Formally Enhanced Verification at Runtime to Ensure NoC Functional Correctness

Ritesh Parikh, Rawan Abdel Khalek, Valeria Bertacco

The Department of Computer Science and Engineering, University of Michigan
{parikh,rawanak,valeria}@umich.edu

ABSTRACT

As silicon technology scales, modern processor and embedded systems are rapidly shifting towards complex chip multi-processor (CMP) and system-on-chip (SoC) designs. As a side-effect of complexity of these designs, ensuring their correctness has become increasingly problematic. Within these domains, Network-on-Chips (NoCs) are a de-facto choice to implement on-chip interconnect; their design is quickly becoming extremely complex in order to keep up with communication performance demands. At the same time, functional design errors escaping into a system's interconnect may have a detrimental impact on the whole system.

In this work, we propose ForEVeR, a solution that complements the use of formal methods and runtime verification to ensure functional correctness in NoCs. Formal verification, due to its scalability limitations, is used to verify smaller modules, such as individual router components. To complement this, we propose a network-level detection and recovery solution to deliver correctness guarantees for the complete network. ForEVeR augments the baseline NoC with a lightweight checker network that alerts destination nodes of incoming packets ahead of time, and which is used to detect errors in the communication. If a bug is detected, a recovery mechanism delivers the in-flight data safely to the intended destination via the checker network. ForEVeR's experimental evaluation shows that it can recover from NoC design errors at only 5.8% area cost for an 8x8 mesh interconnect, over a time interval ranging from 0.5K to 30K cycles per recovery event. Additionally, it incurs no performance overhead in the absence of errors.

1. INTRODUCTION

Continuous technology scaling has enabled computer architects to design larger and more complex CMPs and SoCs for improved performance. As a result, the corresponding increase in communication demands has rapidly sidelined traditional interconnects such as buses and it has led to an industry wide shift towards more scalable and efficient solutions such as NoCs. In NoCs, data is transmitted in units called 'packets', that can further be divided into smaller blocks called 'flits' depending on the implementation. Packets are injected into the network via a dedicated network interface (NI) and they are transmitted to their destinations through various routers according to some routing protocol. To keep up with the computation capability of cores/IPs, NoCs are becoming increasingly complex by employing various techniques to efficiently manage load. Networks often implement irregular topologies and a wide variety of complex protocols, such as adaptive routing. Moreover, routers come with various advanced features such as virtual channels, speculation, complex allocation schemes, lookahead routing and various flow control and congestion management schemes.

With all these performance optimizations it is challenging to ensure correct functionality under all circumstances. Moreover, the complex nature of interactions within a network, require validation of the complete interconnection system as a whole. Pre-silicon verification efforts in this space use a combination of simulation based verification and formal methods. Simulation based testing will always be incomplete as we cannot exhaustively test the countless different interactions within a network. On the other hand, formal methods such as model checking, though complete and effective for verifying small portions of the designs, do not scale to provide end-to-end system level guarantees. Thus, design errors could escape verification and manifest at runtime, causing corruption of network/processor state, loss of data, failing of software application or of the entire system. Thus, we need an appropriate runtime solution that ensures correct execution even in face of design bugs.

To counter the shortcomings of design time verification, a simplistic approach would be to naively safeguard against all possible failures scenarios at runtime. Such an approach would be expensive in area cost and the corresponding design modifications may be intrusive, further complicating the design. ForEVeR's approach is based on the insight that although formal methods do not scale to the complexity of entire NoC, yet they are complete and can ensure component level correctness, which in turn could greatly simplify the otherwise agnostic runtime detection and recovery. Thus, ForEVeR is designed to cover only those features of the design that cannot be formally proven to work correctly. In this manner, the real-estate and design time dedicated to runtime verification directly benefits from the designer's ability to formally verify.

1.1 Contributions

To the best of our knowledge ForEVeR is the first work to provide correctness guarantees in NoCs via complementary use of formal verification and runtime validation. Formal methods, specifically model checking are employed to verify the basic router functionality, i.e. to prove that packet integrity is maintained within a router. On the other hand, runtime validation ensures forward progress within the network. Together, formal verification and runtime validation guarantee that all data is eventually delivered to the respective destinations without being dropped or corrupted. To this end, ForEVeR augments the baseline NoC with a lightweight checker network, that is guaranteed to deliver messages correctly. For each data packet sent over the primary NoC, an advance notification is transmitted over the checker network to report a future packet delivery. Each destination maintains a count of expected packets and initiates recovery on 'buggy' behavior of counter values. During recovery, all packets in-flight in the primary NoC at the time of bug detection are reliably delivered to intended destinations via the checker network.

The ability to formally ensure packet's integrity within a router might vary depending on the complexity of the router logic. To cover scenarios where local router functionality cannot be completely verified using formal methods, we propose router-level runtime checkers that monitor each router for 'correct' behavior of the un-verified parts. If these checkers flag an error, our network level recovery is triggered and the router pipeline is forced into a degraded performance mode of operation. In degraded mode, a router disables most of it's advanced features to a threadbare version with enough functionality to support the network-level recovery process. Correctness of the system in this simplistic mode of operation can be formally verified, guaranteeing complete recovery past the occurrence of the bug by running in degraded mode.

Our solution is independent of NoC topology, router architecture and routing protocol. With proper synergy between formal and runtime validation, ForEVeR can detect and recover from a wide variety of interconnect design errors, hence ensuring forward progress with no data corruptions. Moreover, ForEVeR adds simple, verifiable, mostly decoupled hardware to the baseline interconnect; runtime monitors that check the un-verified router functionalities for erroneous behavior and a lightweight checker network to deliver advance notifications to destination nodes. Finally, ForEVeR uses the same checker network to enable a unified recovery mechanism to overcome bugs in the NoC system. ForEVeR comes at an area cost of 5.8% for 8x8 mesh interconnect, incurring a minimal performance impact on NoC operation, only when an error manifests.

2. RELATED WORK

Recently Bayazik and Malik [4] suggested a generic hybrid verification methodology that proposes the use of hardware checkers to avoid state explosion in model checking by validating assumptions and abstractions at runtime. ForEVeR, on the other hand, uses formal methods and runtime verfication in a hybrid methodology to provide separate correctness goals for NoCs. Moreover, [4] is a generic methodology for verfication of complex properties that cannot be directly applied to NoC correctness. Other works such as [6] look at formally verifying an abstracted model of NoC, and thus could not guarantee correctness of the actual implementation.

Ensuring the runtime correctness of NoCs has been the subject of several previous research, focusing on a variety of aspects. Several works [17, 2, 12, 11] target deadlock, prominent in adaptive routing, while others target a wider variety of errors through general end-to-end detection and recovery techniques [14]. Traditionally the deadlock problem in NoCs is overcome by deadlock avoidance, where certain routes are forbidden, sometimes hurting performance. Other works, such as DISHA [2], try to detect and recover from deadlocks. DISHA detects deadlocks by timeouts and recovers by progressively routing blocked packets through a deadlock-free dedicated link. Other works [12, 11], propose more sophisticated detection mechanisms based on monitoring activity at router channels. In contrast, ForEVeR safeguards against all kind of functional bugs, including deadlock and livelock. In addition, ForEVeR is an end-to-end solution using hardware mostly decoupled from the primary NoC, thus making minimal changes to the NoC design and ensuring that added hardware does not further complicate the design. Other end-to-end approaches for NoC runtime correctness have been surveyed by [14]. The most common error recovery scheme for NoCs is the acknowledgement based retransmission technique of [14], where error detection code is sent along with the data packets, to check for data corruption at the receiver. An acknowledgement is sent back in case of successful transfer, otherwise the sender times out and re-transmits the packets from its backup buffers. Apart from large backup buffers and performance degradation due to additional acknowledgement packets, this approach suffers from the obvious disadvantage of not being able to overcome errors such as deadlock and livelock. Moreover, since it uses the same untrusted network for re-transmission, it cannot guarantee packet delivery.

We are not aware of any runtime solutions that deal with all kinds of design errors in NoCs. However, there are few such works for processors, [3, 13, 20]. In general, these solutions add checker hardware to verify the operation of untrusted components, switching to a verifiable degraded mode on error detection. ForEVeR is similar to such solutions, in the sense that it uses a combination of runtime monitors and a functionally correct checker network to verify the operation of the complex primary network, switching to a rudimentary mode of operation during recovery.

Finally, ForEVeR detection mechanism relies on use of router-level runtime monitors, when formal methods fail to ensure router correctness. The idea of using runtime checkers has been proposed for various purposes. [7] champions the use of runtime monitors for post-silicon debug and in-field diagnosis, as a generic design methodology. In contrast, ForEVeR employs definite hardware monitors, specifically designed for checking router's runtime correctness and in addition provides support for error recovery. Similar hardware checkers were used in [16] to monitor router's control logic for corruption due to soft faults. Our checkers, apart from being simpler, are designed specifically to supervise routers for correct functioning, being as less intrusive as possible at the same time.

3. METHODOLOGY

The success of complementary verification depends on intelligent division of correctness goals between the two complete verification techniques, namely formal and runtime verification. Formal verification is a theoritically complete, guaranteeing correctness under all execution scenarios. However, model checking, which has been the work-horse for formal verification, suffers from the state explosion problem that prevents it from proving end-to-end correctness properties of large systems. Additionally, formal verification solutions are not keeping up with the design trends and as a result design complexity is growing faster than the ability to verify.

In contrast, runtime validation approaches are not affected by state explosion as they only monitor the execution scenarios carried out during the actual design run. Yet they provide for complete verification as paths not taken at runtime are of no concern to design correctness. Another advantage over formal techniques is that they can recover from the errors dynamically, making runtime validation an effective method to heuristically detect system-level violations and recover from them in-field. However, standalone runtime solutions tend to be area intensive. Moreover, these agnostic runtime solutions are often intrusive, further complicating the already complex designs.

Based on these observations, we employ formal methods, specifically model checking, to verify the basic router functionality, which includes proving that packet integrity is maintained within a router. In other words we verify that flits are not dropped and they follow the header flit in a wormhole fashion. However, formal methods might not be sufficient for proving correctness of complex routers and could need additional runtime monitoring hardware along with some reconfiguration logic that enables the degraded mode of routers during recovery. As for design errors that inhibit forward progress of the system, such as starvation and deadlock, they are detected by a network level runtime detection mechanism, followed by a recovery step, which reliably delivers all the packets in error to their respective destination nodes. Figure 1 shows the high level overview of ForEVeR, where partially verified routers are connected together to form a NoC system that is augmented with detection and recovery logic to provide runtime correctness. Runtime checkers and recovery logic are used at router level to protect complex router components against design flaws. The primary NoC is augmented with a lightweight checker network that is used to deliver advance notifications to the counting logic at destination nodes. During recovery, the checker network is used to reliably transmit in-flight primary network packets to their respective destinations.

4. ROUTER CORRECTNESS

In the context of our scheme, a correctly functioning router should ensure that a packet's integrity is maintained within or across routers.

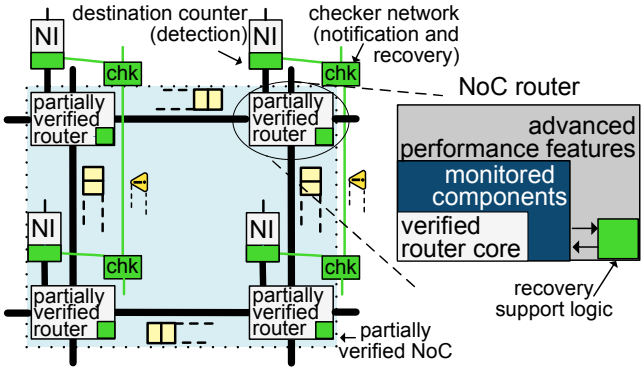

Figure 1: High-level overview of ForEVeR. A combination of router-level runtime monitoring and network-level detection and recovery scheme, along with component formal verification, ensures correct NoC operation.

If a router guarantees that no flits are dropped and all body flits follow the head flit in a wormhole, then we consider it to be functioning correctly. Our definition of router's correctness is motivated by the fact that even with no guarantees of forward progress, if each router in the network ensures to hold these properties, all data in-flight in the network is maintained in a coherent state and correct network state can be restored without any need of data duplication.

In this section we describe our approach to ensure router correctness using either formal or runtime verification. Without loss of generality we discuss our ideas for a fairly complex 3-stage pipelined router that is input-queued and that uses virtual channel (VC) flow control, lookahead routing and switch speculation. A detailed schematic of this router is shown in Figure 2. A router essentially ties together its datapath components, such as input buffer, channel and crossbar, with a control plane that consists of input VC control (IVC), route computation unit (RC), VC allocator (VA), switch allocator (SA), output VC control (OVC) and flow control manager. The control plane manages the error free flow of data from input channels to the output channels via input buffers and crossbar respectively. Since the data path components are fairly simple and can be easily verified, we specifically focus our verification effort on controlling logic. Moreover, it is well known that the most complex verification tasks arise from the interaction of concurrent components. In the framework of a router, the interactions between the concurrently operating VCs are handled by RC, VA and SA units. These units utilize information provided by the flow control mechanism, which is used to transmit buffer state information among neighboring routers. Other control units such as IVC and OVC operate mostly on a standalone basis, with information provided by the RC, VA and SA units, and hence can be formally verified using existing formal tools.

Based on the above observations, first a full formal proof of router correctness is attempted. If due to the complexity of the logic involved, formal methods fail to provide correctness guarantees, only parts of the router that can be easily handled by existing formal tools are verified. Then runtime hardware checker are used to protect the vital router components that handle the interactions among concurrent units, which keeps the area cost low. In addition, provisions have to be made to prevent the NoC from entering an unrecoverable state; for example this might happen when a flit is dropped or corrupted before the monitoring hardware flags an error. In either case, operation of the router during recovery has to be formally verified.

4.1 Formal Verification

The verification process can be efficiently divided into two sub-

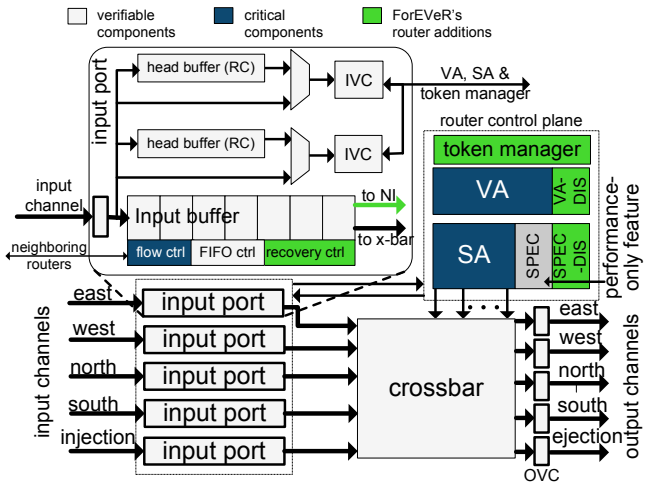

Figure 2: Router modifications in ForEVeR. VA, SA and flow control units are monitered by runtime checkers. To implement recovery, NoC router is augmented with VC and speculation disablers along with a token manager and a recovery FIFO controller

correctness goal	verified property	sub	time(s)
No dropped flit (datapath and standalone control units)	* incoming valid flit written to buffer * buffer behaves in FIFO manner * flit gets from buffer to OP channel	6 20 17	90 660 170
Packet keeps wormhole (complex interactions of concurrent components)	* only valid body flits follow head flit with no mixing between packets and no flit/packet duplication	42	2400

Table 1: Formal verification of router correctness.

goals: ensuring no flits are dropped and proving that body flits follow the header flit in a wormhole. To prove that the router does not drop flits, it is necessary to verify that all valid flits received through input channels are written into valid buffer entries, followed by the verification of first-in first-out functionality of the buffers. Finally, it should be proven that a flit read from the input buffer should get to some output channel, within a fixed number of clock cycles depending on the router pipeline. Verifying that a packet maintains its wormhole nature is more involved as now correctness has to be proven over an entire packet rather than a flit. Apart from proving that flits follow the head flit, formal methods should ensure that no other flit from any other packet meddles with the wormhole. Also, it is essential to verify that only valid flits are transmitted and that there is no flit/packet duplication within the router. This requires extensive verification of various router control components and their interactions. Table 1 summarizes the correctness goals for NoC router and the entailed properties that require proof. To illustrate the verification procedure of writing specification properties, we provide an example where we discuss in detail how a correctness property is divided into sub-properties. The property '*incoming valid flits written to IP buffer*' is used as an example. This property holds if it can be verified that an incoming valid flit always has a valid VC tag and the corresponding VC buffer has a free slot (no overflow). Additionally, the flit contents should be written to the free slot of only the requested VC buffer. Finally, an incoming invalid flit should not be written to any of the VC buffers. Some router implementations maintain a separate header buffer corresponding to each VC buffer and thus similar properties should be verified for the header buffers, where instead of any valid flit, only valid header flits are considered.

The number of sub-properties (*sub*), expressed as System Verilog Assertions (SVA) [1], to represent each correctness property is also reported in Table 1. Finally, Table 1 states the time taken to verify each property using Synopsys Magellan [19] on an Intel Xeon

processor running at 2.27 GHz and using 4GB of main memory. It should be noted that formal guarantees for starvation freedom in allocation schemes and proper functioning of the route computation module need not be provided, as the network level detection and recovery scheme efficiently handles these scenarios.

4.2 Runtime Verification

As mentioned earlier, due to the area overhead of runtime checkers, only components that handle interactions between the concurrently operating modules are monitored. These components are the hardest to verify and result in majority of hard-to-catch bugs [9]. We also pointed out that the routing unit, VC allocator and switch allocator orchestrate the actions of input and output VCs and that the flow control unit interprets and communicates the control information between routers. Among these units, errors in the routing stage are not detrimental to our scheme as long as the other router functionalities are guaranteed to be correct. Thus the routing unit is not monitored at runtime. All other vital units are supervised for correct operation by runtime checkers, as shown in Figure 2. Once an error is flagged by these checkers, router level reconfiguration is performed that forces the router to a formally verifiable degraded mode, with minimum functionality to support network level recovery. This is followed by network level recovery initiation that we discuss in section 5.2.

4.2.1 Detection and Recovery

VC and switch allocator A design flaw in VC allocator may give rise to various erroneous conditions, some of which are benign as they either do not violate our definition of router correctness or are effectively detected and recovered by our network level correctness scheme. Assignment of an unreserved but erroneous output VC to an input VC is an example of such a benign error, as in the worst case it may only lead to misrouting or deadlock. Starvation is another example that needs no detection or remedy at a router level. Critical errors arise when an unreserved output VC is assigned to two input VCs or an already reserved output VC is assigned to a requesting input VC. This situation will lead to packet mingling and/or packet/flit loss. Similar to VC allocator, a design flaw in switch allocator may or may not have an adverse affect on ForEVeR's operation. An error in switch allocator may send a data flit to a different direction than the corresponding header flit; it may also cause the same flit to be sent to multiple outputs; or multiple flits from different packets to be directed towards the same output at the same time. All these cases lead to packet data corruption and an un-recoverable network state. To monitor VC and switch allocators at runtime for corrupt behavior, we propose the use of Allocation Comparator (AC) unit, that is a stripped down version of a similar unit that was proposed in [16] for soft error protection. The AC unit is purely combinational logic that performs all comparisons within one clock cycle. It simultaneously analyses the state of VC and switch allocators for duplicate or invalid assignments. If an error is flagged, all VC and switch allocations of the previous cycle are invalidated. Flits traversing the crossbar just after the error is flagged, are discarded at the output. To avoid loosing flits due to this invalidation/discard operation, an extra storage slot per VC buffer is reserved for use during such emergencies. To implement this, VA, SA and crossbar units are modified to accept invalidation command from the AC.

Flow control To safeguard against flow control errors, a hardware monitor is inserted to detect buffer overflow errors. Additionally, to avoid dropped flits, input buffers are equipped with two emergency slots per VC. On receiving a flit at buffer full condition, indicating an overflow, the downstream router tells the upstream router to switch to a slightly modified version of ACK-NACK flow control [8]. The second emergency slot is reserved for a possible in-flight flit during this upstream signalling. The modified ACK-NACK flow control eliminates the need for negative acknowledgements and re-ordering ability at the downstream router. This is achieved by stopping further transmission on the link until an acknowledgement is received for a previously transmitted flit. The flit awaiting acknowledgement is re-transmitted every two cycles (round trip latency of the links), before being dropped on receiving an acknowledgement. This scheme, though detrimental for performance, is extremely simple and can be implemented with little modification to the existing flow control mechanism. In addition, the router works in this mode only during recovery, switching back to its high performance mode after recovery is complete. Note that to safeguard against all errors at most two emergency slots per VC buffer are required. Since buffers are usually designed as circular FIFOs, this scheme entails only slight modifications to the *buffer full* logic.

4.2.2 Degraded Mode

When a bug is detected by hardware monitors, the router switches to a degraded mode with formally verified execution semantics, by either disabling complex units or replacing vital ones with simpler spare counterparts. This mode is equipped with bare-minimum features to support the network level recovery, that is initiated immediately after discovering a bug. To prevent the NoC routers from servicing new packets in probable erroneous state, all packet level operations such as route computation and VC allocation are disabled during recovery, as discussed in section 5.2. Similarly advance "performance only" features such as switch speculation and prioritizing mechanisms are disabled. Since stuck packets have to be drained out of NoC routers, it still requires the switch allocator and flow control manager to work properly. To this end, the router reconfigures to use a spare simple arbiter that polls each input VC for switch allocation. Similarly, flow control switches to an acknowledgement based mode to prevent flit loss as discussed in section 4.2.1. The resulting degraded router has significantly less concurrency and thus can be verified to function correctly.

5. NETWORK CORRECTNESS

With router correctness guaranteed, we need a network level solution that ensures forward progress in the NoC system. More specifically it should efficiently detect and recover from design errors that inhibit forward progress in the network (deadlock, livelock and starvation) and misrouting errors. To this end, ForEVeR adds a lightweight and verifiable checker network that works concurrently with the original NoC, providing a reliable fabric for transfer of notifications and recovered packets during detection and recovery phases respectively. Our checker network should be a simple, low latency optimized network that can consistently deliver notifications before the actual packets arrive through the primary network. We, therefore, leverage the single cycle latency, packet-switched routers of [10], organized as a ring network.

In the detection phase, each packet sent on primary network is accompanied by a corresponding notification over the checker network, both directed to the same destination. Each destination maintains a count of expected future packet deliveries through the primary network, decrementing the count on receiving a packet from the primary network. A distributed detection scheme monitors the counter values for zeros, initiating recovery on not observing a zero value during the entire *check epoch* of certain cycles. During the recovery phase, in-flight packets are recovered from the primary network, and reliably transmitted through the checker network. Fig-

ure 1 shows a baseline NoC augmented with the checker network. Interactions between the NoC router and checker network are handled by the NI unit, which also houses the detection and recovery initiation logic.

5.1 Detection

All design errors that inhibit forward progress result in packet(s) jammed within the network, and thus our detection mechanism should be designed to detect such scenarios. Moreover, it should be simple enough to be implemented with small area overhead and minimal changes to the existing infrastructure. To this end, we use notification messages travelling via the reliable checker network as the means for destinations to keep a count of the future packet deliveries. A bug in the primary network will always lead to an unaccounted packet at the destination, and thus the counter value will never go to zero, under the assumption that notifications always reach the destination before their counterpart packets. Therefore, our distributed detection scheme flags an error if it does not observes a zero counter value at any particular destination inside an entire *check epoch*. Figure 3 depicts the working of our distributed detection scheme. Counting logic is added to the NI to keep a count of number of expected packets at destination nodes, as shown in Figure 2. A timer monitors the counter value for zeros during entire *check epoch* length, failing which recovery is triggered. With proper size of the *check epoch*, this simple scheme is effective in catching bugs as we show in our experimental results section 6.1 and it can be implemented with lightweight counting logic. On the other hand, misrouting errors that do not cause deadlock or livelock are detected at destinations by analyzing the routing information carried by header flit.

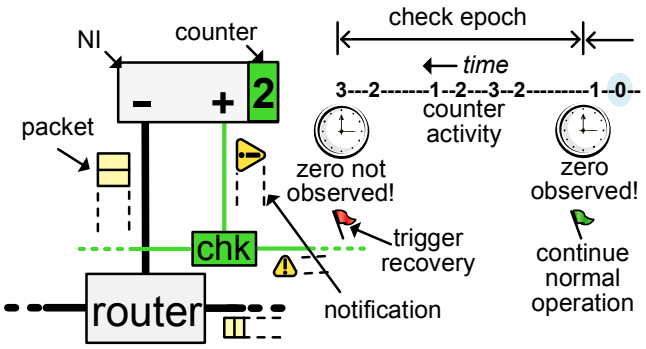

Figure 3: ForEVeR's detection scheme. Each destination tracks the notification counters for zero values. Recovery is triggered if zero is not observed during the entire *check epoch* at any destination.

5.2 Recovery

When an error is reported either by the router level runtime monitors or by the network level detection scheme, the NoC enters a unified recovery phase, consisting of *network drain* step followed by a *packet recovery* step. In *network drain* phase, the network is allowed to operate normally to drain its in-flight packets for a preset amount of time, with the exception of switching the erroneous routers to a degraded mode if recovery was initiated by router level checkers. During this phase, new packets are not injected into the network, as shown in Figure 4(a). Recovery is aborted at the end of *network drain* if all destinations receive the packets they were expecting, indicating a false positive due to the limited accuracy of the detection scheme. It should be noted that false positives, though a performance hit in absence of errors, do not affect the correctness of the system.

The network then enters *packet recovery*, where we try to recover packets that are stuck within the network. To this end, a token is circulated through all routers in the NoC via the checker network, and NoC routers can operate only when they hold this token. In addition, VC allocators of all the NoC routers are disabled to prevent them from processing new data packets from neighboring routers. When a router receives the token, it examines the fronts of its VC buffers in a serial manner, looking for packet headers. In case of a successful search, the packet is retrieved and sent over the checker network as shown in Figure 4(b). Since vital router functionalities for packet drain are still active (even in the degraded mode), the entire packet can be safely diverted to its destination through the checker network. Once the token has circulated through all primary routers, the entire process of *packet recovery* is repeated until either each destination receives all the pending packets or no more packets are retrieved, in which case a design bug has slipped through our scheme. To enable the ForEVeR scheme, NoC routers are augmented with certain simple units, as shown in Figure 2. First, a token manager is added to the routers to manage token passing. In addition, *virtual channel (VC) allocation disabler* (VC-DIS) and *switch speculation disabler* (SPEC-DIS) are included to prevent routers from processing new packets during the packet recovery phase and to keep the ForEVeR operation simple and easily verifiable for correctness. The recovery operation is implemented with very little overhead, making use of the router's existing functionalities to drain out packets from their buffers, with the help of the FIFO recovery controller.

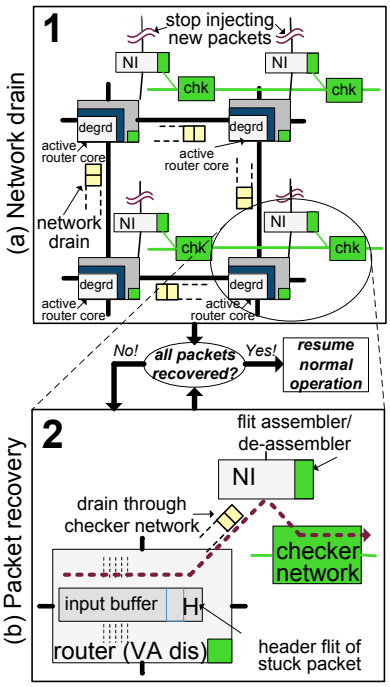

Figure 4: ForEVeR recovery process. Network drain is followed by packet recovery until all primary network packets are recovered.

Due to the limited bandwidth of the checker network, each primary network flit is transmitted as several checker packets. During recovery, only one router is transmitting its stuck packets to a single destination at a time, greatly simplifying the dis-assembling/assembling process. To send the entire primary network flit as multiple checker packets, the channel of the checker network is augmented with head and tail indicators. The flit with head indicator carries the destination address and reserves an exclusive path between the source and

one particular destination. All intermediate valid flits traversing the ring network are ejected at the same destination till a flit is received with a tail indicator, in which case the process repeats itself on transmission of another flit with a head indicator. Moreover, all transmissions on the checker network during recovery occur in the same (clockwise) direction to avoid wormhole overlap of two packets. In our evaluation system with 64 nodes, the checker network channel is 8 bits wide (6-bit address, 2-bit head-tail indicators). Thus each 64-bit primary network flit takes 12 checker networks packets (1 head, 11 body/tail) to transfer.

5.3 Verification of Recovery Operation

All components involved in the detection and recovery processes must be formally verified to guarantee correct functionality. Verification of the detection mechanism involves ensuring the correct functioning of the counting and timer logic at NIs and due to the simplicity of the logic involved this makes up for a trivial verification task. Formally verifying the recovery operation is more involved and requires two major tasks: first, verifying the checker network functionality; and second, verifying the interaction between checker and primary network during recovery, to ensure proper restoration of erroneous packets.

Checker network. It should be verified that the checker network correctly delivers all packets to their respective destinations within a bounded time. To this end, this correctness goal was partitioned into three sub-properties: *eventual injection (inj_prop)*, guaranteeing injection of a waiting packet into the network; *forward progress (fw_prop)* ensuring that packets progress on a path towards their destination; and *timely ejection (ej_prop)* that guarantees packet ejection at correct destination.

Interaction with primary network. The primary network's units that interact with the checker network to salvage stuck packets from the primary routers must be ensured to function properly. During recovery, primary routers work in a rudimentary mode by disabling all complex hardware units not involved in the recovery process, such as the VC allocators and SW speculators, thus making the verification task tractable. First, it is verified that the checker network could extract a complete packet from an individual primary router's VC buffer (*rec_prop*), leaving it empty (*rec_emp_prop*). The complement of this property is also validated (*not_rec_prop*) to check that only valid packets are extracted from the primary network. This was followed by checking for fairness and exclusivity among the primary routers while undergoing recovery (*fair_ex_prop*), ensuring that packets are salvaged from one router at a time.

correctness goal	verified property	time(sec)
Checker network correctness	inj_prop	8
	fw_prop	156
	ej_prop	86
Interaction with primary network	rec_prop	15
	rec_emp_prop	10
	not_rec_prop	46
	fair_ex_prop	29

Table 2: Formal verification of ForEVeR's recovery operation

Table 2 summarizes the correctness goals for ForEVeR's recovery process and the time required to prove the etailed properties.

6. EXPERIMENTAL RESULTS

We evaluated ForEVeR by modeling a NoC system in Verilog HDL, as well as with a cycle-accurate C++ simulator, both based on [8]. Our system is a 8x8 XY-routed mesh network, using routers with 2 VCs and 8 entry buffers per VC, that are similar to the router described in section 4. In addition, the NoC was augmented with a checker network enabled with detection and recovery capabilities. The Verilog implementation was used to formally verify the NoC routers and the hardware components taking part in the recovery process. To this end, we inserted specification properties expressed as System Verilog Assertions [1] in the HDL design and verified them with Synopsys Magellan [19], a model checking tool. ForEVeR's area overhead was estimated using synthesis results from Synopsys Design Compiler [18] targeting the Artisan 130nm library. On the other hand, the C++ simulator was used to assess the accuracy of our network level detection scheme and to evaluate the performance impact of recovering from functional bugs that were deliberately inserted in our model. The scheme was analyzed with two different types of workloads: directed random traffic (uniform), as well as applications from the PARSEC suite [5].

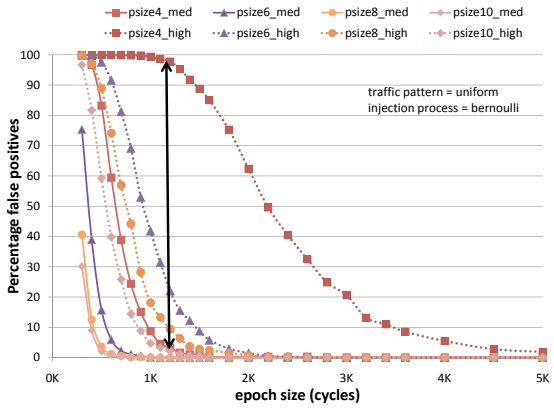

Figure 5: ForEVeR's detection scheme under uniform random traffic. Figure plots the false positive rate with increasing *check epoch* size, for various packet sizes. False positive rate drops rapidly with larger *check epoch* sizes and decreasing network load.

6.1 Network Level Detection Accuracy

ForEVeR's runtime performance overhead is affected by the accuracy of its detection scheme. False positives in absence of bugs lead to unnecessary recovery initiation where injection into the network is stopped while it is drained. Given a sufficiently long draining period, the system will resume normal operation after the network drain phase as all in-flight packets in the error free network will be delivered correctly to their destinations. Thus a false positive in our detection mechanism does not affects network's correctness but hits the runtime performance. The false positive rate of our detection scheme depends on the duration of *check epoch* relative to traffic conditions. Note that false positives are triggered when the destination counter is non-zero for the entire *check epoch*, and hence a heavily loaded network will trigger more false recoveries as unaccounted notifications will accumulate at destinations with their corresponding packets being delayed due to congestion in the primary network. Intuitively, a longer *check epoch*, though increasing the detection latency, will also reduce the false positive rate by giving more time to the latency hit primary network packets to balance the notifications accumulating at the destination nodes. Figure 5 shows a study using uniform random traffic with varying packet sizes, that plots the false positive rate in the absence of bugs with increasing *check epoch* size. It can be seen that false positive rate decreases rapidly with the increasing *check epoch* size, with the false positive rate dropping to a negligible value beyond a certain *check epoch* size ($Epoch_{min}$), the value of which depends on network load. It can also be seen that for the same *check epoch* size, a heavily loaded network exhibits a much higher false positive rate than a moderately loaded network.

Despite using a large *check epoch*, a heavily loaded network may

result in a destination continuously receiving new notifications before the "expected packet" count is neutralized by primary network packets, therefore initiating a false recovery. Extensive simulations indicate that this scenario is possible only when the network is operated at loads well past its saturation. However, NoC workloads are characterized by self-throttling nature of applications, which prevents them from operating past saturation [15], making a good case for our detection scheme. In addition, large *check epochs*, used to prevent false positives at high network load, can be tolerated even though it increases the detection latency as this affects system performance only when a bug manifests. To validate our design's effectiveness and to calibrate the *check epoch* size, we ran rigorous simulations using both uniform random traffic and PARSEC benchmarks. After operating ForEVeR normally for a set period, we drop a random primary network packet to imitate an error in the primary network, calculating the number of false positives and negatives for various *check epoch* lengths at the same time. A design flaw in the primary network will at least result in one stuck main network packet, and hence this simulation technique models the worst-case scenario.

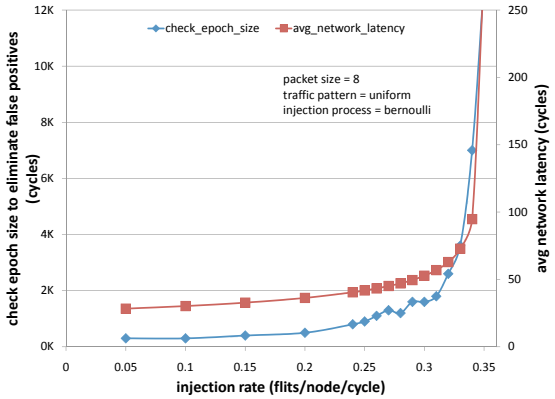

Figure 6: ForEVeR's detection scheme under uniform traffic. Figure shows the variation of $Epoch_{min}$ and latency with increasing network load. $Epoch_{min}$ is within tolerable limits for deeply saturated networks.

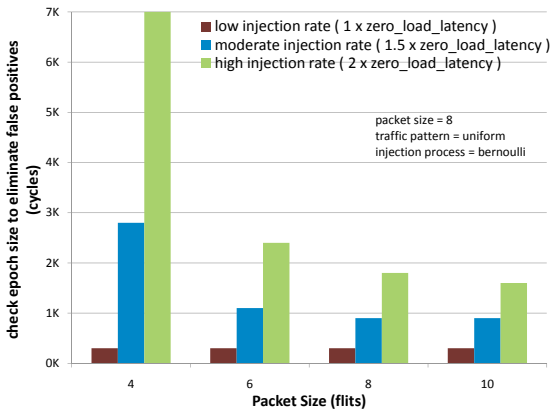

Figure 7: ForEVeR's detection under uniform traffic. Figure shows the variation of $Epoch_{min}$ with different packet sizes, at low, moderate and high network load. $Epoch_{min}$ size decreases with increase in packet size.

Uniform random traffic Figure 6 plots the average network latency and the minimum *check epoch* length at which the false positives are practically eliminated ($Epoch_{min}$) as traffic load on the network is varied. Note that the value of $Epoch_{min}$ was determined by extensive simulations and a theoretical determination of a lower bound on *check epoch* length to completely eliminate false positives is not required as false positives due to in-accuracy in the estimated value of $Epoch_{min}$ entail only a performance penalty and are not a correctness issue. $Epoch_{min}$ exhibits a slow increase with rising injection rate, till network saturation, from where it shows a steep rise. In the worst case shown by the graph, $Epoch_{min}$ of 7K cycles is sufficient to eliminate all false positives, when the network is in deep saturation, operating at an average latency of about 4 times the zero-load latency. Figure 7 shows a similar study plotting $Epoch_{min}$ at low, moderate and high injection rates for four different packet sizes. Apart from showing that $Epoch_{min}$ is within manageable limits in all cases, it also indicates the decrease of $Epoch_{min}$ value with increasing packet size. For similar loads, the network using larger packet sizes has fewer in-flight packets causing fewer notifications to accumulate at destinations and hence lower $Epoch_{min}$ values.

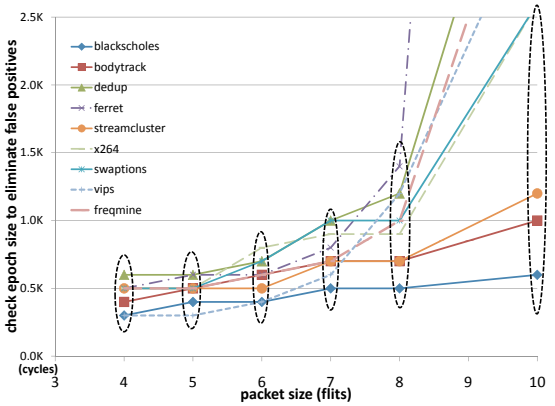

Figure 8: ForEVeR's detection scheme under PARSEC benchmarks. Figure shows the variation of $Epoch_{min}$ with different packet sizes for 9 PARSEC benchmarks. $Epoch_{min}$'s size across all benchmarks is within tolerable limits for packets of size up to 8 flits (avg. network latency of 800 cycles)

PARSEC Benchmark traces for evaluation of our detection and recovery scheme were extracted from a 64 core CMP system running PARSEC workloads, with our baseline NoC using 4-flit data packets as the on-chip interconnect. The average network latency across all benchmarks for these traces was 26 cycles. However, to examine our scheme under more demanding conditions, we decreased the channel width of our baseline NoC, in effect increasing the network load. Using the same traces, NoCs with longer data packets (due to decreased channel width) are used to unrealistically stress the network during simulation. It should be noted that such high load scenarios (average network latency up to 1600 cycles) will never arise in an actual NoC because of the self-throttling nature of the applications. Figure 8 plots the variation of $Epoch_{min}$ with different packet sizes for the benchmarks of the PARSEC suite, demonstrating that zero false positives can be achieved with reasonable *check epoch* sizes even with high network loads.

Finally to avoid false negatives in our scheme, the checker network is constrained to deliver notifications before the corresponding data packets arrive through the primary NoC. A checker network design capable of this can be achieved by considering several design alternatives, such as bundling together multiple notifications before transmission, or using multiple checker networks, etc. In our evaluation system, the checker network almost always delivers notifications ahead of data packet delivery through the primary NoC, except for certain very low latency cases, where certain main network packets take shorter routes through the primary NoC, whereas the notifications travel longer routes in a ring-based checker net-

work. To counter this case, the destination counters, rather than updating immediately on arrival of primary NoC packets, are updated after a certain delay period that is determined by the maximum latency difference between primary and checker network possible at zero load.

6.2 Recovery Overhead

To analyze ForEVeR's performance impact and its ability to recover from various types of design errors, we injected 9 different design bugs into our C++ implementation of ForEVeR, as described in Table 3. Bugs 1-8 are errors that inhibit forward progress or lead to incorrect routing and are detected by the network-level detection scheme, whereas Bug 9 is detected by router-level hardware checkers after being randomly activated in any one router servicing a packet. We ran PARSEC workloads while triggering one bug during entire execution and varying bug trigger time (5 points, 10,000 cycles apart), place of bug injection (10 random places) and packet size (4,6,8 flits). ForEVeR was able to detect all errors with no false positives or negatives and recover from them, executing all workloads to completion and delivering all packets correctly to their destinations. ForEVeR's performance overhead on bug manifestation is due to the recovery phase that includes *network drain* and *packet recovery*. During *network drain*, phase the primary NoC is allowed to drain for a fixed period of 500 cycles, that was determined by simulations as the time required to drain a congested network. Table 3 reports the average *packet recovery* time (in network cycles) taken by each bug, averaged over all benchmarks, packet sizes, activation times and places of bug injection. It should be noted that routing errors are quickly detected at erroneous destinations, whereas errors that effect the router operation are detected immediately by the router level hardware monitors.

bug	description	recovery time
deadlock	some packets deadlocked in the network	4821
livelock	some packets in a livelock cycle	3084
VA_vc_strv	input VC never granted an output VC	2827
VA_port_strv	no input VC in a port granted output VC	3055
SW_vc_strv	one input VC never granted switch access	2123
SW_port_strv	no input VC in a port granted switch access	2490
misroute1	one packet routed to random destination	1724
misroute2	two packets routed to random destinations	1810
router_bug	hardware monitors in routers detect a bug	1764
average		2633

Table 3: Functional bugs injected in ForEVeR. Table reports average recovery time (*recover*) for each bug.

On average, ForEVeR spends approximately 2,633 cycles in the *packet recovery* phase. Since, ForEVeR's main overhead is because of draining packets stuck in the primary NoC through the checker network to their respective destinations, the recovery time directly depends on the number of stranded packets. Therefore bugs that affect larger parts of the design, such as an entire port (*VA_port_strv*), take more time to recover than bugs that influence smaller portions, such as one VC (*VA_vc_strv*), as they usually involve greater number of packets. Similarly, *deadlock* errors that potentially stop many packets from progressing, require the highest number of cycles to recover.

6.3 Area Results

Since verification dedicated hardware does not contribute to performance or functionality of the design, it should be architected to be as area efficient as possible. The amount of hardware required to implement router level correctness is minimal and varies with the designer's ability to verify different router components and hence we focus on the area overhead of ForEVeR's network-level runtime detection and recovery scheme. Table 4 reports our results indicating that ForEVeR leads to a 5.8% area overhead over a primary network router. Area overhead is dominated by additions made to the router for enabling the network-level recovery scheme, contributing 2.7%, whereas NI additions and checker router combined are responsible for the remaining 3.06%. Moreover, wiring cost due to additional buffer port used during recovery is the major contributor to router area overhead.

baseline (8x8 mesh)			ForEVeR's overhead		
design	**area (mm^2)**	**%**	**design**	**area (mm^2)**	**%**
router	5.167	100	router additions	0.139	2.70
			NI additions	0.077	1.49
			checker router	0.081	1.57
total	5.167	100	overhead	0.298	**5.76**

Table 4: ForEVeR area overhead for network level correctness.

7. CONCLUSIONS

In this work, we presented ForEVeR, a complete verification solution that complements the use of formal methods and runtime verification to ensure functional correctness in NoCs. Formal verification is used to verify simple router functionality, leveraging a network-level detection and recovery scheme to provide NoC system correctness guarantees. ForEVeR augments the NoC with a simple checker network used to communicate notifications of future packet deliveries to corresponding destinations. A runtime detection mechanism keeps a count of expected packets, triggering a recovery on unusual behavior of the counter values. Following error detection, all in-flight packets in the primary NoC are safely drained to their intended destinations via the checker network. ForEVeR's detection scheme is highly accurate and can detect all design errors. The complete scheme incurs only 5.8% area cost for 8x8 mesh NoC, taking only up to 30K cycles to recover from errors.

8. REFERENCES

[1] System Verilog Assertions. http://www.systemverilog.org/.

[2] K. V. Anjan and T. M. Pinkston. An efficient, fully adaptive deadlock recovery scheme: DISHA. In *Proc. ISCA*, 1995.

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

[4] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. In *Proc. ICCAD*, 2005.

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

[6] D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. A generic model for formally verifying noc communication architectures: A case study. In *Proc. NoCs*, 2007.

[7] M. Boule, J.-S. Chenard, and Z. Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Proc. ISQED*, 2007.

[8] W. Dally et. al. *Principles and Practices of Interconnection Networks*. 2003.

[9] H. Foster, L. Loh, B. Rabii, and V. Singhal. Guidelines for creating a formal verification testplan, 2006.

[10] J. Kim and H. Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proc. NoCArc*, 2009.

[11] P. Lopez, J. M. Martinez, and J. Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proc. HPCA*, 1998.

[12] J. M. Martínez et. al. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proc. ICPP*, 1997.

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

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

[15] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next generation on-chip networks: what kind of congestion control do we need? In *Proc. Hotnets*, 2010.

[16] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proc. DSN*, 2006.

[17] D. Starobinski et. al. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Networks*, 11(3), 2003.

[18] Synopsys. Synopsys Design Compiler.

[19] Synopsys. Synopsys Magellan. http://www.synopsys.com.

[20] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *Proc. DATE*, 2007.