Post-Silicon Platform for the Functional Diagnosis and Debug of Networks-on-Chip

Rawan Abdel-Khalek, University of Michigan
Valeria Bertacco, University of Michigan

Submitted to DCMP12 Special Issue
The increasing number of units in today's systems-on-chip and multicore processors has led to complex intra-chip communication solutions. Specifically, networks-on-chip (NoCs) have emerged as a favorable fabric to provide high bandwidth and low latency in connecting many units in a same chip. To achieve these goals, the NoC often includes complex components and advanced features, leading to the development of large and highly complex interconnect subsystems. One of the biggest challenges in these designs is to ensure the correct functionality of this communication infrastructure. To support this goal, an increasing fraction of the validation effort has shifted to post-silicon validation, because it permits exercising network activities that are too complex to be validated in pre-silicon. However, post-silicon validation is hindered by the lack of observability of the network's internal operations and thus, diagnosing functional errors during this phase is very difficult.

In this work, we propose a post-silicon validation platform that improves observability of network operations by taking periodic snapshots of the traffic traversing the network. Each node's local cache is configured to temporarily store the snapshot logs in a designated area reserved for post-silicon validation and relinquished after product release. Each snapshot log is analyzed locally by a software algorithm running on its corresponding core, in order to detect functional errors. Upon error detection, all snapshot logs are aggregated at a central location to extract additional debug data, including an overview of network traffic surrounding the error event, as well as a partial reconstruction of the routes followed by packets in flight at the time. In our experiments, we found that this approach allows us to detect several types of functional errors, as well as observe, on average, over 50% of the network's traffic and reconstruct at least half of each of their routes through the network.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiprocessors

Additional Key Words and Phrases: Networks-on-chip, post-silicon validation, functional correctness, performance monitoring

1. INTRODUCTION

Driven by the continuous scaling of silicon technology, large chip-multiprocessors (CMPs) are becoming a prevalent architecture, targeting parallel applications and high performance computing. In today's market, designs such as the Intel's Xeon Phi co-processors and the Tilera's multiprocessor chip family embed more tha 50 cores on a single chip. With the increasing number of processor cores, it is highly important to have a communication medium that provides adequate bandwidth and allows these cores to communicate effectively. Therefore, networks-on-chip (NoCs) have emerged as a scalable and favorable interconnect solution. In a typical NoC architecture, each processor core connects to the interconnect through a network interface. Messages sent over the network are divided into packets, which are then transmitted between cores through a series of routers, following a path determined by the routing protocol. With increasing demands for higher bandwidth and better performance, NoC design complexity is on the rise. Routers often incorporate advanced features such as speculation, escape channels, and aggressive and prioritized allocation schemes [Al Faruque et al. 2006]. Power constraints drive designs towards dynamic power management as well as adaptive resource and network reconfiguration [Jafri et al. 2012; Mishra et al. 2010; Kim et al. 2011] . Irregular topologies, adaptive routing algorithms and dynamic resource allocation are some of the solutions proposed to adapt network characteristics to application demands [Krishna et al. 2011] . Complexity is also introduced when local routing and allocation decisions are optimized by incorporating more global knowledge

of network traffic and resource availability. The growth in complexity, along with the large size of these systems, translates to increased difficulty in verifying the functionality of the NoC.

Pre-silicon validation is one of the main steps in the verification process of any hardware design. It consists of simulation-based verification, which simulates the RTL description of the design, and formal verification, which theoretically proves the design's functional correctness. Both these approaches do not scale-well for large and complex systems, as simulation-based methods are very slow and can not exhaustively explore the entire design space, and formal verification suffers from the state space explosion problem. These methods often work well for the verification of small or individual components of the NoC, such as the arbiter or the buffers or even the router. However, the limited scalability and performance of these tools, prevent them from exercising and validating the entire design space, particularly with regards to the validation of system-level or inter-module operations. In recent work [Parikh and Bertacco 2011], the authors distinguish between router properties that are amenable to pre-silicon verification and global network properties that can not be fully verified in pre-silicon and for which they propose runtime execution checkers.

Post-silicon functional validation starts when the first few silicon prototypes become available and it is used to ensure that the new chips are functioning correctly and in compliance with the specifications. Test regressions and applications run directly on the manufactured hardware, and thus a major advantage of this approach is that tests execute on the machine at or close to nominal chip performance. Consequently, engineers can run much longer tests, enabling a deeper and more thorough validation of the hardware design. This is particularly significant when compared to earlier verification phases, which are much slower and less scalable. From this perspective, post-silicon validation has the opportunity to expose functional bugs that might have been missed during pre-silicon verification. However, despite its high-speed and high-coverage advantages, post-silicon validation suffers from extremely low observability and controllability of the design's behavior. Observability is often restricted to signals connected to the chip's input and output pins, as well as a small number of signals within the design. As a result, when tests fail, it is difficult to detect and debug the failure as verification engineers can not fully observe the internal operations of the design.

In this paper, we address the limitations of post-silicon validation targeting specifically NoC subsystems. We introduce a novel debug platform that greatly boosts the observability of the network activity and facilitates the detection and debug of functional errors. Post-silicon validation offers the high performance necessary to investigate the correctness of network-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, starvations and livelocks. Without a framework to effectively detect and localize these errors, NoC verification would be extremely challenging.

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 multi-cores and systems-on-chip. We collect information about the traffic in-flight during a network's normal operation, by instrumenting each router to periodically take snapshots of the packets traversing it at the time. The snapshots are stored in a designated portion of the local L2 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

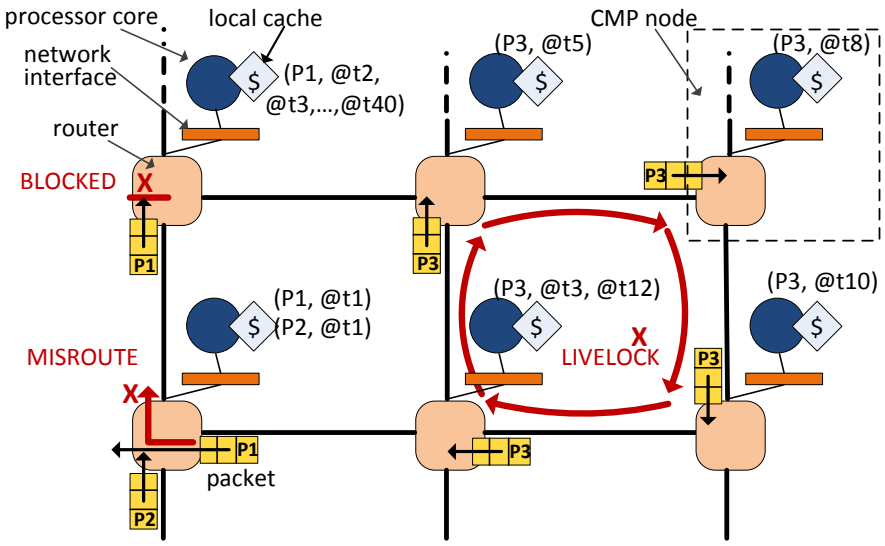

Fig. 1. **Overview of our NoC post-silicon validation platform.** Our solution boosts observability during post-silicon validation by instrumenting routers to monitor network traffic over time. The data collected is used to track packets in transit, detect functional errors, and provide detailed debug information. In the figure, we show three possible bugs that our solution can detect and we illustrate qualitatively the type of information collected by the routers over time.

throughout a test execution. If a functional error manifests, then it must affect the behavior of at least one packet. Therefore, we run a distributed software checking algorithm 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 flight at the time surrounding the bug occurrence. The reconstructed paths provide an overview of the traffic in transit during the time preceding the manifestation of the error. Figure 1, depicts a high-level illustration of our technique. We show a portion of an example network consisting of six routers, each connected through a network interface to a processor core and a local cache. Packets in-flight, shown passing through the different routers, are captured by periodic snapshots. These snapshots are accumulated in each router's corresponding local cache. For example, the snapshot log of the top-left router, indicates that packet P1 appeared in that router at times t1 through t40. During the local check phase, these logs are analyzed and erroneous behavior is detected. Figure 1 highlights three of the types of errors that our solution can detect: deadlock, misroute and livelock.

Our proposed solution is independent of the specific NoC design, its topology, routing algorithm and router architecture. It is also applicable to a system-on-chip where some network routers are not associated with a processor core and a local cache. In that scenario, those routers can be augmented with additional buffers to store the snapshot log and then configured to periodically transfer their logs to neighboring caches for analysis. If resources are limited or if traffic congestion is a potential issue, the system-on-chip can be designed so that snapshot hardware is limited to routers that are attached to a local cache and processing core, in which case error detection and diagnosis is performed based only on the data gathered at processor nodes. Lastly, our solution also extends to NoCs that use multiple clock domains.

Our work provides the following main contributions:

- A post-silicon solution that detects and diagnoses functional errors preventing a network from making forward progress, including deadlocks, livelocks, starvation and misrouting errors. We find experimentally that we can detect bugs with an average latency of 50,000 to 300,000 cycles from their occurrence. Moreover, our diagnosis solution provides observability of over 50% of the packets in most cases, and it reconstructs at least half of their paths.
- A set of on-chip hardware components 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 on-chip hardware additions entail minimum area overhead.
- A complete post-silicon validation flow methodology infrastructure based on the framework discussed above. We discuss how our solution can be used to achieve high quality validation coverage in post-silicon, and, in particular, how to tune the parameters of our framework to a NoC design under test.
- A method to utilize the proposed framework and hardware-based sampling infrastructure to monitor NoC performance during runtime execution by taking snapshots of end-to-end latency values at every router. Users can estimate the network latency, which in turn reflects upon the CMP's performance.

3. RELATED WORK

Several solutions have been proposed for the post-silicon functional validation of hardware designs. In this section, we first give an overview of some of the recent network-on-chip designs and advancements that often lead to greater design complexity. We then describe related post-silicon validation works, while grouping them into four major categories: generalized approaches for post-silicon validation, validation techniques for processor and memory models, previous work in NoC post-silicon validation, and debugging solutions for NoC-based systems.

NoC designs. As the number of cores on chip increases and with the ever-growing demands for higher performance, lower power consumption and better QoS guarantees, various NoC designs and advancements are continuously proposed. For example, several works have presented approaches to dynamically manage and reduce the network's power consumption [Jafri et al. 2012; Mishra et al. 2010; Kim et al. 2011] . Others have proposed alternative scheduling [Stuijk et al. 2006], resource reservation methods [Li et al. 2008] and arbitration schemes that achieve better QoS or provide QoS bounds for certain types of traffic [Al Faruque et al. 2006]. Several works have also described changes in network topology [Das et al. 2009] and features in order to better meet the demands of applications running on the CMP [Krishna et al. 2011]. With these added features and increasing complexity, the functional verification of the NoC is a challenging task.

Common approaches to post-silicon debug utilize boundary scan registers (BSRs) [IEEEstd1149.1 1990]. Test data can be serially shifted through the BSRs and applied to the component being tested and test results and traces can be serially read out and transferred off-chip to be analyzed for debugging. A main drawback of this technique is that execution has to be stopped at regular intervals so that debug data can be serially sent off-chip for analysis. Another conventional approach uses on-chip trace buffers to collect execution traces and then off-loads them for analysis when they are full [Abramovici et al. 2006].

Numerous research ideas proposed in this area focus on *trace signal selection* [Ko and Nicolici 2008; Liu and Xu 2009; Yang and Touba 2009; Ko and Nicolici 2010]. For example, [Ko and Nicolici 2008; Liu and Xu 2009; Chatterjee et al. 2011] propose using the trace signals to restore the states of unmonitored signals. Thus, they develop a

restoration ratio metric to quantify the restoration quality of signals and to be used to guide a heuristic for trace signal selection. Other works target *data compression* or the reduction of the volume of data that needs to be transferred off-chip [Panda et al. 2010; Vishnoi et al. 2009; Panda et al. 2011; Lai et al. 2009]. Moreover, some proposals address *trace data transfer* [Liu and Xu 2009; Vermeulen et al. 2001; Abramovici 2008], where, for example, [Liu and Xu 2009] present an interconnection network design to transfer monitored trace signals to the trace buffer or port. As opposed to these general approaches, 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 and do not require the addition of large buffers. Once the designated cache space is full, data is first processed locally and only if and when an error is detected, it is collected at a central location for additional analysis and debugging. Therefore, our two-phase analysis approach has also the benefit of eliminating the need to regularly off-load data from all buffers.

Several post-silicon debug solutions focused on the **validation of processor designs** [Park et al. 2009; Park et al. 2010; Rotithor 2000; Wagner and Bertacco 2008]. For example, in [Park et al. 2009], authors record control flow and data values for instructions passing through the pipeline during execution. This data is later scanned out and analyzed to localize bugs. In addition, other recent approaches have been developed for the post-silicon validation of coherence [DeOrio et al. 2008] and consistency [DeOrio et al. 2009; Lv et al. 2011] of the memory subsystem in multi-core architectures. In contrast our solution focuses on the post-silicon validation of network-on-chip and our detection and debug mechanism targets functional errors that prevent the network from making correct forward progress,

In the context of **post-silicon validation for networks-on-chip**, 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 [Vermeulen and Goossens 2009]. These monitors can analyze performance, filter traffic to monitor transactions of interest, and verify the integrity of the data through checksum calculations. [Ciordas et al. 2004] 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 [Ciordas et al. 2006] 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. [Vermeulen and Goossens 2009] focus on using the monitors for performance analysis of the network operations. [Ciordas et al. 2006; Ciordas et al. 2004] 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. Moreover, the types of events and transactions would depend on the network-on-chip design that is being tested. We share with them the idea of monitoring traffic at the routers to provide observability of the network's internal operations. Moreover, in contrast with their solutions, we propose a complete framework that selects the exact data to be monitored, independently of the router architecture or the network's routing protocol, and then uses it to detect and debug functional errors. An additional drawback of these approaches is that, to continuously monitor execution, data must either be stored in large buffers or regularly transferred over the network for analysis. The former increases area and power overheads, and the latter perturbs the network's normal execution. The authors of [Ciordas et al. 2006; Ciordas et al. 2004] also report a high area overhead (17%-24%) for 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 if 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 [Tang and Xu 2007]. 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 [Yi et al. 2010; 2008], 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. In these solutions, the proposed platforms make use of the NoC-based interconnect to debug the cores or the communication protocol in the system, but they do not address debugging functional errors in the interconnect itself, which is the target of our work.

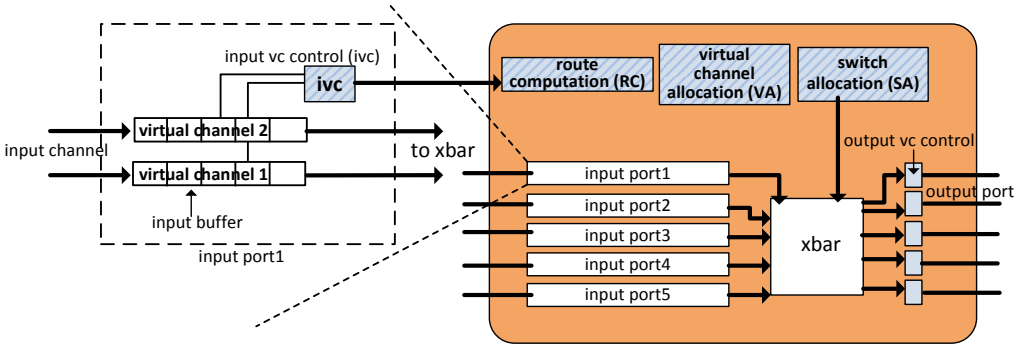

Fig. 2. **General architecture of a virtual channel router.** The figure shows a router with two virtual channels per input port. A received packet is first stored in one of the input buffers. The route computation (RC) stage determines the output port to which it will be sent. The virtual channel allocation (VA) unit assigns an output virtual channel and the switch allocator (SA) arbitrates for the use of the crossbar.

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 buffers, after which it is processed through three main stages. First, *route computation* determines, based on the network's routing algorithm, the output port to which the packet will be sent. This is followed by *virtual channel allocation*, which 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. Figure 2 shows a high-level overview of the router architecture, highlighting the three stages. The example router has five input ports, with each including two virtual channel buffers and an input virtual channel control logic. After route computation and virtual channel allocation, the flits traverse the cross-bar to their designated output ports.

In our post-silicon platform, execution is divided into a series of epochs, each consisting of a logging phase and a checking phase, as shown in Figure 3. 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

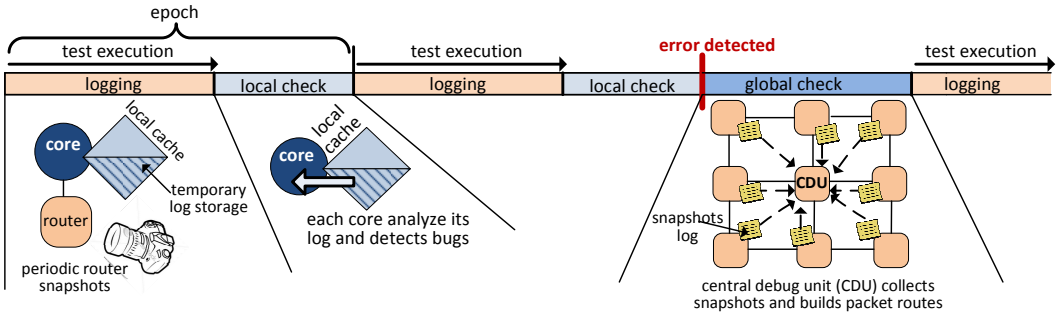

Fig. 3. **Execution flow of the NoC debug platform.** Execution is partitioned into epochs, each consisting of a logging phase and a local check phase. During the logging phase, snapshots of each router's contents are periodically taken and logged in a reserved portion of the local cache. During the local check phase, each local log is analyzed by a software algorithm running on its corresponding processor core. If an error is detected, the global check phase collects the snapshot logs from all caches and reconstructs the route of packets that were in-flight during the logging phase.

router. When the available space has been fully utilized, the logging phase terminates and network execution is halted. At that point, each core runs several software-based checks to analyze the snapshots that have been accumulating in its local cache, in order 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 transferred through the NoC and collected 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 a reconstruction of the paths followed by packets in flight.

4.1. Logging

4.1.1. Logging in the routers. During the logging phase, each router is configured to take snapshots of packets traversing it at the time, as illustrated in Figure 4. Snapshots are taken periodically at fixed time intervals and the physical clock of the router is used to track time. This interval is a user-defined value, which is set according to the characteristics of the NoC and the traffic density. To identify the packets to be logged, a router determines if header flits are 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 one of its input buffers and stores a copy of the header flit in the corresponding header buffer. Figure 4 illustrates this activity with an example, where, at time t40, a router has three packets, P7, P5 and P9 in input buffers IP1, IP2 and IP4, respectively. The header buffer associated with each input buffer stores a copy of the header flit of each packet.

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. As an example, the router architecture used in our experimental evaluation, Section 7, consisted of ten input virtual channel buffers (5 input ports and two virtual channels per port). With network packets consisting of 16 flits, every buffer can have at most one packet at a certain time and each header buffer needs at most one entry. A similar approach can be utilized to determine the header buffer size required when considering other router architectures and packet sizes. When a header flit is identified, the snapshot hardware can extract the packet's 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 consists of a sequential number

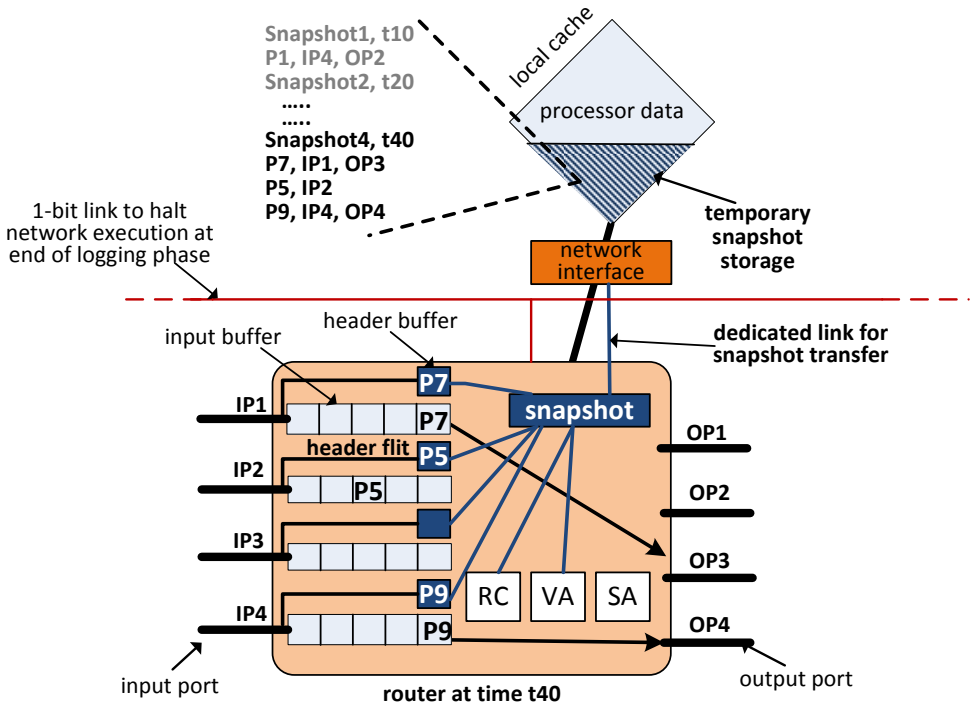

Fig. 4. **Logging.** Routers periodically take snapshots of packets traversing them. A header buffer is added for every input buffer to keep track of the header flits of packets stored in the input buffer. The snapshot hardware captures the header data as well as information from the route computation (RC) and virtual channel allocation (VA) modules of the router.

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 to an output port, whose value can be obtained from the route computation stage. Similarly, if the output virtual channel has been determined for that packet, the specific channel ID can be extracted from the virtual channel allocation phase. Besides providing debug data, logging the output port and output virtual channel allows us to determine the downstream router along that packet's path. Therefore, as illustrated in Figure 4, the snapshot captured at time t40 consists of packets P7, P5, and P9, along with their respective input ports, IP1,IP2 and IP4. Packets P7 and P9 have been allocated to an output port and thus their entries also include their respective output ports, OP3 and OP4.

In addition to packet-specific information, being able to trace these packets over time is important for debugging. Thus, a snapshot also stores the physical time at which it is taken. If the network uses multiple clock domains, physical times can be offset at different nodes. In this case, we rely on the notion of logical time implemented through Lamport clocks [Lamport 1978], where every router includes a counter to keep track of its logical clock, which is advanced every time a new packet is received. In this setup, packet headers also include a logical 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 it 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 includes the logical timestamp of the packet.

Note that, when the system operates with a single global clock and thus we can log the physical timestamp, we only need to store one timestamp for each snapshot, instead of one for each snapshot entry.

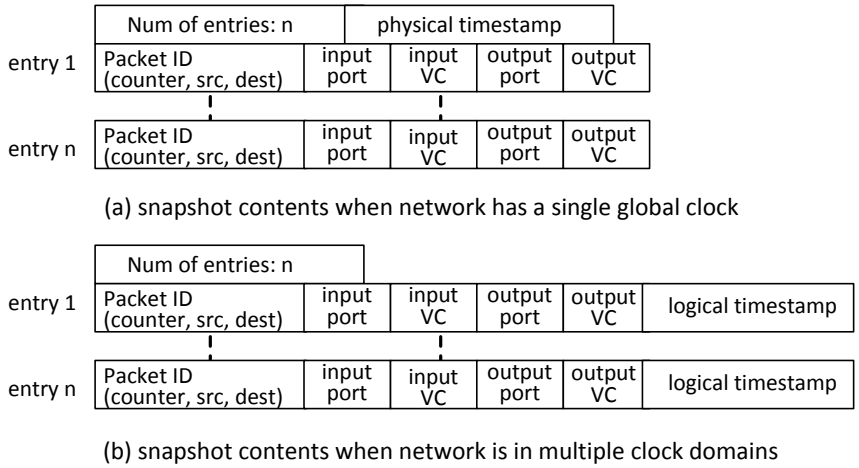

Fig. 5. **Router snapshot format.** A snapshot comprises data organized in several entries, one for each packet found in the router at the time of the snapshot. For each entry, we record a packet ID, a timestamp, and as much information as possible about the packet's incoming and outgoing ports and channels.

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 5 shows the information logged in each snapshot. In our experimental platform described in Section 7, 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. The width of that link determines the number of cycles needed to transfer the snapshot from the router. In our experimental setup, a 64-bit link can ensure that the snapshot is transferred within 10 cycles. As shown in Figure 4, the collected snapshot at time t40 is stored in the local cache, along with older snapshots. When available storage is full, the corresponding node transmits a flag through a 1-bit dedicated link, halting the network temporarily and initiating the local check phase.

4.1.2. Log Storage In Local Caches. When a snapshot is captured in the router, it is transferred through a dedicated link to the network interface and then to the local L2 cache of that node, as shown in Figure 4. Therefore, the local L2 cache, which is typically set-associative, designates one or more ways to be used solely for the storage of snapshot data. This effectively disables these set entries from the perspective of the processor and the cache allocation and replacement policies.

From the perspective of the snapshot data, even though the reserved cache lines belong to different sets, they can be collectively viewed as one big FIFO. Snapshots are written and read from each line in order of increasing cache index. Depending on the size of the snapshot relative to the cache line size, a snapshot could be padded to fill the line or it could span multiple cache lines. Since every snapshot stores the number of entries it contains, and given that the entry size is constant, determining the size of each snapshot is straightforward and it is needed for addressing the cache, reading

the snapshots, and flagging a full log. In order to write/read from the snapshot log, an address-generating unit generates the cache index to address the line in the next set. In addition, a running counter can keep track of the total size of snapshots logged so far. When this counter exceeds the user-sepecified log size, a local check is triggered. Similarly, snapshots are read from one line at a time. Having the number of snapshot entries and knowing the fixed snapshot entry size, allows proper parsing of the data and the snapshot boundaries.

In our experimental setup detailed in Section 7, the average size of snapshots for bitcomp traffic at varying injection rates ranges between 9 to 12 bytes. Similarly, the average snapshot size for the Parsec benchmarks is approximately 9 bytes. Considering, as an example, a 256KB L2 cache that is 4-way set associative with 32B lines, then we can configure one of those ways to be used to store only the incoming snapshots, which is equivalent to 64KB. However, in practice, it maybe useful to trigger a local check before all the physically available storage space has been used, so that the bug is detected closer to its ocurrence and debug data is more relevant. In our experiments, we set the maximum log size to be 30KB. Therefore, we rely on the running counter to keep track of the total size stored so far and to trigger a local check when it exceeds the preset limit. Finally, note that if snapshots are very large and span several lines, then it is possible to exhaust the entire reserved space before the maximum log size is reached. In this case, the local check can be triggered earlier or the execution can be repeated while reserving more ways for snapshot data.

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. If the design under validation is a system-on-chip that includes network nodes not connected to a core, then the local check algorithm only executes on the processor cores in the system. Nodes without an associated core would have to transmit their log data to the nearest core-node throughout the epoch. Alternatively, if the network bandwidth is a design bottleneck, the local checks would simply be performed on the nodes connected to a core, possibly missing errors that can only be detected from non-core nodes.

Figure 6 shows the pseudocode for the local check algorithm. It 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 or misroute bug. In the remaining of this section we discuss each of the possible erroneous situations that may arise. After all local checks have completed, if no error has been detected, 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 connected to a processor core. In-flight packets are dropped and the logs are sent to the CDU from each cache, one at a time. Transmitting only one log at a time greatly reduces the complexity of network operations during the transmission of the logs, boosting the likelihood of error-free transmission, since at this stage of the verification process (post-silicon), the network's most basic operations –such as transmitting correctly one singly packet– have been extensively validated.

4.2.1. 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 lo-

```
1. LocalCheck (snapshotLog){
2.  foreach packet in snapshotLog {

3.   foreach ntr in GetSnapshotEntries(packet) {
4.    time = PhysicalTimeEntry(ntr);
5.    next_time = PhysicalTimeEntry(ntr+1)
6.    if (next_time - time > snapshotInterval)
7.     FlagError(Livelock)}

8.   if (CountEntries(packet) > threshold)
9.     if packet in lastSnapshot(snapshotLog)
10.       FlagError(Deadlock)
11.     else FlagError(Starvation)

12.   src = GetSrc(packet)
13.   dest = GetDest(packet)
14.   if router !InPath(src, dest)
15.    FlagError(Misroute) } }
```

Fig. 6. **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.

cally 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 6 (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.

4.2.2. 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 first determines the number of consecutive snapshots in which each packet appears (Figure 6, line 9). Then, based on the snapshot rate, it deduces the number of cycles the packet has been waiting. A starvation error is flagged when the number of cycles exceeds a user-set threshold.

4.2.3. 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. However, 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 observed in the latest snapshot. Therefore, the local check algorithm checks if any of the packets in the snapshots satisfy both these conditions and, in that case, flags them as deadlocked (lines 10-11 in Figure 6). In the case that a coincidentally starved packet

happened to still be in the router at the time when the last snapshot was taken, it is possible to misclassify the starvation error as a deadlock.

4.2.4. Misroute. Under deterministic routing, a packet traveling between a source and destination pair should always go through the same route, based on the routing algorithm. In this scenario, a misroute occurs if a packet is routed to a node that is not on this deterministic 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 misrouting errors, 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 6). In the case of non-deterministic routing, a packet going from a given source to a destination node can travel through multiple valid routes. Therefore, the valid paths bit-vector can be replaced by a statistical distribution of acceptable paths for each source-destination pair. Packets following routes that are outliers, relative to the statistical distribution, can be flagged as potential misroutes.

4.3. Local Check Down-sampling

To reduce the time spent in each local check phase, we 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 fraction of the snapshot entries. This is accomplished by processing uniformly distributed burst intervals of the log, until the number of snapshot entries processed reaches the target sampling rate. This local check *sampling rate* is a user-defined value, which we evaluate in Section 7.1 and discuss in detail in Section 5, in the context of our proposed post-silicon methodology.

It is possible to decrease the total time spent in local checks by simply increasing the snapshot interval. A larger snapshot interval means that snapshots are taken less frequently and the available storage fills up slowly. Therefore, within a given execution time, there are fewer local check phases and the overall time spent in local checks is less. Note that in these cases, the time spent in each local check phase remains the same, since the local check phase is still initiated when the cache space is full. However,the advantage of using local check sampling, as opposed to increasing the snapshot interval, is that it reduces the amount of data that needs to be analyzed during each local check phase without compromising the granularity at which data is being logged. By looking at burst intervals of the log, local check sampling can continue to capture the fine-grained traffic behavior that is provided by the smaller snapshot intervals.

4.4. Global Check

The goal of the global check phase is to provide useful diagnostic information that can facilitate the debugging of an error detected in the local check phase. Figure 7 shows the pseudo-code of the global check algorithm. 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 by increasing order of snapshots' physical timestamps, when a global clock is present, or the logical timestamps when the network uses multiple clock domains (lines 4-5 in Figure 7). In the latter case, because the logical timestamp

```
1. GlobalCheck (combinedSnapshotLogs){
2.  foreach packet in combinedSnapshotLogs {
3.
4.  packet_entries = GetSnapshotEntries(packet)
5.  sort(packet_entries)
6.
7.   foreach entry  in packet_entries {
8.      router = entry->router;
9.      upstream_router= GetUpRouter(router, entry->input_port);
10.     downstream_router= GetDownRouter(router, entry->output_port);
11.     packet_path -> Add(upstream_router, router, downstream_router); }
12. }}
```

Fig. 7. **Global check algorithm.** In the global check phase, snapshot logs from all routers are collected at a central debug unit. Snapshot entries relating to a packet are grouped together and sorted by increasing timestamp. The packet's path is then reconstructed, while using the input and output ports to infer a portion of its path beyond the router in which the packet was observed.

of a packet is monotonically incremented in every hop, sorting according to increasing timestamp values allows the reconstruction of the path in the correct order. 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 (lines 7-11 in Figure 7. Indeed, we can 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 at which the snapshot was taken, exposing interactions that possibly triggered the error. For example, by examining a router's snapshot log, we can identify a subset of the packets that traversed it at the time surrounding the bug occurrence and deduce the router's internal state, such as the buffers that were in-use and the virtual channel and output port allocations at the time.

5. PLATFORM PARAMETERS

The proposed post-silicon platform includes several parameters to be tuned to provide high observability and debug information. First, the *snapshot interval* value determines the frequency at which traffic is sampled and information is logged. Therefore, lower snapshot intervals allow for a finer-grained detection and debugging. Similarly, having a higher *sampling rate* during the local check phase increases the probability of detecting erroneous traffic behavior, as more data is available for analysis. Our experimental evaluation in Section 7 validates the above expectations. In particular, we found that the detection of some errors such as misroute and starvation benefits from higher sampling rates and lower snapshot intervals. Whereas other errors, like deadlocks and livelocks, are almost always detected even with local check sampling as low as 20% and a snapshot interval of at least 50 cycles.

Based on these observations, we propose a methodological approach to deploy this platform in a post-silicon validation flow. In this flow, validation is divided into two phases. During the first phase, tests are executed with a high snapshot interval and a low sampling rate. This allows the detection of errors that have a widespread effect on the behavior of erroneous packets, such as deadlocks and livelocks. This is shown in the first block of Figure 8, where the platform parameters are tuned to an initial value and the regression tests are simulated. The post-silicon simulation, which is equipped with

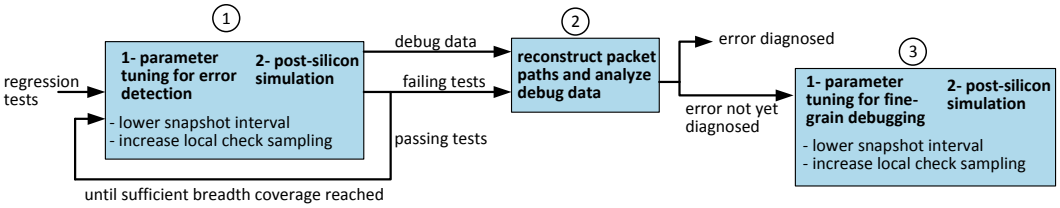

Fig. 8. **Post-silicon validation flow.** Tests are first executed with high snapshot values and low sampling rates on our post-silicon simulation platform. Failing tests are debugged using the generated debug data and the reconstructed packet paths. In case the error could not be diagnosed, the same test is repeated while varying the platform parameters to achieve finer-grained debugging. Passing tests are also repeated while iteratively sweeping the snapshot interval from high to low values and increasing the local check sampling rate.

our snapshot hardware and local check functionalities, generates debug data, which includes an overview of network traffic and partial path reconstruction of packets in-flight. This information can be used to diagnose and debug the failing tests. If the error could not be diagnosed, these tests can be repeated with lower snapshot interval values and higher sampling rates in order to achieve finer-grained debugging. This is depicted in block 3 in the flow diagram in Figure 8. In the second phase, passing tests can then be repeated, while iteratively lowering the snapshot interval and increasing the local check sampling rate. During every iteration of this process, a verification engineer would be increasing the platform's detection accuracy. Therefore, this second phase, is likely to expose bugs whose effects are more transient in nature and which are harder to capture, such as starvation and misrouting errors.

Moreover, the specific values selected for the platform parameters affect epoch length, the packets that are captured by snapshots, and the fraction of each log that is analyzed. Therefore, running the same test while sweeping these parameters can provide observability over different parts of the test's execution, which allows for better error detection. Similarly, failing tests can be repeated with different parameters or with finer-grained snapshots (lower snapshot intervals) in order to provide more debugging capabilities. As an example, one limitation of the local check algorithm is that it can miss a livelock bug, if the livelock cycle does not complete within the logging epoch or a starvation bug, particularly if the bug manifests very close to the end of an epoch. Repeating these tests with different snapshot intervals modifies the epoch length and allows us to overcome this limitation.

The last set of our NoC validation platform parameters are the *reserved cache space* and the *deadlock and starvation thresholds* used in the local checks. All these three parameters can be set upfront depending primarily on the nature of the network traffic and the NoC characteristics, such as topology and router architecture. For example, benchmarks with lower injection rates may require a smaller storage to trigger the local check frequently enough to detect errors. Finally, note that since we utilize a portion of the cache to store the logs, our framework can fail to expose some execution scenarios that would have occurred during runtime operation when the full cache is used. In such cases, it is possible for us to miss some bugs. However, this does not hinder the value of our solution. In fact, smaller available cache space causes conflict and capacity misses to be more pronounced. This would likely generate more traffic and better exercise the network, potentially allowing us to achieve the needed coverage.

6. RE-USING THE POST-SILICON FRAMEWORK FOR RUNTIME PERFORMANCE MONITORING

Hardware that is added to the system to faciliate post-silicon validation is fabricated and released along with the rest of the chip. This hardware is typically unused after the product is deployed and thus incurs an added area cost without providing any functionality or benefit from the prespective of the end-user. However, in our case, we propose to re-purpose our post-silicon sampling infrastructure for monitoring the network-on-chip's performance during runtime operation. Runtime performance monitoring can allow users to measure the quality of service delivered to applications running on the CMP and to determine whether the NoC is matching the performance requirements of the system. In addition, runtime performance data provide insights about the distribution of traffic and the underlying communication patterns, which in turn can help users analyze the effectiveness of application scheduling on the CMP processor cores.

For this purpose, our framwork is configured to log information about end-to-end latency of packets, instead of periodically taking snapshots of the network traffic. During this mode of operation, each packet's header flit is augmented with the time at which the packet is injected into the network. As explained in Section 4.1, additional space in the header flit is often available. Moreover, in contrast to post-silicon validation, during runtime, our framework no longer needs to keep track of packet IDs, since we only want to monitor performance values. Therefore, we can re-use some of the header reserved header space to store the injection time.

At periodic intervals, each router examines its header buffers to determine if a valid packet is present in the router. If such a packet exists and it is also destined to the node associated with that router, the packet's injection time is logged. As in the post-silicon approach, the snapshot is then transferred through the dedicated link and stored in the local L2 cache. If the design is a system-on-chip, nodes that are not associated with a processor core can log this information in additional storage and transmit them to core-nodes. If the network bandwidth is limited, runtime performance monitoring can be restricted to routers that are connected to a processing core. Note that this process requires no additional hardware modifications and simply re-uses the existing sampling infrastructure to take the snapshots and store them in the cache. Once in the cache, a software program calculates the end-to-end latencies as the difference between the time at which the snapshot was taken and the logged injection times. These latencies can then be accumulated into one value that represents the running average of end-to-end latencies measured by that router. In addition, it can also be useful to keep track of the maximum and minimum latency values observed at each router during execution. This runtime performance monitoring approach has minimal impact on the operation of the network and the CMP. First, the cache space allocated for the snapshots is minimal, since each snapshot only logs packets' injection times and the time at which the snapshot is taken. For example, considering a router architecture with ten input virtual channel buffers (5 ports, 2 virtual channels per port) and packet sizes of sixteen flits, a router can have at most ten header flits in its buffers at a given time. In this case, a snapshot consists of at most 10 injection time values that need to be stored in the cache. Moreover, once the average, the maximum, and minimum latency values are computed, the snapshot data can be deleted from the cache. This is in contrast to the post-silicon approach, where snapshots consisted of much more data (Figure 5) and had to be accumulated in the cache for analysis during the local check phase.

To evaluate this runtime performance monitoring approach, we utilized the experimental setup described in Section 7. Our network was an 8x8 mesh and was simulated with bitcomp random directed traffic. We first examined the accuracy of the sampled

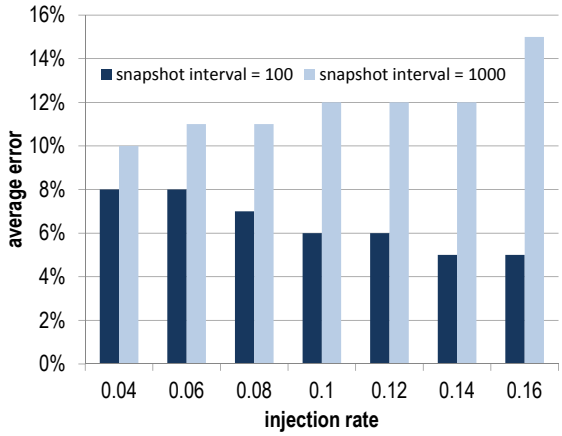

Fig. 9. **Accuracy of performance monitoring.** Error in end-to-end latency for bitcomp directed random traffic averaged for all routers in the network in an 8x8 mesh. For snapshot intervals of 100 cycles, where end-to-end latencies at every router are sampled every 100 cycles, the average error ranges between 5% to 8%. Increasing the snapshot interval, decreases the accuracy of the monitored end-to-end latencies relative to their actual values.

latency values relative to the actual end-to-end latency values at each router. Figure 9 shows the percentage error for the bitcomp traffic at varying injection rates and for two distinct snapshot intervals, 100 and 1,000 cycles. The error values are averaged over all routers in the network and all simulations (10 seeds per injection rate). When latencies are sampled every 100 cycles, the percentage error observed was found to be between 5% to 8%. Increasing the snapshot interval means that the latency values are sampled less frequently and the average error increases. However, even with interval of 1,000 cycles, the measured latency values were within 10% to 15% of the actual latencies.

Some other approaches to NoC performance monitoring have been proposed in the literature, as discussed in Section 3. The main advantage of our framework over these techniques is that it can be seamlessly configured for both post-silicon validation or runtime performance monitoring . A similar approach to ours was proposed in [van den Brand 2005] where the author utilizes the monitoring framework proposed by [Ciordas et al. 2004] for performance monitoring. Probes are added to NoC routers, with each probe consisting of a ”sniffer” that monitors traffic and an event generator. Monitored traffic is abstracted into events and transmitted over the NoC to a central unit or off-chip to extract performance measurements. We share with this solution the idea of router-level performance monitoring; however, we differ in two main regards. First, instead of continuously monitoring network traffic, we periodically sample performance measurements at each router. Second, the monitored information is logged in the local cache and analyzed by the processor core attached to each router. We therefore avoid utilizing the network to regularly transmit performance measurements, which introduces unnecessary performance slowdowns.

7. EXPERIMENTAL EVALUATION

To evaluate our debug platform, we modeled a CMP interconnect with Booksim, a cycle accurate C++ based network simulator [Dally and Towles 2003]. Our baseline system was considered 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 modified the simulator so that routers take periodic snapshots of packets traversing them and we im-

plemented the local check functions and the global check reconstruction method. We simulated two types of workloads: directed random traffic (uniform, bitcomp) and applications from the PARSEC benchmark suite [Bienia et al. 2008]. The space allocated for the snapshot logs was 30KB for the random traffic, less than 10% of a typical size L2 cache of 256KB.

Table I. Functional bugs injected into our network-on-chip baseline system.

bug name	bug description
deadlock	several packets permanently blocked in a deadlock cycle
livelock1	packet indefinitely circulates through a cyclic path of four routers
livelock2	packet is indefinitely passed from one router to another
starvation	packet is temporarily prevented from acquiring resources
misroute-1	packet is misrouted once along its path to destination
misroute-3	packet is misrouted three along its path to destination
misroute-9	packet is misrouted nine times along its path to destination

7.1. Error Detection

We first analyzed our platform's ability to detect functional errors. We modeled seven types of bugs in the baseline system, each representing variations of errors that would prevent the network from making correct forward progress. These include: A *deadlock bug*, where several packets are permanently blocked in a deadlock cycle. Two types of livelock bugs, *livelock1*, where a network packet indefinitely circulates through a cyclic path of four randomly chosen routers and *livelock2*, where a network packet is indefinitely passed from one router to another. A *starvation* bug, where a packet is temporarily prevented from acquiring the resources it needs to progress along its path. Three misroute bugs, *misroute-1, misroute-3, misroute-9*, where a packet is misrouted one, three or nine times, respectively, along its path from its source to destination. 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 II shows the detection rate when simulating bitcomp traffic over our seven bugs and two snapshot intervals (every 10 cycles and every 50 cycles). Note that, for the network considered in our experiments, it takes at least 18 cycles for a 16-flit packet to pass through a router. Once the header flit of that packet completes route computation and virtual channel allocation, the remaining fifteen flits bypass these two stages and follow the same route. We also 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 (less than 52%). 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 over time. On the other hand, misroute and starvation errors are transient and the affected packets can never be observed. In addition, this effect is more pronounced when the snapshot interval is increased to 50 cycles, because snapshots are now taken even less frequently. Note that for misrouting bugs, as the number of misroutes increase the detection rate increases as well. This is because the error manifestation becomes more pronounced and the probability of capturing the packet at a time when it is traversing a wrong path increases. Moreover, if sampling is activated

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

	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%
livelock1	100%	100%	100%	100%	100%	100%
livelock2	100%	100%	100%	100%	100%	100%
starvation	24%	7%	2%	0%	0%	0%
misroute-3	36%	17%	4%	6%	6%	3%
misroute-9	52%	29%	9%	4%	3%	3%

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.

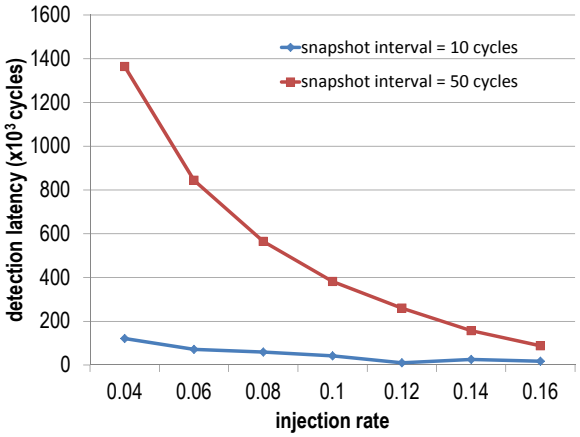

Fig. 10. **Time to detect errors with varying snapshot intervals and injection rates.** Higher snapshot interval values increase the detection latency as log space takes longer to fill up and trigger the local check phase.

7.2. Error Detection Time

In our post-silicon platform a bug is first detected during the local check phase, which is initiated when the reserved cache space is full. Therefore, the detection latency is directly related to the length of the logging phase, and the chosen snapshot interval is one of the main factors that affect it.

Figure 10 plots the bug detection latency for bitcomp traffic at varying injection rates and two snapshot intervals of 50 and 100 cycles. The injection rate was swept from 0.04 flits/node/cycle (injection rate near zero-load latency) to 0.16 flits/node/cycle (injection rate when network is near saturation). The results are averaged over all bugs, with each bug injected once during each simulation at a fixed time of 100,000 cycles. Note also that each run was set to execute for 1M cycles. This experiment highlights the

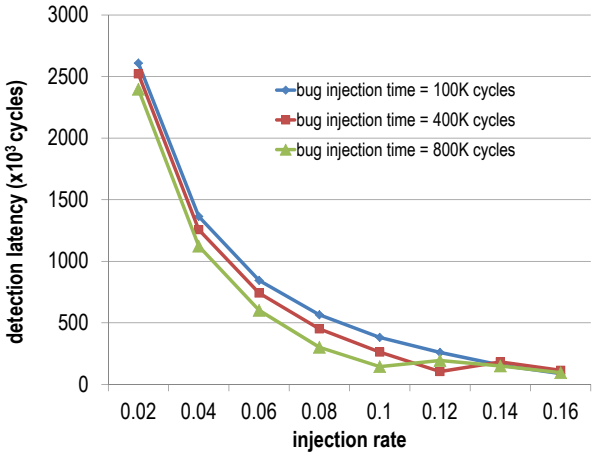

Fig. 11. **Time to detect errors with varying bug injection times.** Larger bug injection times lower the system's detection latency. Snapshot logs are closer to being full deeper into the simulation, which triggers local checks soon after bug injection. At higher injection rates, there is a less pronounced variation in detection latency with varying bug injection times.

impact of congestion and snapshot interval on detection latency. For a snapshot interval of 10 cycles, the time between bug injection and detection varied between 17,000 cycles (at high injection rate) to 12,000 cycles (at low injection rate). This is due to the fact that at higher injection rates, the network is more congested and more packets are logged at every snapshot. Therefore, the logs fill up faster and the local check phase is triggered much earlier allowing the system to detect the bug soon after it happened. On the other hand, when traffic congestion is low, the logging phase lasts longer, as it takes more time for the designated snapshot logs to fill up. Consequently, the error detection latency increases.

We also studied the impact of bug injection time on detection latency. Figure 11 highlights the results obtained for bitcomp traffic at a snapshot interval of 50 cycles. For low injection rates, as the bug injection time increases, the time it takes the system to detect the bug decreases. This is due to the fact that at low injection, snapshot logs fill up infrequently and much later during the execution. Therefore, a larger bug injection time means that the bug is injected closer to when a local check is triggered. As the injection rate increases, the effect of bug injection time is no longer pronounced. In those cases, the high traffic creates smaller epochs and very frequent local checks. Therefore, bugs are detected soon after they happen, irrespective of their injection times.

7.3. Error Diagnosis

We also evaluated the quality of information that can be obtained from the snapshots when they are aggregated during the global check phase. Table III shows 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 with respect to the total number of packets injected in the simulation. Second, we looked at the path reconstruction rate, which is the average fraction of each route that 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 error manifested).

Table III. **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.

		snapshot rate 10 cycles		snapshot rate 50 cycles	
low injection	%packets observed	54%		20%	
	avg. path reconstruction	53%		30%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	43%	misroute-1	0%
		deadlock	63%	deadlock	54%
		livelock1	74%	livelock1	66%
		starvation	36%	starvation	0%
		livelock2	29%	livelock2	6%
		misroute-3	56%	misroute-3	0%
		misroute-9	58%	misroute-9	0%
medium injection	% of packets observed	67%		28%	
	avg. path reconstruction	52%		31%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	55%	misroute-1	0%
		deadlock	67%	deadlock	49%
		livelock1	57%	livelock1	48%
		starvation	47%	starvation	0%
		livelock2	37%	livelock2	9%
		misroute-3	59%	misroute-3	22%
		misroute-9	64%	misroute-9	17%
high injection	%packets observed	85%		50%	
	avg. path reconstruction	58%		35%	
	avg. path reconstruction rate of faulty packets per bug type	misroute-1	60%	misroute-1	0%
		deadlock	77%	deadlock	54%
		livelock1	73%	livelock1	56%
		starvation	0%	starvation	0%
		livelock2	47%	livelock2	18%
		misroute-3	40%	misroute-3	23%
		misroute-9	67%	misroute-9	11%

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 rate of the erroneous packets, we notice that when the bug is detected, we can reconstruct on average 36%- 60% of its path. In some cases, for example the starvation bug under high traffic injection, the path reconstruction of the faulty packet is reported as 0%. This is because the faulty packet was not captured in any snapshots and 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 traffic 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 rates, a larger snapshot value would be sufficient to attain the same result, because of the higher congestion in the network.

7.4. Performance Evaluation

Our NoC debug platform periodically stops network execution to locally check the collected snapshot logs. The time spent in the local check phase is therefore the main source of performance overhead. To evaluate this overhead, we implemented and integrated our snapshot and local check algorithms with the Booksim simulator. Note that, since the local checks are intended to run in software on the CMP's cores, our implementation of these algorithms could not be cycle-accurate. Therefore, we instrumented the local check algorithms to utilize the x86 timestamp counters and report the execution time in cycles while running on a 2.4GHz Core2 Quad machine. We then added the execution time (in cycles) of the local checks occurring in each run to the benchmark execution time (in cycles) that was obtained from the cycle-accurate Booksim simulator. We compared these results to the benchmark execution time on the baseline system, which does not have any of our post-silicon debug functionalities. For our experiments, we simulated both PARSEC and random traffic benchmarks. We also assumed that 30KB of the local L2 cache were reserved for the snapshot log, when simulating random traffic workloads, and 10KB when simulating the PARSEC benchmarks. Note that the snapshot storage is approximately 10% or less of a typical L2 cache size of 256KB. In addition, the lower injection rate of the PARSEC benchmarks forced us to choose a lower storage size so that the local log could fill up and trigger a local check at least once during the benchmark's execution.

We first examined the performance **impact of varying snapshot interval values**. Figure 12a) shows the the execution time of the different workloads normalized to their respective execution times on the baseline system (without any of our NoC debug functionalities). In these experiments, the local check sampling rate was set to 20%. The normalized execution time only takes into account the overhead of the periodic local checks triggered during each run, as this is the main source of overhead in our framework. For random traffic, we varied the injection rate from low injection (when the network is close to zero-load latency) to high injection (before network saturation). With larger snapshot intervals, local checks are invoked less frequently and the overall benchmark execution times are smaller. This can be observed in Figure 12a), where larger snapshot intervals result in smaller values for normalized execution relative to the baseline system. For example, when the snapshot interval is set to 10 cycles, the execution time for bitcomp traffic ranges from 2x to 32x slower than a baseline system that does not implement our solution. When increasing the snapshot intervals to 50 and a 100 cycles, the execution time improves by a factor of 5 and 10, respectively. Note that despite this slowdown, the speed at which benchmarks are executed is still relatively high, especially when compared to the speeds of simulations during pre-silicon verification, which are typically in the order of 10-100Hz. However, as explained in Section 7.3, 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. PARSEC workloads experience on average 1-2x slowdown when executing on our framework with a snapshot interval of 10 cycles. This overhead consists of the additional time spent on locally checking the logs whenever they are

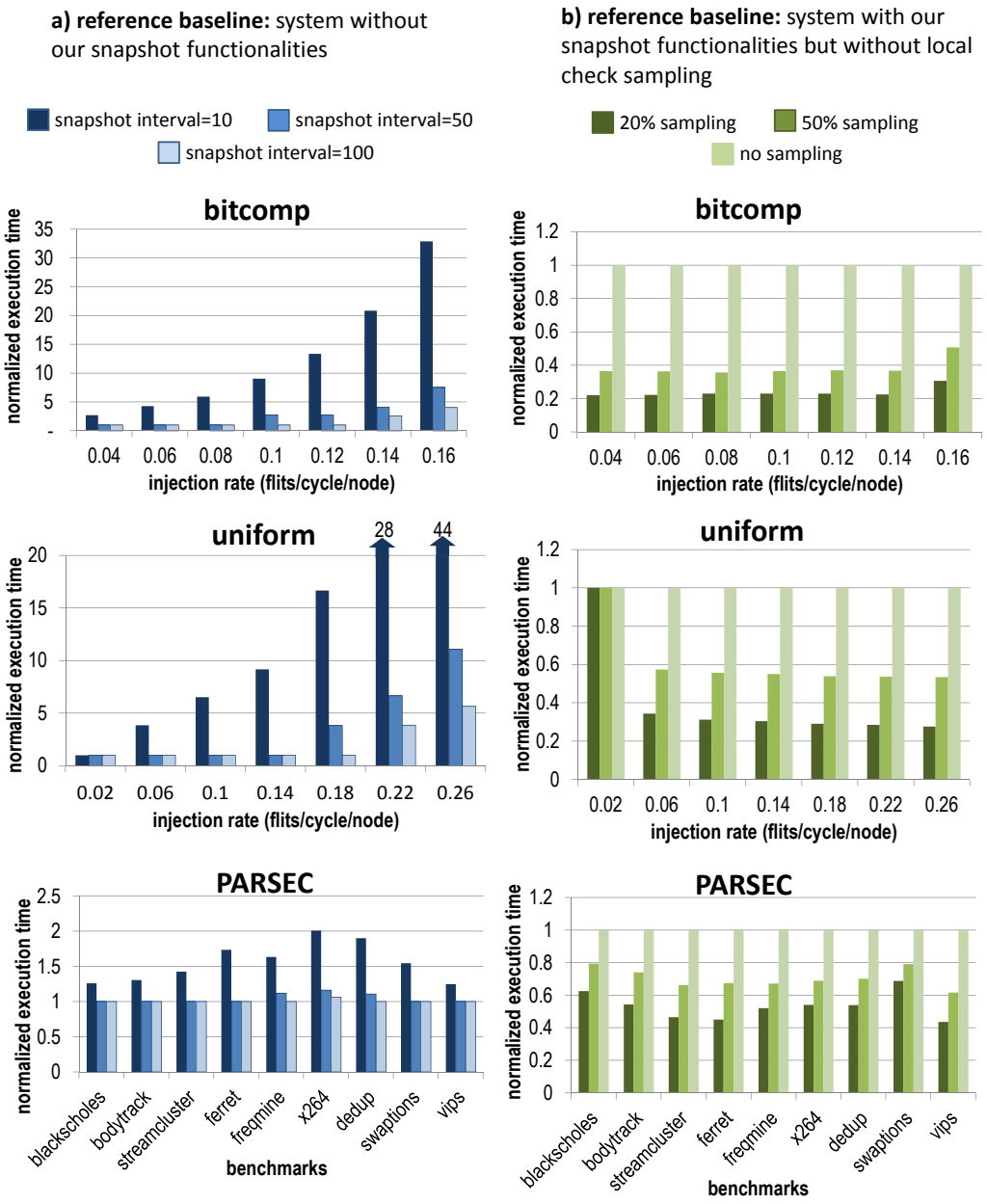

Fig. 12. **Normalized execution time for PARSEC and random traffic workloads with varying snapshot intervals and local check sampling rates.**

full. However, note that at low injection rates, increasing the snapshot interval might not be feasible, as was the case of the PARSEC benchmarks that would never trigger a local check (normalized execution time is 1)

We also evaluated the **effect of varying the local check sampling rate** on performance, in Figure 12b). In these experiments, we are using a snapshot interval of 10

cycles while varying the local check sampling rate between 20% and 50%. The results are normalized to a system without any local check sampling. As expected, sampling the logs during the local check phase speeds up the process. For example, at a sampling rate of 50%, where only half of each snapshot log is analyzed, the execution time is twice as fast, on average, for the random traffic. An even lower sampling rate of 20% improves the execution time to at least a factor of 3. A similar trend is observed for the PARSEC benchmarks.

Therefore the snapshot interval and the local check sampling rate constitute a tradeoff between the performance overhead of our post-silicon validation platform and its ability to detect and diagnose errors. Our post-silicon validation flow described in Section 5 tunes these parameters based on the goals of the simulation. During the first phase of the validation process, large snapshot interval values and lower sampling rates allow the system to run at higher performance. When failing tests are repeated for finer-grain debugging and when passing tests are iteratively repeated with different parameters, the performance of the system decreases to attain better error detection and debug.

7.5. Area

We also evaluated the area overhead of the router modifications that are needed to capture the local snapshots. The hardware implementation is described in Section 4.1 and illustrated in Figure 4. The hardware additions primarily consist of the header buffers, one buffer associated with each input buffer, as well as control logic to identify valid packets within the router and extract their corresponding state from the various router stages. Moreover, we also included a small storage to temporarily store the captured snapshot while it is being transferred to the local cache. We synthesized the modified baseline router with the Artisan 45nm target library. We found that the baseline router area was $0.075mm^2$ and the area overhead of our additions was $0.0067mm^2$, which corresponds to 5,296 NAND2-equivalent gates and constitutes 9% of the total router area.

8. 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. We incorporated our solution in a complete post-silicon validation flow that tunes the different system parameters and guides the validation process. Our platform targets functional errors that prevent the network from making correct forward progress, such as deadlock, livelock and starvation errors. This is accomplished by instrumenting routers to 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. In addition, we proposed a method to utilize the post-silicon platform for runtime performance monitoring of end-to-end latencies in networks-on-chip. Our experimental results show that during post-silicon validation, our debug platform can effectively collect information critical to the detection and diagnosis of functional errors, and that, during runtime performance monitoring, it can estimate end-to-end latencies at network routers with low error.

Acknowledgments. This work was supported by STARnet, a Semiconductor Research Corportation program sponsored by MARCO and DARPA, and NSF grant #1217764.

REFERENCES

ABRAMOVICI, M. 2008. In-system silicon validation and debug. *IEEE Design & Test of Computers 25,* 3, 216–223.

ABRAMOVICI, M., BRADLEY, P., DWARAKANATH, K., LEVIN, P., MEMMI, G., AND MILLER, D. 2006. A reconfigurable design-for-debug infrastructure for socs. In *Proceedings of the 43rd annual Design Automation Conference*. DAC '06.

AL FARUQUE, M., WEISS, G., AND HENKEL, J. 2006. Bounded arbitration algorithm for qos-supported on-chip communication. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*. 76–81.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*.

CHATTERJEE, D., MCCARTER, C., AND BERTACCO, V. 2011. Simulation-based signal selection for state restoration in silicon debug. In *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '11.

CIORDAS, C., BASTEN, T., RADULESCU, A., GOOSSENS, K., AND MEERBERGEN, J. 2004. An event-based network-on-chip monitoring service. In *High Level Design Validation and Test Workshop*. HLDVT'04.

CIORDAS, C., GOOSSENS, K., BASTEN, T., RADULESCU, A., AND BOON, A. 2006. Transaction monitoring in networks on chip: the on-chip run-time perspective. In *Proc. IES*.

DALLY, W. AND TOWLES, B. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

DAS, R., EACHEMPATI, S., MISHRA, A., NARAYANAN, V., AND DAS, C. 2009. Design and evaluation of a hierarchical on-chip interconnect for next-generation cmps. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*. 175–186.

DEORIO, A., BAUSERMAN, A., AND BERTACCO, V. 2008. Post-silicon verification for cache coherence. In *Proceedings of International Conference on Computer Design*. ICCD'08.

DEORIO, A., WAGNER, I., AND BERTACCO, V. 2009. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. In *Proceedings of International Symposium on High Performance Computing Architecture*. HPCA'09.

IEEESTD1149.1.1990. IEEE standard test accesss port and boundary scan architecture. *IEEE Std. 1149.1-1990*.

JAFRI, S. M. A. H., GUANG, L., JANTSCH, A., PAUL, K., HEMANI, A., AND TENHUNEN, H. 2012. Self-adaptive noc power management with dual-level agents - architecture and implementation. In *PECCS*. 450–458.

KIM, G., KIM, J., AND YOO, S. 2011. Flexibuffer: reducing leakage power in on-chip network routers. In *Proceedings of the 48th Design Automation Conference*. DAC '11. ACM, New York, NY, USA, 936–941.

KO, H. F. AND NICOLICI, N. 2008. Automated trace signals identification and state restoration for improving observability in post-silicon validation. In *Proceedings of the conference on Design, automation and test in Europe*. DATE '08.

KO, H. F. AND NICOLICI, N. 2010. Automated trace signals selection using theRTL descriptions. In *Proceedings of International Test Conference*. ITC'10.

KRISHNA, T., PEH, L.-S., BECKMANN, B. M., AND REINHARDT, S. K. 2011. Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44 '11. ACM, New York, NY, USA, 71–82.

LAI, C.-H., YANG, F.-C., KAO, C.-F., AND HUANG, I.-J. 2009. A trace-capable instruction cache for cost efficient real-time program trace compression in soc. In *Proceedings of the 46th Annual Design Automation Conference*. DAC '09.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*.

LI, Z., ZHU, C., SHANG, L., DICK, R., AND SUN, Y. 2008. Transaction-aware network-on-chip resource reservation. *IEEE Comput. Archit. Lett. 7,* 2, 53–56.

LIU, X. AND XU, Q. 2009. Trace signal selection for visibility enhancement in post-silicon validation. In *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09.

LV, Z., CHEN, H., CHEN, F., AND LV, Y. 2011. Fast verification of memory consistency for chip multiprocessor. In *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*. CIS '11.

MISHRA, A. K., SRIKANTAIAH, S., KANDEMIR, M., AND DAS, C. R. 2010. Cpm in cmps: Coordinated power management in chip-multiprocessors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. IEEE Computer Society, Washington, DC, USA, 1–12.

PANDA, P. R., BALAKRISHNAN, M., AND VISHNOI, A. 2011. Compressing cache state for postsilicon processor debug. *IEEE Transactions on Computers 60,* 4, 484–497.

PANDA, P. R., VISHNOI, A., AND BALAKRISHNAN, M. 2010. Enhancing post-silicon processor debug with incremental cache state dumping. VLSI-SoC'10.

PARIKH, R. AND BERTACCO, V. 2011. Formally enhanced runtime verification to ensure NoC functional correctness. In *Proceedings of the International Symposium on Microarchitecture*. MICRO'11.

PARK, S.-B., BRACY, A., WANG, H., AND MITRA, S. 2010. Blog: post-silicon bug localization in processors using bug localization graphs. In *Proceedings of the 47th Design Automation Conference*. DAC '10.

PARK, S.-B., HONG, T., AND MITRA, S. 2009. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *Transactions on Computer-Aided Design of Integrated Circuit Systems 28,* 10, 1545–1558.

ROTITHOR, H. 2000. Postsilicon validation methodology for microprocessors. *IEEE Desing Test 17,* 4, 77–88.

STUIJK, S., BASTEN, T., GEILEN, M., GHAMARIAN, A., AND THEELEN, B. 2006. Resource-efficient routing and scheduling of time-constrained network-on-chip communication. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*. 45–52.

TANG, S. AND XU, Q. 2007. A multi-core debug platform for NoC-based systems. In *Proceedings of the conference on Design, automation and test in Europe*. DATE'07.

VAN DEN BRAND, J. 2005. Runtime networks-on-chip performance monitoring. M.S. thesis, Technische Universiteit Eindhoven.

VERMEULEN, B. AND GOOSSENS, K. 2009. A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs. In *Proc. VLSI-DAT*.

VERMEULEN, B., OOSTDIJK, S., AND BOUWMAN, F. 2001. Test and debug strategy of the pnx8525 nexperia-atm digital video platform system chip. In *Proceedings of the IEEE International Test Conference 2001*. ITC'01.

VISHNOI, A., PANDA, P., AND BALAKRISHNAN, M. 2009. Cache aware compression for processor debug support. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*

WAGNER, I. AND BERTACCO, V. 2008. Reversi: Post-silicon validation system for modern microprocessors. In *Proceedings of International Conference on Computer Design*. ICCD'08.

YANG, J.-S. AND TOUBA, N. A. 2009. Automated selection of signals to observe for efficient silicon debug. In *Proceedings of VLSI Test Symposium*. VTS'09. 79–84.

YI, H., PARK, S., AND KUNDU, S. 2008. A design-for-debug (DfD) for NoC-based SoC debugging via NoC. In *Proceedings of Asian Test Symposium*. ATS'08.

YI, H., PARK, S., AND KUNDU, S. 2010. On-chip support for NoC-based SoC debugging. *IEE Trans. on Circuits and Systems 57,* 7.