock domains.

```
LocalCheck (snapshotLog){
 foreach packet in snapshotLog {

  foreach ntr in GetSnapshotEntries(packet) {
   time = PhysicalTimeEntry(ntr);
   next_time = PhysicalTimeEntry(ntr+1)
   if (next_time - time > snapshotInterval)
    FlagError(Livelock)}

  if (CountEntries(packet) > threshold)
    if packet in lastSnapshot(snapshotLog)
       FlagError(Deadlock)
     else FlagError(Starvation)

   src = GetSrc(packet)
   dest = GetDest(packet)
   if router !InPath(src, dest)
    FlagError(Misroute) } }
```

Figure 5: **Local check algorithm.** To detect livelock, the algorithm checks if a packet appears in non-consecutive snapshots. For blocked packets (deadlock and starvation), it checks if a packet appears in several consecutive snapshots. Finally, misroute errors are detected by comparison with the set of valid paths between the packet's source and destination.

We also use each entry's input port and output port to try to reconstruct the path beyond the router in which the packet was observed. We determine the upstream (downstream) router, based on the input port (output port) field and the network topology.

Besides reconstructing packet routes, the global check algorithm can highlight the packets that were present in each router at the time the snapshot was taken, exposing interactions that possibly triggered the error. For example, by examining a router's snapshot log, we can determine a subset of the packets that traversed it and deduce the router's internal states, such as the buffers that were in-use and the virtual channel and output port allocations at the time.

5. EXPERIMENTAL EVALUATION

To evaluate our debug platform, we modeled a CMP interconnect with Booksim, a cycle accurate C++ based network simulator [4]. We considered our baseline system to be an 8x8 mesh NoC with input-queued virtual channel routers. Each router has 5 ports, 2 virtual channels and 8 flit-buffers. We augmented the simulator so that routers take periodic snapshots of packets and we implemented the local check functions. We simulated two types of workloads: directed random traffic (uniform, bitcomp) and applications from the PARSEC benchmark suite [1].

	snapshot interval=10 cycles			snapshot interval=50 cycles		
injected bugs	no sampling	50% sampling	20% sampling	no sampling	50% sampling	20% sampling
misroute	19%	14%	2%	6%	0%	0%
deadlock	100%	100%	100%	100%	100%	100%
livelock	100%	100%	100%	100%	100%	100%
starvation	24%	7%	2%	0%	0%	0%

Table 1: **Error detection rate** for our four types of bugs. The results are reported for two snapshot intervals, with and without local check sampling.

5.1 Error Detection

We first analyzed our platform's ability to detect functional errors. We modeled four types of bugs in the baseline system, each representing an error that would prevent the network from making correct forward progress. These include a deadlock and a livelock bug, a misrouting bug, where

		snapshot rate 10 cycles		snapshot rate 50 cycles	
low injection	%packets observed	54%		20%	
	avg. path reconstruction	53%		30%	
	avg. reconstruction rate of faulty packets per bug type	misroute	43%	Misroute	0%
		deadlock	63%	Deadlock	54%
		livelock	74%	livelock	66%
		starvation	36%	starvation	0%
medium injection	% of packets observed	67%		28%	
	avg. path reconstruction	52%		31%	
	avg. reconstruction rate of faulty packets per bug type	misroute	55%	misroute	0%
		deadlock	67%	deadlock	49%
		livelock	57%	livelock	48%
		starvation	47%	starvation	0%
high injection	%packets observed	85%		50%	
	avg. path reconstruction	58%		35%	
	avg. reconstruction rate of faulty packets per bug type	misroute	60%	misroute	0%
		deadlock	77%	deadlock	54%
		livelock	73%	livelock	56%
		starvation	0%	starvation	0%

Table 2: **Diagnosis capability** is evaluated by measuring the % of packets observed, the % of each path that we could reconstruct, and the % reconstruction for the erroneous paths.

a packet is misrouted once along its path to the destination node, and a starvation bug, where a packet is temporarily prevented from acquiring the resources it needs to progress along its path. The bugs were injected in a randomly chosen router or set of routers by modifying the simulator to model the effect of the bug on the packets in transit at the time. We ran both the random traffic and PARSEC workloads, while triggering each bug once during the simulation and repeated each experiment with 11 random seeds.

Table 1 shows the detection rate when simulating bitcomp traffic over our four bugs and two snapshot intervals (every 10 cycles and every 50 cycles). In addition, we varied the sampling rate of the local check algorithm, which, as explained in Section 4.2, constitutes a trade-off between detection coverage and the time it takes to complete the local check phase. In these experiments, the threshold for detecting starvation and deadlock was set to 100 snapshots. Results show that deadlock and livelock bugs are always detected, whereas misroute and starvation have a much lower detection rate (0% -24%). This is because, when a packet is deadlocked or livelocked, it remains in this state from when the bug manifests until the end of the simulation, and thus it has a high probability of being captured by the snapshots. On the other hand, misroute and starvation errors are transient and the affected packets can be missed and never observed. This effect is more pronounced when the snapshot interval is increased to 50 cycles. In addition, if sampling is activated during the local check phase, the detection of misroute and starvation decreases, again because of the transient nature of these errors. Whereas, local check sampling does not affect the detection of livelock and deadlock. Finally, we noticed that for the snapshot interval of 10 cycles, the simulations where the traffic injection rate was high (close to network saturation), exhibited false positives due to the false detection of starvation bugs. Starvation errors were flagged even before our bugs were injected. This is because the network is highly congested and the chosen detection threshold was small. However, at a snapshot interval of 50 cycles, no false positives were reported.

5.2 Error Diagnosis

We also evaluated the quality of information that can be obtained from the snapshots when they are aggregated dur-

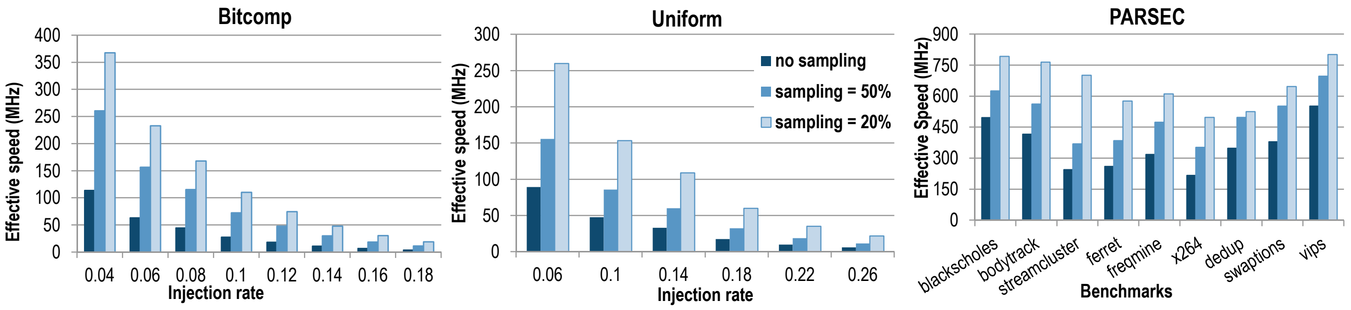

Figure 6: **Effective simulation speed of our NoC debug platform with varying local check sampling rates.** The baseline system without our debugging functionality is assumed to be running at 1GHz.

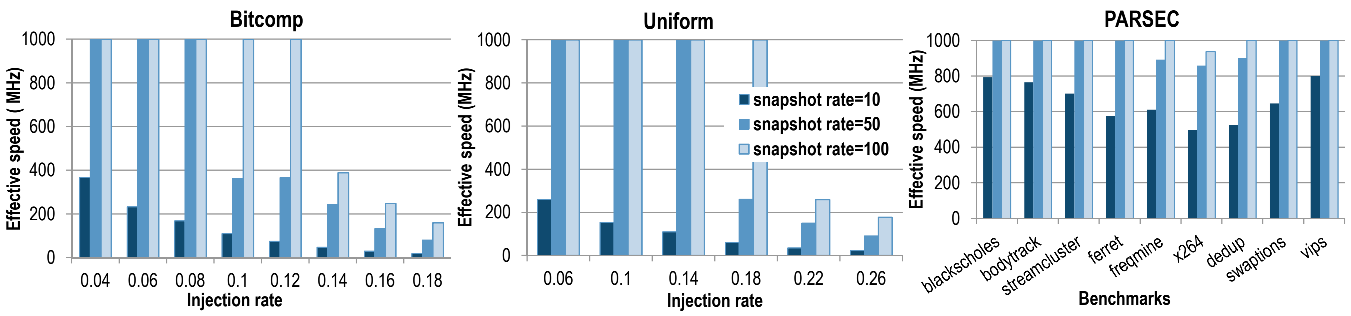

Figure 7: **Effective simulation speed of our NoC debug platform with varying snapshot interval.** The baseline system without our debugging functionality is assumed to be running at 1GHz

ing the global check phase. Table 2 highlights the results for bitcomp random traffic at low, medium and high injection rates and two snapshot intervals (10 cycles and 50 cycles) and with a sampling rate of 50%. We particularly looked at 3 measurements. First, we calculated the percentage of packets that were observed in at least one snapshot out of the total number of packets injected in the simulation. Second, we looked at path reconstruction, which is the average percentage of each route we were able reconstruct from the aggregated snapshots. Finally, we measured the path reconstruction of the packets that were detected as faulty (*i.e.*, the packets where the detected error manifested).

For a snapshot interval of 10 cycles, the percentage of observed packets is 54% at low injection and increases to 85% at high injection. This increase is due to the fact that at higher injection rates, the network is more congested and routers have more packets traversing them, which allows each snapshot to capture a larger fraction of the packets in flight. As for path reconstruction, we note that on average 52% to 58% of each route was reconstructed from the snapshots. When looking at the path reconstruction of the erroneous packets, we notice that when the bug is detected, we can reconstruct on average 36%- 60% of its path. For the starvation bug at high injection, the path reconstruction of the faulty packet is reported as 0%, since the error was not detected in any of the runs. Even though at higher injection, the network is more congested and starvation is more likely to occur, we maintained the same bug injection rate: one starvation bug injected once during each run. Therefore, with more traffic in-flight and given the probabilistic nature of the snapshot algorithm, the erroneous packet that was affected by the bug, was never observed.

When the snapshot interval is increased to 50 cycles, the percentage of packets that are observed in the collected snapshots decreases to 20% at low injection and 50% at high injection. Similarly, the overall average path reconstruction decreases to 35%. With higher snapshot intervals, snapshots are taken less frequently and thus more packets are missed. Therefore, the snapshot interval directly influences network observability. Moreover, the impact of the chosen snapshot interval varies depending on the injection rate. For test cases that have low traffic injection, a smaller snapshot interval is required to observe at least 50% of all packets. However, at high injection rate, a larger snapshot value would be sufficient to achieve the same result, because of the higher congestion in the network.

5.3 Debug Platform Simulation Speed

Our NoC debug platform periodically stops network execution to locally check the collected snapshot logs. We therefore evaluated the effective speed of the baseline system with the snapshot and local check functionalities enabled. To this end, we simulated the system with PARSEC and random traffic and ran these simulations on a 2.4GHz Core2 Quad machine. We utilized the x86 timestamp counters to calculate the average execution time, in cycles, of the local check algorithm. In these experiments, we assume that the baseline system (the cores and the NoC) without our NoC debug functionality is operating at 1GHz. We also assume that 30KB of the local L2 cache is reserved for the snapshot log, when simulating random traffic workloads and 10KB when simulating the PARSEC benchmarks. Note that the snapshot storage is around 10% or less of a typical L2 cache size of 256KB. In addition, the lower injection rate of the PARSEC benchmarks required us to choose a lower storage size so that the local log fills up and triggers a local check at least once during the benchmark's execution.

Figure 6 shows the effective simulation speed of the system equipped with our NoC debug solution, using a snapshot interval of 10 cycles and while varying the local check sampling rate. The effective simulation speeds only take into account the overhead of the local checks triggered during each run. For the random traffic, we vary the injection rate from low injection (when the network is close to zero-load latency) to high injection (before network saturation). Without sampling, the platform's effective simulation speed is 38MHz for the random traffic and 359MHz for the PARSEC benchmarks on average. However, with sampling, the platform's effective speed can be increased to an average of 130MHz and 660MHz respectively. Overall, this performance is still several orders of magnitude better that what can be achieved in pre-silicon, all while providing much better observability and debug information than traditional post-silicon solutions. Moreover, performance can be boosted further by sampling.

We also evaluated the effect of varying the snapshot interval on the platform simulation speed, for a fixed sampling rate of 20%. With higher snapshot intervals, local checks are invoked less frequently and the overall effective speed of the simulations is higher. This can be observed in Figure 7. For example, when the snapshot interval is increased from 10 cycles to 50 and a 100 cycles, we observe an increase in the simulation speed of bitcomp traffic by a factor of 3 and 7, respectively. However, as explained in Section 5.2, a larger snapshot interval reduces the observability of in-flight packets and the ability to reconstruct their paths. A similar trend is observed for the uniform and PARSEC workloads. However, note that for low injection, increasing the snapshot interval might not be feasible, as was the case of the PARSEC benchmark that would never trigger a local check.

Since the snapshot interval and the local check sampling rate constitute a trade-off between the execution speed of our post-silicon validation platform and its ability to detect and diagnose errors, we propose making them configurable parameters that can be tuned depending on workloads and traffic patterns.

5.4 Area

We also evaluated the area overhead of the router modifications that are needed to capture the local snapshots. The router additions are described in Section 4.1 and illustrated in Figure 3. We synthesized the modified baseline router with the Artisan 45nm target library. The baseline router area was $0.075mm^2$ and the area overhead is 9%.

6. PLATFORM PARAMETERS

Our solution provides high observability and debug information. However, some of its aspects have limitations. For instance, the snapshot interval may affect the ability to detect transient bugs, such as misroute and starvation. Moreover, depending on when the local logs fill up and trigger a local check, it is possible for the platform to miss a livelock bug, if a livelock cycle does not complete within the logging epoch or a starvation bug if the bug manifests very close to the end of an epoch. Note, however, that good tuning of the several parameters available can overcome most or all of these situations. Moreover, multiple runs of a same test with different parameter settings can also boost the detection capabilities of the solution. For instance, misroute and starvation bugs are best detected with a small snapshot interval. Moreover, a larger local check sampling rate improves the platform's detection accuracy, but decreases the effective execution speed. Finally, the designated portion of the cache reserved for the local logs is another parameter which must be tuned. Benchmarks with lower injection rates may require a smaller storage to trigger the local check frequently enough to detect errors. Note also that although we utilize a portion of the cache to store the logs, this does not hinder the value of our solution. In fact, the smaller available cache space would likely generate more traffic and better exercise the network, since conflict and capacity misses would be more pronounced.

7. CONCLUSION

We presented a post-silicon solution to support the functional verification of networks-on-chip by increasing the observability of the network's internal operation and providing debug information to facilitate the diagnosis and debugging of errors. Our platform targets functional errors that prevent the network from making correct forward progress, such as deadlock, livelock and starvation errors. Routers periodically take snapshots of packets traversing them and log these snapshots in a reserved portion of each processor's local cache. When the space allocated for the logs is exhausted, a software algorithm running on each cores examines its local snapshot log for incorrect packet behavior. Once an error is detected, the local logs are combined and additional debug information is extracted. Results show that our debug platform can effectively collect information critical to the detection and diagnosis of functional errors during post-silicon validation.

Acknowledgments. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

8. REFERENCES

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

[2] C. Ciordas, T. Basten, A. Radulescu, K. Goossens, and J. Meerbergen. An event-based network-on-chip monitoring service. In *Proc. HLDTV*, 2004.

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

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

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

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

[7] S. Tang and Q. Xu. A multi-core debug platform for NoC-based systems. In *Proc. DATE*, 2007.

[8] 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.

[9] H. Yi, S. Park, and S. Kundu. A design-for-debug (DfD) for NoC-based SoC debugging via NoC. In *Proc. ATS*, 2008.

[10] H. Yi, S. Park, and S. Kundu. On-chip support for NoC-based SoC debugging. *IEE Trans. on Circuits and Systems*, 57(7), 2010.