# ForEVeR: A Complementary Formal and Runtime Verification Approach to Correct NoC Functionality

RITESH PARIKH and VALERIA BERTACCO, University of Michigan, Ann Arbor

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. As a result, design errors in the NoC may go undetected and escape into the final silicon.

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 deliver correctness guarantees for the complete network, we propose a network-level detection and recovery solution that monitors the traffic in the NoC and protects it against escaped functional bugs. To this end, ForEVeR augments the baseline NoC with a lightweight checker network that alerts destination nodes of incoming packets ahead of time. If a bug is detected, flagged by missed packet arrivals, our 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 4.9% area cost for an 8x8 mesh interconnect, over a time interval ranging from 0.5K to 30K cycles per recovery event, and it incurs no performance overhead in the absence of errors. ForEVeR can also protect NoC operations against soft-errors: a growing concern with the scaling of silicon. ForEVeR leverages the same monitoring hardware to detect soft-error manifestations, in addition to design-errors. Recovery of the soft-error affected packets is guaranteed by building resiliency features into our checker network. ForEVeR incurs minimal performance penalty up to a flit error rate of 0.01% in lightly loaded networks.

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

General Terms: Verification, Reliability

Additional Key Words and Phrases: Network-on-chip, NoC, functional correctness, formal verification, runtime verification

**ACM Reference Format:**
Ritesh Parikh and Valeria Bertacco. 2014. ForEVeR: A complementary formal and runtime verification approach to correct NoC functionality. ACM Trans. Embedd. Comput. Syst. 13, 3s, Article 104 (March 2014), 30 pages.
DOI: http://dx.doi.org/10.1145/2514871



## 1. INTRODUCTION

Current trends in microprocessor design entail the inclusion of an increasing number of relatively simple processor cores communicating via an interconnect fabric. Correspondingly, embedded systems deploy system-on-chip architectures, comprising several IP components in one single chip. As a result, the demands for high bandwidth inter-core communication have rapidly sidelined traditional interconnect architectures, such

This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and NSF grant #0746425.
Author's address: R. Parikh; email: parikh@umich.edu; V. Bertacco; email: valeria@umich.edu.

DOI: http://dx.doi.org/10.1145/2514871

as simple buses, due to their limited scalability and performance. In contrast, networks-on-chip (NoCs) are characterized by highly concurrent communication paths and better scalability, and are thus becoming the de-facto choice for interconnect architectures. Moreover, to keep up with the performance of the cores/IPs on-chip, NoC design is becoming increasingly complex, employing various techniques to efficiently manage high communication loads. In NoCs, data is transmitted as packets, that can further be divided into smaller, fixed length blocks, called flits, for efficient transfer. Packets injected via network interfaces (NI) are transmitted to their destinations through a network of routers and links, abiding some routing protocol. The routers themselves often include advanced features, such as pipelining, speculation, prioritization, complex allocation schemes, etc., and are organized in a wide range of topologies, implementing complex routing algorithms. With these advanced performance features, it is a challenge to ensure correct functionality under all circumstances for the entire network.

Despite massive industry efforts in verification, escaped functional bugs that manifest at runtime are a reality. This is a prominent issue in processor designs, where design bugs are often detected in the field after product release. We did a study on the number of escaped bugs by compiling information collected from several processor errata documents [Intel 2007, 2010]. The results of this study are shown in Figure 1 for various generations of Intel processors. In the figure, we plot the number of bugs that are discovered against a timeline. It can be seen that there has been a steady increase in the number of bugs detected per month with each design generation, especially as the designs moved to dual-core (Core 2 Duo) and multi-core (Core i7). We also observed that as a result of the large number of interactions and the intricate communication in modern multi-core CMPs, errors in the communication subsystem now account for a significant portion of the reported bugs. For example, in the Core 2 Duo and Core i7, at least 10% and 13% of the reported design errors are communication system related [Intel 2007, 2010]. The erratas provide brief and product-specific description of the bugs, and it is usually hard to partition them in subcategories. Note that, the reported percentages cover bugs related to the entire communication infrastructure, including communication protocols like cache coherence, message dependent deadlocks, and bugs in both the implementation and protocols of on-chip communication fabric (bus in this case). As CMPs transition towards more complex NoC-based interconnects, this trend is expected to continue, with a lot of hard-to-find bugs rooted in the communication infrastructure, and particularly in the implementation and protocols of the communication fabric, which is the target of this work.

Pre-silicon verification efforts are used to ensure correctness using a combination of simulation based verification and formal methods. Simulation based verification, though helpful in catching many easy to find bugs, will always be incomplete as it cannot exhaustively test the countless different execution scenarios within a network. However, formal methods such as model checking, are complete, but only effective in verifying small portions of the design and they do not scale to verify end-to-end system level correctness. Recently, the research community has started exploring runtime verification solutions where the system's activity is monitored at runtime, after product deployment, and checked for correctness. Runtime verification can reduce the cost of design bugs that escape design-time verification by detecting their occurrence and preventing the corruption of network/processor state, loss of data and/or failing of the entire system. Their cost, however, includes silicon area for runtime monitoring and recovery, dedicated design effort and often a performance impact on the overall system due to continuous monitoring activities.

To counter the shortcomings of design-time verification, a simplistic approach would be to naïvely safeguard against all possible failure scenarios at runtime. Such an

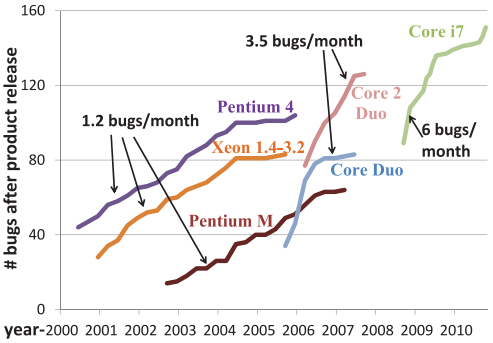



Fig. 1. *Bugs discovered after deployment* for few Intel processors. The discovery rate increased drastically from uni-core to multi-core processors. A significant percentage of the bugs relate to the interconnect.

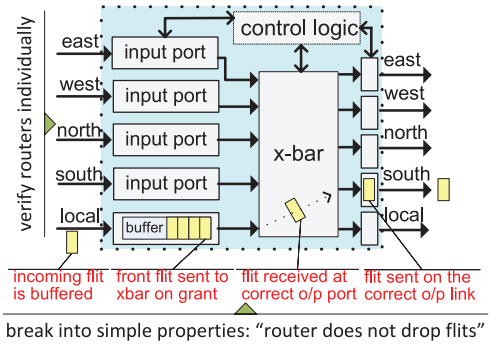



Fig. 2. *ForEVeR's verification guidelines.* State explosion avoided by (i) verifying routers individually, (ii) breaking specification into simple subproperties, and (iii) checking liveness requirements at runtime.

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 reduce the burden on runtime bug detection and recovery. Thus, ForEVeR proposes a complementary functional verification solution for networks-on-chip, which leverages formal techniques for individual network routers and components, and runtime verification for the network-level. ForEVeR's runtime modules are designed to protect only those aspects that cannot be formally proven to work correctly. In this manner, the silicon area and design-time effort dedicated to runtime verification directly benefits from the designers' ability to formally verify.

ForEVeR also detects and recovers from network corruptions because of soft errors: an increasing concern with waning reliability of silicon. Recent studies [Dixit and Wood 2011; Nightingale et al. 2011] confirmed the alarming rate of soft errors affecting both logic and memory components. NoCs are susceptible to soft errors as they occupy a significant portion of the on-chip real estate [Vangal et al. 2008; Wentzlaff et al. 2007], and additionally, they suffer from single event upsets (SEUs) due to crosstalk and coupling noise in link wires. ForEVeR protects the NoC against soft errors affecting all component types (control, datapath and links), and does so by mostly reusing hardware dedicated to our design error detection and recovery scheme.

### 1.1. Complementary Verification: Motivation

Model checking, the most widely adopted formal verification technique in industry, has been successfully applied to verify many component-level designs [Foster et al. 2006; Brayton and Mishchenko 2010]. However, model checking suffers from state space explosion when used on large systems. To determine the appropriate granularity for applying formal methods to NoC verification, we performed a case-study on a simple $2\times2$ mesh NoC design, with 5-port wormhole routers [Dally and Towles 2003]. We wrote System Verilog (SVA) properties describing high-level network behavior and tried to formally verify them using Synopsys Magellan. For example, we verified the property that "network does not drop any flits." To this end, a traffic generator injected one data packet and a SVA property tested for a single packet ejection from the NoC within a time window. The model checking engine was unable to complete the task. Therefore, to avoid state explosion, we experimented with properties that can be verified at the

router-level. However, formal verification would fail even in these scenarios if properties were not specified carefully. In general, it was hard to verify liveness properties, and bounded properties with a large time bound.

However, in our subsequent efforts, we were successfully able to verify router-level specifications by breaking them down into simple subproperties. An example of the property: for instance, "router does not drop any flits," is shown in Figure 2. For this particular example, we also decoupled the liveness requirement ("flit leaves eventually"), with the property we targeted to verify ("router does not drop any flits"). Our analysis led us to three important guidelines that form the basis of ForEVeR: i) verifying properties at router-granularity is tractable, in contrast to monolithic network-level verification, ii) broad-scope properties can be tackled by breaking them into simple subproperties, and iii) formal verification is inadequate in proving liveness or large time-bound properties: the bugs that cannot be exposed in the process are good candidates for runtime detection and recovery.

### 1.2. Contributions

ForEVeR (*F*ormally *E*nhanced *V*erification at *R*untime for NoCs) targets functional bugs in the NoC fabric, and it is a solution independent from topology, router architecture and routing schemes. Leveraging the synergy between formal and runtime verification, ForEVeR can detect and recover from all types of functional errors in the interconnect, including bugs that stall the forward progress of the network, for instance, deadlock. The runtime component of ForEVeR also enables the designers to deliberately implement aggressive allocation, routing or prioritization schemes, that may occasionally lead to starvation, deadlock or livelock. If these scenarios are rare, the overall performance improvement outweighs the recovery overhead. 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. Moreover, ForEVeR comes at a small area cost of 4.9% for an $8\times8$ mesh interconnect, while incurring a minimal performance impact only when an error manifests. Finally, ForEVeR also protects the NoC against soft errors, utilizing the same detection and recovery hardware used for ensuring NoC correctness. With only a few hardware enhancements, ForEVeR can provide soft error coverage comparable to state-of-the-art reliability techniques.

## 2. RELATED WORK

*NoC Correctness.* Very few research works have proposed complementary use of formal and runtime techniques. Among them, Bayazit and Malik [2005] leveraged hardware checkers in model checking to avoid state explosion by validating abstractions at runtime. However, Bayazit and Malik [2005], unlike ForEVeR, cannot be directly applied to ensure NoC correctness. Another work proposed property checking at both design-time and runtime [Tsiligiannis and Pierre 2012]. However, as recovery from design errors in not supported, this is not a complete correctness solution.

NoC simulator distributions [Dally and Towles 2003; Fazzino et al.], RTL simulation and emulation platforms [Hammami et al. 2012] are widely used for performance and functional verification. However, both simulation and emulation techniques are inherently incomplete, since they cannot check all possible execution scenarios. In contrast, formal techniques can provide the guarantees of complete correctness; however, they either cannot be automated, as in theorem proving or they are limited by the state explosion problem, as in model checking. This has led designers to use formal verification exclusively for small portions of NoC designs [Kailas et al. 2009] or to verify the abstracted model of the implementation [Chatterjee et al. 2012; Borrione et al.

2007; Holcomb et al. 2011]. Other works [Borrione et al. 2007; Verbeek and Schmaltz 2012] utilize theorem proving to guarantee NoC correctness and are also able to verify liveness properties like deadlock-freedom [Verbeek and Schmaltz 2011]. However, these abstracted NoC models, often do not model advance features like virtual channels, flow control techniques, etc. Moreover, properties proven over the abstracted NoC model cannot guarantee correctness of the actual microarchitectural implementation. In contrast, ForEVeR guarantees the correctness of all the executions of the NoC implementation, and also enables designers to deploy aggressively designed NoCs that are not exhaustively verified.

Ensuring the runtime correctness of NoCs has been the subject of much previous research, focusing on a variety of aspects. Several of these works target deadlock, a prominent issue in adaptive routing. Traditionally, the deadlock problem in NoCs is overcome by deadlock avoidance [Starobinski et al. 2003; Dally and Seitz 1987; Lin et al. 1994], or through detection [Martinez et al. 1997; Lopez et al. 1998] and recovery [Anjan and Pinkston 1995]. Other works tackle a wider, still incomplete, set of NoC errors through end-to-end detection and recovery techniques, and are surveyed in [Murali et al. 2005]. The most common among these is the acknowledgement-based retransmission (ACK-Retransmission) technique, where error detection codes are transmitted along with data packets, to check for data corruption at the receiver. An acknowledgement is sent back after each successful transfer. In case of failure, the sender times out and retransmits the locally stored packet copy. Apart from large storage buffers and performance degradation due to the additional acknowledgement packets, this approach is incapable of overcoming deadlock, livelock and starvation errors. Moreover, since it uses the same untrusted network for retransmission, ultimately it cannot guarantee packet delivery. In contrast, ForEVeR safeguards against all kind of functional bugs, including, i) bug manifestations that cause data corruption, such as, dropped, duplicated or misrouted packets, and ii) bug manifestations that stall the forward progress of the network, such as, deadlock, livelock, and starvation. In addition, ForEVeR is an end-to-end solution leveraging hardware units mostly decoupled from the primary NoC and requiring minimal changes to the primary NoC.

SafeNoC [Abdel-Khalek et al. 2011] is an alternative end-to-end runtime solution for NoC correctness, which uses a secondary verified network to transfer data checksums for error detection. During recovery, in-flight data is collected and send to the processors that reconstruct the original data packets in software. Although, ForEVeR shares with SafeNoC the use of a secondary network, its complementary verification technique provides a low overhead solution that guarantees NoC functional correctness under all execution scenarios. SafeNoC, being a pure runtime solution, exhibits high area and design overhead, and in addition, it has several failure scenarios. First, it provides no protection against dropped packets or flits. Second, it cannot recover from errors arising from aliasing of signatures and, finally, the reconstruction algorithm does not guarantee completion. Several orthogonal runtime verification proposals [Austin 1999; Meixner et al. 2007; Wagner and Bertacco 2007] focus on microprocessor correctness. In general, these solutions monitor the operation of untrusted components, switching to a verifiable degraded mode upon detection.

Finally, ForEVeR's detection mechanism relies on the use of router-level runtime monitors, when formal methods fail to ensure router correctness. Runtime checkers has been proposed for various purposes, like soft-error induced anomaly detection in NoCs [Prodromou et al. 2012; Park et al. 2006] or post-silicon debug and in-field diagnosis [Boule et al. 2007]. In contrast, ForEVeR leverages specialized hardware monitors coupled with recovery support, specifically for NoC correctness.

*NoC Reliability*. The most common reliability techniques augment NoC packets with ECC information that can be checked on an end-to-end or a switch-to-switch basis [Murali et al. 2005; Dutta and Touba 2007]. ECC codes can also correct a few bit errors, beyond which the system relies on costly data retransmission using a backup copy. Retransmission schemes cannot tackle all erroneous scenarios, especially the ones arising from control logic malfunction, for instance, deadlock. Aisopos and Peh [2011] proposed to retransmit clean copies of data directly from the memory subsystem, alleviating the need of large backup buffers. However, the required changes to the cache coherence protocol present a design and verification challenge. Further, their solution also cannot recover from deadlock scenarios. A technique to recover from soft errors in both the router's data-path and control logic was presented in Park et al. [2006]. It uses switch-to-switch ECC and retransmission for data-path errors, while the router control logic is protected with hardware checkers. Deadlocks due to soft errors are broken by using additional buffers at each router port. However, this approach leads to a prohibitive area overhead (as we will show in Section 7.5), especially for networks with large data packets. In contrast, ForEVeR does not require backup data storage for soft error recovery, and it can also overcome forward progress errors such as deadlock. Additionally, ForEVeR also protects both the router data-path and control logic against soft errors, at low design complexity.

## 3. METHODOLOGY

ForEVeR addresses the correctness of a NoC by attacking the problem both at design-time and at runtime. During system development, ForEVeR recommends a methodology for formal verification of the individual routers' properties that do not require any network-wide knowledge. Specifically, ForEVeR's router-level verification ensures that routers maintain packet integrity. In the case that even individual network routers are too complex to be amenable to formal verification, ForEVeR proposes runtime hardware to monitor only those aspects of the routers' functionality that could not be verified during system development.

To complete the protection against design errors, the network-level behavior is monitored at runtime, assuming correctness of operations internal to the individual routers. Specifically, ForEVeR's network-level runtime hardware detects errors that stall NoC's forward progress, and in addition, it also provides a mechanism to recover from such errors, once the hardware monitors expose the occurrence of an anomaly. ForEVeR leverages the same network-level recovery mechanism to also overcome functional errors that are flagged by the router-level runtime monitors.

As discussed in Borrione et al. [2007], the functional correctness of a NoC can be organized along four high-level requirements. Three of them can be satisfied by guaranteeing their validity locally at each network router: *no_packet_drop*, requires that no packet is lost while traversing the network; *no_data_corruption* states that packets' payloads should not become corrupted while traveling from source to destination; finally, *no_packet_create* requires that no new packet is generated within the network (packets can only be injected from network's source nodes). If each individual router satisfies these properties, then they hold for the NoC system as a whole, since network links are simple wires and cannot embed functional bugs that corrupt, create or drop packets. Finally, the last requirement (*bounded_delivery*), specifies that each packet is delivered to its intended destination within a finite amount of time and it ensures that there is forward progress in the transmission. This last requirement cannot be validated locally, since it affects the entire network.

To this end, Figure 3 shows a high level overview of the hardware additions required by ForEVeR and, in particular, it highlights the components required to enable *bounded_delivery*. Partially verified routers are connected together to form a NoC that

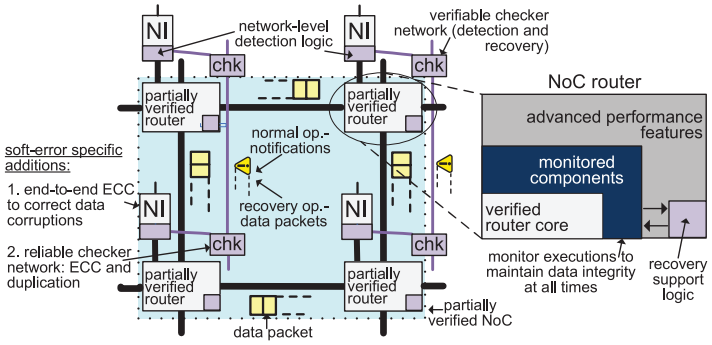



Fig. 3. *High-level overview of ForEVeR.* A combination of router-level verification/runtime-monitoring and network-level detection and recovery ensures correct NoC operation. All components in purple indicate ForEVeR hardware additions. The figure also highlights ForEVeR's hardware additions for soft-error protection.

is completed by detection and recovery logic. Runtime checkers and recovery logic are used at the router-level to protect complex router components against design flaws (if they cannot be formally verified at design-time). In addition, the primary NoC is augmented with a lightweight checker network, used to transmit advanced notifications to the monitors at the destination nodes. During recovery, the checker network is also used to reliably deliver in-flight data packets to their respective destinations. Note that each component of a router can be classified as i) verified at design-time, ii) monitored at runtime or iii) providing advanced performance features to be disabled during recovery.

Figure 3 also highlights ForEVeR's soft-error protection scheme. We make the following observations to provide a cheap soft-error protection solution with ForEVeR: first, the router-level monitors designed to detect anomalous behavior due to design errors, can also prevent the network from entering an unrecoverable state on soft error manifestation. Second, ForEVeR can provide low-cost soft error resilience by utilizing the same recovery hardware to reliably transmit the soft-error affected packets. Finally, to protect the recovery hardware (e.g., checker network) itself against soft errors, simple ECC and duplication based techniques can be leveraged without significant overhead. Overall, with these hardware modifications, ForEVeR can provide soft-error coverage comparable to other state-of-the-art techniques.

## 4. ROUTER CORRECTNESS

A correctly functioning router should ensure that packet's integrity is maintained while each packet is transfered within the router. This can be achieved by guaranteeing that routers do not drop any individual packet's flits and that flit ordering from head to tail is preserved during the transmission in a wormhole fashion. In this section we discuss how to organize the formal verification of the router at design-time. In the case that some aspects of the router cannot be proven correct, we propose runtime hardware to monitor and correct any related functional bug. We discuss our ideas for a fairly complex and generic 3-stage pipelined router that is input-queued and that uses virtual channel (VC) flow control, look-ahead routing and switch speculation. A high-level schematic of this router is shown in Figure 4(a). The datapath components consist of input buffers, channels and crossbar, and are controlled by input VC control (IVC), route computation unit (RC), VC allocator (VA), switch allocator (SA), output VC control (OVC) and buffer manager.

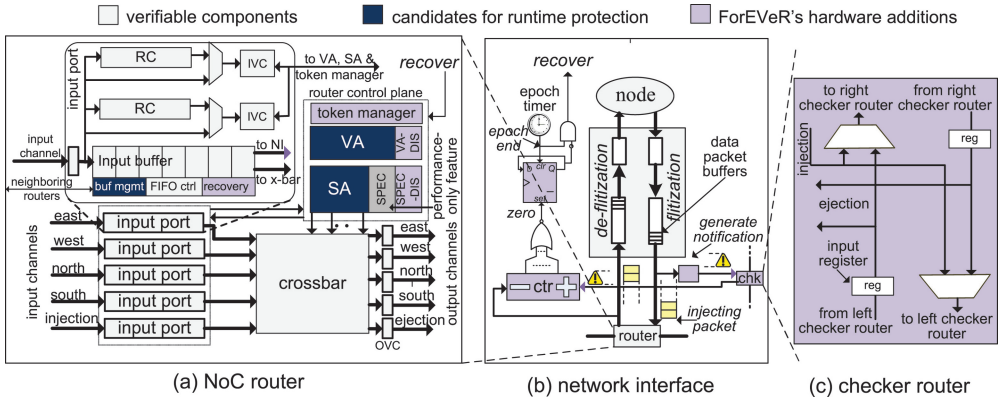



Fig. 4. *ForEVeR's hardware overview.* (a) *Router additions.* VC allocator (VA), switch allocator (SA) and flow control units are monitored by runtime checkers. To implement recovery, each NoC router is augmented with VC and speculation disablers, along with a token manager and a recovery FIFO controller. (b) *Network interface additions.* A counter, timer and zero set register is used to detect undelivered packets at the network-level. (c) *Checker router.* A packet-switched router designed with simple muxes and flow control is used in a ring topology to support the recovery process.

Table I. Organization of Router's Formal Verification

| correctness goal | property to be verified | #sub-prop | time(sec) |
|---|---|---|---|
| *no_packet_drop* (datapath and router activity control) | * incoming valid flit written to buffer | 4 | 90 |
| | * buffer operates as FIFO | 20 | 660 |
| | * all flits are transferred from input to output channel | 17 | 170 |
| *no_packet_create* (control components) | * no flit/packet duplication at IVC | 4 | 30 |
| | * no flit/packet duplication at crossbar | 1 | 10 |
| | * no flit/packet duplication at OVC | 2 | 10 |
| *no_data_corruption* (interactions of concurrent components) | * valid body flits follow valid head flit in order (leaving IVC) | 25 | 1,800 |
| | * valid body flits follow valid head flit in order (leaving crossbar) | 5 | 350 |
| | * valid body flits follow valid head flit in order (leaving OVC) | 5 | 200 |

### 4.1. Formal Verification

The verification process can be efficiently partitioned into three subgoals: i) ensuring that no flit is dropped (*no_packet_drop*), ii) showing that no flit is created or duplicated (*no_packet_create*), and iii) ensuring that packets maintain integrity as they travel through the router (*no_data_corruption*). For the first subgoal, we must verify that all valid flits received at input channels are written into valid buffer entries, that the buffers operate in a FIFO manner, and that each flit, after gaining access to the output channel, moves from input buffer to the output channel in a fixed number of clock cycles (depending on the router pipeline depth). To accomplish the second subgoal, we verify that flits are not duplicated as they travel through the various stages of router pipeline (IVC, crossbar and OVC). We also verify that these stages do not create flits out of thin air. The third subgoal encompasses the behavior of entire packets, rather than individual flits, ensuring that all body flits belonging to any particular packet should follow that packet's head flit in a wormhole order, as the packet traverses the router's datapath. We pursued the formal verification of the router design that we used in our experimental evaluation, using the structure described above and Synopsys Magellan, a commercial formal verification tool. Table I summarizes our subgoals, how many

properties were proven for each of them and how much computation time they required. Properties were described as System Verilog Assertions and verification executed on an Intel Xeon running at 2.27 GHz and equipped with 4GB of memory. For instance, the property *incoming valid flits written to IP buffer*' holds if i) all incoming flits have always a valid VC tag, ii) the corresponding VC buffer has a free slot (no overflow), iii) the flit contents should only be written to the specified slot of the requested VC buffer, and iv) invalid flits should not be written to any VC buffer. Note that we do not need to verify the route computation module, as our network-level detection and recovery scheme handles possible escaped bugs in this module.

### 4.2. Runtime Verification

When not all router-specific properties can be verified, ForEVeR provides a runtime solution to complement the design-time effort and still guarantee router-level correct functionality. In a router microarchitecture, the control components manage the flow of packets and flits through the datapath components: from input channels to output channels via input buffers and crossbar. Typically, the datapath components are fairly simple and can be completely verified at design-time. Verification of the control components presents a greater challenge. Particularly, components that control the interactions between multiple router activities are more likely to be beyond the capability of design-time verification [Foster et al. 2006]. In the context of a router, the interactions between packet flows are handled by route computation (RC), VC allocation (VA) and switch allocation (SA) units. These units rely on information provided by the buffer manager, used to transmit buffer availability information among neighboring routers. Other control units, such as input (IVC) and output (OVC) VC control, operate mostly on local data, and hence can often be formally verified using traditional formal verification tools. We therefore focused on designing generic runtime monitors for VA, SA, and buffer management units (as indicated in Figure 4(a)). Note that the route computation unit does not need to be protected, because its activity is monitored as part of our network-level solution.

*4.2.1. Detection and Recovery. VC and switch allocator.* A design flaw in a VC allocator may give rise to various erroneous conditions, some of which are tolerable, as they either do not violate router correctness rules, or are effectively detected and recovered by the network-level correctness scheme. Assignment of an unreserved but erroneous output VC to an input VC is an example of such an error as, in the worst case, it may only lead to misrouting or deadlock, which can be detected and recovered by our network-level correctness scheme. Starvation is another example that needs no detection or remedy at the router level. Critical errors, that is, errors that threaten data integrity, 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 flit mixing and/or packet/flit loss. Similar to the VC allocator situation, a design flaw in a switch allocator may or may not have an adverse affect on ForEVeR's operation. To monitor VCs and switch allocators at runtime, we propose the use of an Allocation Checker (AC) unit, a simplified version of a unit proposed in Park et al. [2006] for soft error protection. The AC unit is purely combinational and it performs all comparisons within one clock cycle. It simultaneously analyzes the state of VC and switch allocators for duplicate and/or invalid assignments. If an error is flagged, all VC and switch allocations from the previous clock cycle are invalidated. Flits in flight in the crossbar are discarded at the output. To avoid dropping flits during the invalidation/discard operation, an extra flit storage slot per input port is reserved for use during such emergencies. To implement this runtime monitor, VA, SA, and crossbar units are modified to accept invalidation commands from the AC.

*Buffer management.* A design error in buffer management can lead to either buffer underflow or overflow. Input buffers can be easily modified to detect and refuse communication during an underflow, thus not loosing or corrupting any data. On the other hand, a hardware checker is used to detect buffer overflow errors. Additionally, each input port is equipped with two emergency flit storage slots. Upon receiving a flit when the corresponding buffer is full, the communicating routers switch to a NACK-free variant of ACK-NACK flow control, that guarantees freedom from buffer overflows using a simple scheme. The emergency slots are reserved for flits in flight during this event. During this NACK-free flow control operation, a flit awaiting acknowledgement is retransmitted every two cycles (round trip latency of the links). This scheme, though detrimental for performance, is extremely simple and can be implemented with little modification to the baseline buffer management scheme. In addition, the router operates in this simple and verified 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 input port are required and this storage can be implemented as a simple shift register. In addition, the cost of this extra storage is amortized across multiple VC buffers in a single input port.

*4.2.2. Degraded Mode.* When a bug is detected by the 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 counterparts. This mode is equipped with bare-minimum features to support the network-level recovery, initiated immediately after discovering a bug. During network-level recovery, packets stuck within main network routers are drained sequentially over the verified checker network to their intended destinations. Therefore, in the degraded mode, a main network router should be able to advance flits through its pipeline and drain them one-by-one over the checker network. However, packet-level interactions between subsequent routers, such as route computation and VC allocation, are unnecessary during recovery and are disabled to reduce complexity. In addition, advanced "performance only" features, such as switch speculation and prioritizing mechanisms are disabled. Switch allocator and buffer manager must still work properly to drain packets affected by the bug occurrence. To this end, the SA is replaced by a simple spare arbiter that allocates only a single output port to a single input port at each cycle, eliminating concurrent interactions. Similarly, the buffer management scheme is replaced by an acknowledgement-based control discussed earlier, to prevent flit loss. The resulting degraded router has significantly less concurrency, making it amenable to formal verification.

## 5. NETWORK CORRECTNESS

As discussed in Section 3, at the network level we must guarantee that all packets are delivered to their intended destination within a bounded amount of time. Specifically, our network-level solution must detect and recover from design errors that inhibit forward progress in the network (deadlock, livelock and starvation) or cause misrouting of packets. To achieve this, ForEVeR augments the design with a lightweight and verifiable checker network that operates concurrently with the original NoC, providing a reliable fabric to transfer notifications and packets to be recovered. A notification only carries the packet's destination address, and therefore, our checker network is lightweight and simple. It is organized in a ring topology, comprising of single-cycle latency, packet-switched routers (Figure 4(c)). The design and analysis of the checker network is detailed in Section 5.3. Finally, the network interface logic is augmented with notification generation and monitoring hardware to enable concurrent operation of the main and the checker network, as shown in Figure 4(b).

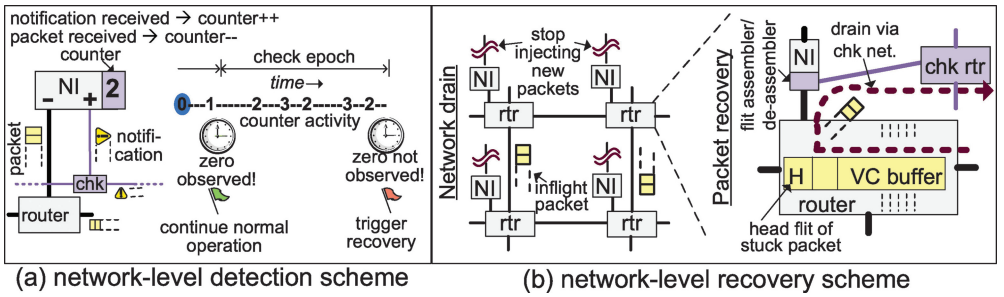



Fig. 5. *ForEVeR network-level (runtime) verification scheme.* (a) *Detection.* The counter at destination NI is incremented (decremented) upon notification (data packet) arrival, and recovery is triggered if zero is not observed during a *check epoch*. (b) *Recovery.* All packet injections are suspended during the network drain phase, while remaining packets are recovered via the checker network during the packet recovery phase.

### 5.1. Detection

During normal operation, each packet transmitted on the primary network generates a corresponding notification over the checker network, directed to the same destination. Each destination maintains a count of outstanding packets expected via the primary network, decrementing the count upon each packet reception. Operation is organized into '*check epochs*', time intervals of fixed length: a distributed detection scheme monitors that, during each *check epoch*, a value zero is observed at least once at each counter. If that is not the case, recovery is initiated, extracting all in-flight packets from the primary network and delivering them reliably through the checker network. Figure 3 shows how the checker network interfaces with the primary network via the network interface units, which also include the logic for detection and recovery initiation.

All design errors preventing forward progress result in packet(s) trapped within the network, consequently, the detection mechanism should be capable of detecting such scenarios. Moreover, it should entail minimal area overhead and design modifications. Our notification message architecture satisfies both these requirements and it allows detection of a bug occurrence because any unaccounted packet at destination will lead to a counter with an always positive value. This reasoning, however, assumes that notifications are always delivered ahead of the counterpart main network packets during normal operation. Otherwise, our detection scheme may exhibit false negatives. We leverage the low latency property of our checker network to consistently deliver notifications before the corresponding main network packets, and keep the false negative rate under check.

Figure 5(a) illustrates the hardware implementation and execution flow of our detection scheme. The detection algorithm increases the counter at the local destination node for each notification received, and decreases it for each packet received. In addition, it stores in a separate register, reset at the beginning of each *check epoch*, whether a zero has been observed. If, at the end of a *check epoch*, any network node has not yet observed a zero, then recovery is initiated. The implementation requires a counter connected to both the primary and the checker network, a timer to track epochs and a zero-observed storage bit. We study in Section 7.2 how the epoch length affects the accuracy of detection. Finally, design errors leading to misrouting of packets are detected by checking the destination node address of the packet requesting ejection against the local node address. In NoC architectures that do not augment packet headers with destination information, ForEVeR incurs an additional overhead of inserting the destination node address in each packet header.

### 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 recovery phase, consisting of a *network drain* step followed by a *packet recovery* step, as illustrated in Figure 5(b). During *network drain*, the network is allowed to operate normally to drain its in-flight packets, while no new packets are injected. If recovery was initiated by a router-level checker, then that router operates in degraded mode during this phase. At the end of this phase, which runs for a fixed time length, recovery terminates if all destinations have received all their outstanding packets. This situation indicates that recovery was triggered by a false detection, which can be caused, for instance, by an always positive counter because of high traffic. Note that false positives only impact the system's performance, but not its correctness.

The subsequent phase, *packet recovery*, recovers all remaining outstanding packets. To this end, a token is circulated through all routers in the NoC via the checker network, and NoC routers can only operate when they hold this token. In addition, all VC allocators are disabled to prevent the processing of new packets. When a router receives the token, it examines all its VC buffers sequentially to find packet headers. If a header is found, the corresponding packet is extracted and transmitted over the checker network, as shown in Figure 5(b). Since key router functionalities are still active in the degraded mode, the packet can be safely delivered to its destination through the checker network. The token circulates through all routers retrieving packets from one router at a time. Retrieving all packets may require repeating the looping of the token through all routers more than once, as some packets may become available for retrieval only after other packets have been extracted. Note that we do not drain a packet when its head flit is not at the front of the buffer upon arrival of the token. Thus, the design complexity of the corresponding FIFO buffer is kept low by utilizing only one read and one write pointer. The hardware components required to provide recovery are shown in Figure 4(a) and include: a *token manager*, a *virtual channel allocation disabler* (VC-DIS) and a *switch speculation disabler* (SPEC-DIS) for each router, to prevent the processing of new packets during recovery.

Due to the limited bandwidth of the checker network, each primary network flit is transmitted as several checker packets. To this end, each main network flit is *disassembled* into multiple checker packets before injection into the checker network. These checker packets are *assembled* back together at the destination nodes. Slight modifications are required to the checker network and small disassembling/reassembling units are added to the network interfaces, to support the recovery operation. However, during recovery, only one router can be transmitting a packet to a single destination, greatly simplifying the disassembling $\rightarrow$ transmission $\rightarrow$ reassembling process.

### 5.3. Checker Network

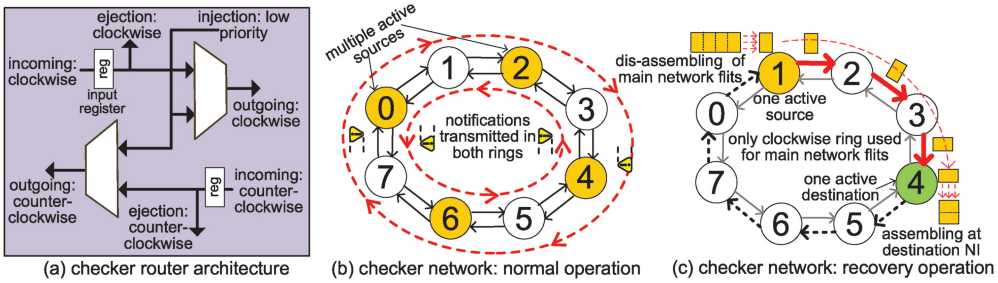
A suitable checker network should have three main properties: i) it should be formally verifiable; ii) it should have a low latency; iii) and finally, it should incur a low area overhead. A formally verifiable network should present a simple router and network architecture and a simple routing algorithm. In contrast, a naïve solution to consistently deliver notifications before their counterpart data packets arrive through the primary network, may require complex router and network designs that are area-intensive and difficult to verify. Fortunately, exploiting the nature of traffic flowing through the checker network (i.e., notifications), we are able to design a network satisfying both these conflicting requirements. Note that these properties are not a strict requirement for the correctness of our solution. However, not meeting them leads to additional costs in area, development time and performance. For instance, a checker network that does

not deliver notifications before their corresponding data packets, may result in an error going undetected over multiple epochs. In such a scenario, error detection latency is affected, but NoC correctness is still guaranteed, as explained in Section 7.2. The checker network, therefore, is tailored to the characteristics of the primary network, so to minimize the occurrence of such cases.

The checker network is used exclusively to transmit notifications to their destinations. The notifications contain no data, only including destination address (6-bits for a 64-node NoC) and a valid bit. Having to transmit only packets of uniform, small sizes presents many advantages, that can be leveraged to design a simple and efficient NoC architecture. First, packet-switched routers can be utilized, without large buffers or many wires between routers. This eliminates the need for complex switching techniques for buffer and/or wire cost amortization, including circuit switching, store-and-forward, virtual-cut through, wormhole and virtual channel switching, which all require a substantial amount of control and book-keeping hardware for proper functioning. Additionally, such simple router design can be implemented to have single-cycle latency, as we discuss below. Second, the small fixed-size packets require no packetization hardware at the network interfaces (NIs) and fully utilize the available bandwidth on packet switched networks (no bandwidth fragmentation). Finally, since the notifications are not stored at destinations, multiple notification can be ejected simultaneously, without requiring any additional buffering.

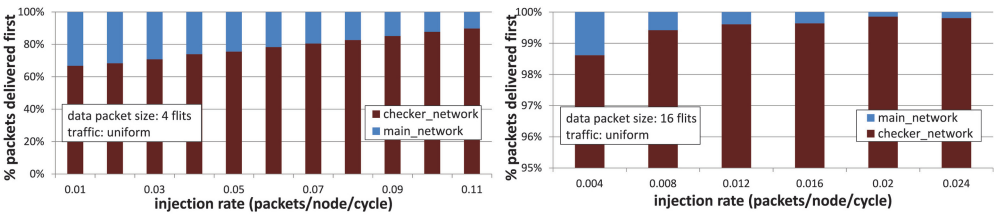
Based on these observations, our checker network should be packet-switched, with channel/buffer allocation and traffic transmission performed at packet granularity. To further simplify the design, we chose a ring topology for the checker network, leveraging a simple, single-cycle latency, packet-switched buffer-less router, based on the solution proposed in [Kim and Kim 2009]. All the nodes in the network are connected in a bidirectional ring. A predetermined allocation ensures that there is no contention for network resources in routing a packet from source to destination; therefore, pipeline registers at the router inputs are sufficient to ensure lossless transmission. To this end, once a packet is injected into the network, it has priority over other packets that are trying to enter the network, and thus the packet is guaranteed to make progress toward the destination. For each packet, a decision is made to inject it in one ring or the one in the opposite direction based on minimal routing distance. Each packet, once in the network, keeps moving forward through the same ring until it is ejected at its destination. Two packets traveling on separate rings may try to eject at the same node in the same cycle, causing contention for the ejection port. This is resolved by providing a separate ejection port for both ring directions at each node. The valid bit from ejection ports connects to the detection counter, while the destination ID is discarded at ejection. The checker router architecture is shown in Figure 6(a), and the checker network normal operation is shown in Figure 6(b).

Ring topologies suffer from higher hop-counts when compared to common topologies like meshes, but in our system this is mitigated by three factors: i) the checker routers have a single-cycle latency, ii) there is no contention within the checker network, and iii) in contrast to notifications, the main network transfers large data packets (e.g., cache lines) that are typically divided up into long wormholes. As a result, the checker network design presented above consistently delivers notifications before their counterpart data packets arrive via the primary network. In exceptional situations, when notifications lag data packets, our scheme may produce a false negative detection result. In general, however, our scheme has a certain amount of tolerance to having a few notifications lagging behind and this rarely leads to false negatives. When it does, false negatives only increase the detection latency and do not affect the correctness of our scheme as we guarantee no loss of flits/packets, and the data would eventually be delivered in an uncorrupted state to the correct destinations upon error detection.




(a) checker router architecture (b) checker network: normal operation (c) checker network: recovery operation

Fig. 6. *ForEVeR's checker network: architecture and operation.* (a) Router Architecture. Single cycle latency and packet switched. Packets are guaranteed to move forward once inside the network. Packets waiting injection have lower priority and there is no contention for ejection. (b) Normal Operation. Notifications are transmitted in both ring directions and multiple sources-destination pairs are active simultaneously. (c) Recovery Operation. Only the clockwise ring is used to transmit disassembled main-network flits, that are then reassembled at the destination's NI. Additionally, only one source can be sending packets to one destination at any time.

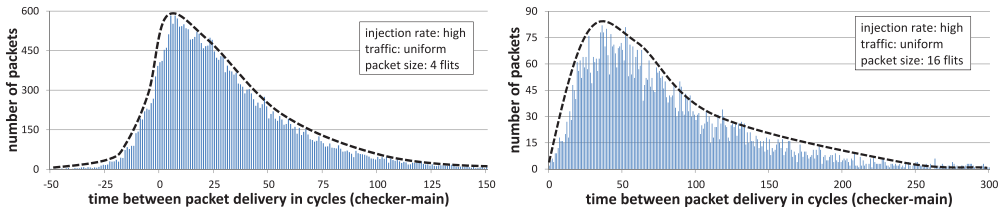



(a) **Small packet size (4 flits).** (b) **Large packet size (16 flits).**

Fig. 7. *Fraction of packets or notifications delivered first.* The checker network delivers almost all notifications before their counterpart primary network packets. The effect is more pronounced at higher injection rates and for larger main network packets.

We ran experiments by injecting uniform traffic with varying packet sizes into the primary network and simultaneously injecting one notification into the checker network for each primary network packet. At the destination, the notifications were matched with corresponding data packets, and the following statistics were logged: i) which network delivered the packet or notification first (main or checker), and ii) the time difference between corresponding deliveries. Figure 7 shows the fraction of packets delivered first by each network for varying injection rates. If the main network uses short packets (Figure 7(a)), the majority of the notifications are delivered before the corresponding main network packets. However, there is a non-negligible fraction of packets (10–35%) for which the main network delivers first. Also, note that due to better congestion management in the ring, a larger fraction of notifications are delivered first when the main network is subjected to heavy traffic. Moreover, when the main network packets are larger (16 flits, Figure 7(b)), the checker network is comparatively less loaded and hence more than 99% of the notifications make it to their destination before the main network packets at any injection rate above 0.008 packets per node per cycle. In summary, the majority of the notifications are delivered ahead of their corresponding network packets under all conditions, with an additional marked improvement on notifications being delivered first when packets are large and the network is under heavy traffic load.

Figure 8 plots the distribution of notification-main packet deliveries for a range of delivery time differences. A positive time-difference indicates that the notifications arrived first. For small main network packets (4 flits, Figure 8(a)), most notifications are delivered 0–50 cycles ahead of the main network packets. Few main network packets




(a) **Small packet size (4 flits).** (b) **Large packet size (16 flits).**

Fig. 8. *Distribution of notification-main packet pairs* over a range of delivery time differences at heavy injection. The checker network delivers almost all notifications before their corresponding network packets.

(11% of total) are delivered before their corresponding notifications, and a negligible number of main network packets (<0.5%) are delivered more than 25 cycles before their notifications. Our detection scheme is tolerant to a few main network packets delivered early, when the notifications follow shortly after (<25 cycles later). In Section 7.2, we describe a technique based on delayed counter update to eliminate false negatives. As can be noted from Figure 8(b), the complete distribution is positive for networks with large packets, with only a negligible number of main network packets (<0.2%) delivered ahead of their notifications.

*Recovery operation.* As mentioned earlier, during recovery, each main network flit is transmitted in multiple segments through the checker network. The network interfaces house the dis-assembling/reassembling logic to support recovery. The checker network recovery operation and logic is greatly simplified because only one router is transmitting packets to a single destination at a time. To this end, the checker network's channels include additional dedicated wires for head and tail indicators that are used during recovery operation. The checker packet with head indicator carries the destination address and reserves an exclusive path between the source and the destination. All intermediate valid packets traversing the ring network are ejected at the same destination until a packet is received with a tail indicator. Moreover, all transmissions on the checker network during recovery occur in the same (clockwise) direction to avoid wormhole overlap of two packets. In contrast to normal operation, the address field is not discarded on ejection, as it contains data in body/tail checker packets. In our evaluation system of 64 nodes, the checker network channel is 8 bits wide, with 6 bits for address and 1 bit each for head and tail indicators. Thus, each 64-bit primary network flit is partitioned into 12 checker networks packets (1 head, 11 body/tail) when transferred on the checker network. We expect design errors to manifest infrequently, and thus this serial transmission scheme, while slow, it should not significantly affect overall system's performance. The operation of the checker network during recovery is illustrated in Figure 6(c).

### 5.4. Verification of ForEVeR's Recovery

All components involved in the detection and recovery processes must be formally verified to guarantee correct functionality. Detection leverages the checker network and the counting and timing logic in the network interfaces. Recovery uses again the checker network and the interface between primary and checker routers for packet draining. To verify the correctness of the checker network, we need to show that it delivers all packets to their intended destination in a bounded amount of time, as discussed in Section 3. We partition this goal into four properties: *injection*, guaranteeing correct injection of packets into the network; *progress*, ensuring packets advance towards their intended destinations; *ejection*, proving timely ejection of packets; and *data_integrity*, ensuring that data remains uncorrupted throughout. We verified all four properties

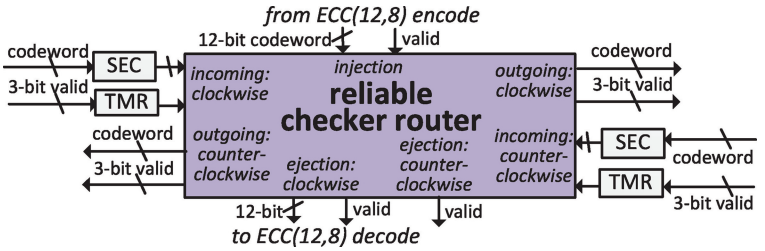



Fig. 9. *Reliable checker network design.* Switch-to-switch single error correction ECC (12-bit codeword, 8-bit data) to protect checker network data and TMR to protect the packet-valid bits.

using our formal verification setup, with the maximum time of 156 seconds spent in verifying the *progress* property.

As discussed in Section 4.2.2, during recovery, the NoC routers operate in a barebone mode with all complex hardware units disabled, thus making the verification task much less challenging. To ensure correct recovery, we have to verify that routers fairly take turns in retrieving valid packets from their respective buffers. To this end, we check the following aspects: i) fairness and exclusivity during extraction (*fairness*) to guarantee that routers take turns in transmitting packets on the checker network. ii) We also verify that complete packets are extracted (*complete_packet*), emptying the buffer completely (*buffer_empty*). We also check that iii) only valid packets are recovered (*valid_packet*). The model checking engine verfied each property in less than 50 seconds.

## 6. PROTECTION AGAINST SOFT ERRORS

ForEVeR includes features to protect against soft errors affecting both the router datapath and control components. Further, it does not require any costly backup data storage and retransmission. Our scheme, however, still relies on end-to-end ECC for fixing one-to-few bit data corruptions. We leverage an end-to-end locality-aware ECC scheme [Shamshiri et al. 2011], that together with clever data-bits/checksum interleaving, can correct multiple bit errors in main network packets. Reliability-oriented modifications to ForEVeR are focused on two aspects: i) enhancing the checker network reliability, and ii) ensuring that integrity of data is always maintained within the main network routers. The second aspect allows us to get rid of the backup storage, and instead drain the erroneous packets via the reliable checker network. Note that from a soft error perspective, integrity of a packet is not affected by few data-bit corruptions, as end-to-end ECC can still successfully reconstruct the original packet. Based on the insight that severe data corruptions are only caused by errors in router control logic, while link and datapath related errors only affect single (or few) bit(s), we can preserve data integrity by only protecting the router control logic.

As previously mentioned, in order for ForEVeR to function properly, the checker network should always deliver unaltered packets to correct destinations. To this end, we augment the checker network with a switch-to-switch single-error-correct (SEC) ECC code. We assume that, due to the small channel width of the checker network, in the worst case, only a single bit can be affected per notification. For our 8-bit wide checker network, a 12-bit wide codeword is required for SEC capability. Additionally, the packet-valid bit is protected by triple-modular redundancy (TMR), to prevent from errors in the ECC output and to avoid transmission of an invalid packet. Thus, the sum total channel width of the reliable checker network is 15 bits (12 codeword, 3 packet-valid TMR). Figure 9 shows the architecture of the reliable checker router. Checker network packets are encoded before injection and are decoded at ejection. Further, at each checker network hop, the packets go through single-error correction phase, before

they are written to the checker router register. Finally, the packet-valid bit is protected with TMR. Note that we are still able to maintain the checker network simplicity, and the reliable design also supports our correctness goals.

As we do not backup clean packet copies at network ends like the retransmission scheme [Murali et al. 2005], we must guarantee that the main network never jeopardizes packet integrity. Remember from Section 4, that we augmented our routers with runtime monitors to prevent them from entering an erroneous state. These checkers are very similar to the ones used in Park et al. [2006] and can also be leveraged to detect anomalies due to soft errors. Note that we detect the anomalous behavior before the network goes into an unrecoverable state (for instance, by dropping a flit) and prevent any data loss/corruption by provisioning only a little emergency storage, as explained in Section 4. Although, majority of the soft error bugs can be overcome by restarting the router operation after soft-error manifestation, we opt for network-wide recovery (Section 5.2) initiation at each error detection. This is because our hardware monitors cannot diagnose the source of error (design bug or soft error), and thus our scheme sticks to a uniform measure in avoiding any unrecoverable state, that is, trigger a network-wide recovery. It should be noted that in rare conditions, soft errors can also lead to forward progress bugs, such as deadlock. Fortunately, our end-to-end counter-based detection scheme (Section 5.1) detects the bug in such a case. During recovery, all main network packets are delivered reliably over the checker network to their final destinations. Since our checker network is protected against soft faults, the data finally delivered is always error-free. The performance and area impact of our reliability scheme is evaluated in Sections 7.3 and 7.4, respectively.

## 7. EXPERIMENTAL RESULTS

We evaluated ForEVeR by modeling a NoC system in Verilog HDL, as well as a cycle-accurate C++ simulator, both based on Dally and Towles [2003]. The baseline system is an 8×8 XY-routed mesh network, routers have 2 VCs and 8-entry buffers per VC, similar to the router design described in Section 4. In addition, the NoC is augmented with a checker network and with the detection and recovery capabilities described in the previous sections. The Verilog implementation was used to formally verify the NoC routers and the recovery components. To this end, we specified properties as System Verilog Assertions and verified them with Synopsys Magellan, a commercial formal verification tool. ForEVeR's area overhead was estimated using synthesis results from Synopsys Design Compiler targeting the Artisan 45nm library. The link area for both the main and checker network was estimated using tile dimensions published for an experimental chip from Intel [Vangal et al. 2008]. We also evaluated the power overhead of our design by modeling both main and checker routers in Orion2.0 [Kahng et al. 2009]. The C++ simulator was used to assess the accuracy of the network-level detection scheme and to evaluate the performance impact of the recovery process. This process was triggered by functional bugs that we inserted in the baseline model to evaluate our solution. The framework was analyzed with two different types of workloads: uniform random traffic, as well as applications from the PARSEC suite [Bienia et al. 2008].

### 7.1. ForEVeR Operation

To analyze ForEVeR's performance impact and its ability to recover from various types of design errors, we injected 9 different design bugs into the C++ implementation of ForEVeR, as described in Table II. Bugs 1–6 are errors that inhibit forward progress, bugs 7–8 are misrouting errors, whereas bug 9 is an error that affects router operation while it is servicing a packet. We ran PARSEC workloads while triggering one distinct bug during each entire execution and varying the trigger time (5 trigger points,

Table II. Functional Bugs Injected in ForEVeR and Average Packet Recovery Time

| Bug name | Bug description | recovery time |
|---|---|---|
| deadlock | some packets deadlocked in the network | 4,821 cycles |
| livelock | some packets in a livelock cycle | 3,084 cycles |
| VA_vc_strv | input VC never granted an output VC | 2,827 cycles |
| VA_port_strv | no input VC in a port granted output VC | 3,055 cycles |
| SW_vc_strv | one input VC never granted switch access | 2,123 cycles |
| SW_port_strv | no input VC in a port granted switch access | 2,490 cycles |
| misroute1 | one packet routed to a random destination | 1,724 cycles |
| misroute2 | two packets routed to random destinations | 1,810 cycles |
| router_bug | hardware monitors in routers detect a bug | 1,764 cycles |
| average | | 2,633 cycles |

10,000 cycles apart), the location of bug injection (10 random locations) and packet size (4, 6 and 8 flits). ForEVeR was able to detect all design errors with no false positives or negatives and correctly recover from them, executing all workloads to completion and delivering all packets correctly to their destinations. Each recovery entailed a performance overhead, due to *network drain* and *packet recovery* activities. Once an error is detected, recovery is triggered by initiating the *network drain* phase, where the primary NoC was allowed to drain for a fixed period of 500 cycles, a parametric value that we set conservatively by simulating the draining of a congested network. If the network was not drained completely within this time interval due to a bug, *packet recovery* is initiated, incurring an additional performance impact but still guaranteeing correct operation. Table II reports the additional average *packet recovery* time incurred for each bug, averaged over all benchmarks, packet sizes, activation times and locations. Note that, apart from forward progress errors that are detected at the network level, routing errors are caught at incorrect destinations, while errors affecting router operation are uncovered immediately by the hardware monitors.

On average, ForEVeR spends approximately 2,633 cycles in *packet recovery* for each bug occurrence. This value is primarily affected by the number of packets that must be recovered; thus, bugs affecting a large portion of the network, such as an entire port (*VA_port_strv*), take more time to recover than bugs that influence smaller portions, such as only one VC (*VA_vc_strv*). Similarly, *deadlock* errors that may affect many packets, require the largest recovery time. We observed the worst case recovery time of 30K cycles across all our simulation runs, including all packet sizes and both uniform and application traffic. A key aspect of ForEVeR design is that it incurs no overhead during normal operation, spending time in recovery only on bug manifestation. Therefore, it can afford a longer recovery time as design bugs manifest infrequently.

Bugs that typically escape into production hardware are extremely rare corner-case situations buried deep in the design state space, that were not uncovered by extensive pre-silicon and post-silicon validation efforts. Hence, it is safe to assume that these bugs are extremely infrequent as they have escaped months of verification efforts. Therefore, even though more than 100 bugs were discovered in the latest Intel chips (Figure 1) after production, released bug patches (mostly software based) do not lead to significant slow down in overall performance. In perspective, ForEVeR's recovery penalty of 2.6K cycles on average is insignificant even for a unrealistic bug rate of one error every 5 minutes. For a 1 GHz NoC, exhibiting an error rate of one error every 5 minutes, this translates to a negligible performance penalty, less than one hundred millionth ($10^{-8}$).

To closely study the relationship between recovery time and number of flits recovered via the checker network, we injected a varying number of packets in the NoC, and prevented them from ejection at the network interfaces. The network-level detection

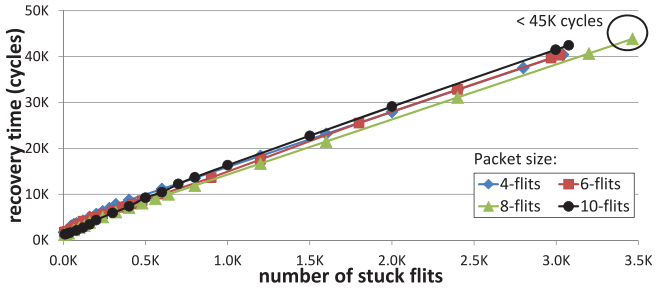



Fig. 10. *ForEVeR's packet recovery time.* ForEVeR's recovery overhead increases almost linearly with the number of flits stuck in the primary NoC that must be transmitted reliably through the checker network. Worst case packet recovery time of 45K cycles is observed in this limit study.

scheme flags an error after the second epoch due to unaccounted primary network packets at destinations, thus triggering a network recovery. Concurrently, we noted the time required to drain all stuck packets through the checker network. Figure 10 plots our results for varying packet sizes, reporting *packet recovery* time vs. number of extracted flits. As seen from the figure, *packet recovery* time varies almost linearly with the number of stuck flits, requiring less than 45K cycles, even in the worst case. Note that in this artificial scenario, we are intentionally jamming the main network with many more flits than usual network occupancy at any instant. Thus, this serves as a limit study, and in practice ForEVeR's *packet recovery* time is limited by 30K cycles, as seen by our results on recovery from desgin bugs.

### 7.2. Network-level Detection Accuracy

ForEVeR's runtime performance overhead is affected by the accuracy of its detection.

*False positives.* False positives occur when an unnecessary recovery is triggered in absence of a bug occurrence, and they are due to inaccuracies in the runtime monitors. The corresponding recovery consists of the execution of a *network drain* phase, where all in-flight packets are delivered to their correct destination nodes. At that point no packets remain in flight, thus there is no packet recovery phase. Note that, a false positive in the detection mechanism does not affect the network's correctness but only its performance. The false positive rate of the detection scheme depends on the duration of the *check epoch*, relative to traffic conditions. Note that false positives are triggered when the destination counter is non-zero for an entire *check epoch*; hence a heavily loaded network will trigger more false recoveries as unaccounted notifications accumulate at destinations while their corresponding packets are being delayed due to congestion in the network. Intuitively, a longer *check epoch* would reduce the false positive rate by allowing more time for packets to reach their destinations. Figure 11(a) shows the decrease in false positive rate with increasing *check epoch* size. The false positive rate drops to a negligible value (less than 0.1%) beyond a certain *check epoch* size ($Epoch_{min}$), whose value depends on network load. Indeed, a heavily loaded network exhibits a higher false positive rate than a moderately loaded network, and hence a heavily loaded network requires a longer $Epoch_{min}$ to practically eliminate all false positives. Extensive simulations indicate that $Epoch_{min}$ rises to intolerable values only when a network is operated at loads well past its saturation. However, NoC workloads are characterized by the self-throttling nature of the applications, which prevents them from operating past saturation loads [Nychis et al. 2010].

*False negatives.* False negatives might cause an error to go undetected for a few epochs. But, since we guarantee no loss of flits/packets, the data would eventually

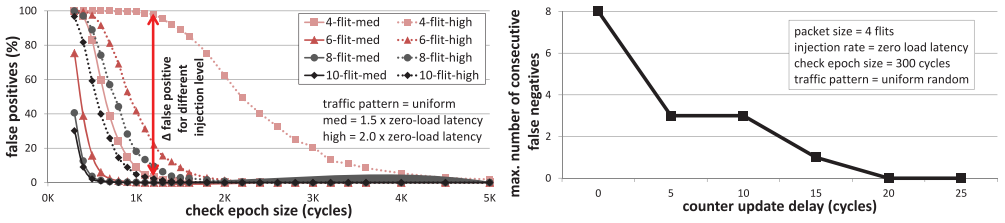



Fig. 11. *Analysis of false positive and negative rates with uniform random traffic.* (a) *False positive rate of ForEVeR vs. check epoch size.* The figure plots the false positive rate vs. *check epoch* size, for various packet sizes. The false positive rate drops rapidly with longer *check epochs* and decreasing network load (difference shown with *red mark*) . (b) *False negative rate of ForEVeR vs. primary network offset* under low latency uniform traffic with a packet size of 4 flits and *check epoch* length of 300 cycles. False negatives decrease when the delay from when monitor counters receive a notification to when they update the counter value increases.

be delivered in an uncorrupted state to the correct destinations upon error detection. Hence, such a scenario will only increase detection latency without affecting correctness. Consider the case when a packet is stuck within the NoC (forward progress error). Ideally, its destination counter will not observe a zero value for the entire epoch and hence recovery would be triggered. However, in a realistic scenario, some other network packets may be delivered to the same destination ahead of their notifications, causing the counter to record a zero value. In such a case, recovery will not be triggered even when the network is in an erroneous state. Therefore, to avoid false negatives in the detection scheme altogether, the checker network should be constrained to deliver all notifications before their corresponding data packets arrive via the primary network. If this cannot be guaranteed for a baseline checker network design, it is still possible to satisfy the constraint by considering design alternatives, such as bundling together multiple notifications before transmission, or using multiple checker networks, etc. As was noted in Section 5.3, our baseline checker network almost always delivers notifications ahead of data packets, except for very low traffic load situations, where primary network packets take shorter routes through the primary NoC, while notifications travel longer routes in the ring-based checker network. To counter these cases, the decrementing of the monitor counters on main packet arrival can be delayed by an amount determined by the maximum latency difference between primary and checker networks at zero load (we call this value *counter_update_delay*). In other words, on a main network packet arrival, the counter value is decremented *counter_update_delay* cycles later, instead of immediately. This artificially gives the checker network packets *counter_update_delay* additional cycles to catch up with the main network packets that may arrive ahead in time. We ran low traffic load simulations using uniform traffic with a small packet size (4 flits) and a short *check epoch* of 300 cycles. With this setup the primary network is only lightly loaded, and hence, has a greater chance of creating false negatives. Figure 11(b) plots the maximum consecutive false negatives observed over 10 different seeds for different *counter_update_delay* values. The reported false negative rate falls quickly and is completely eliminated at a delay of $>20$ cycles. Note that the maximum delivery time difference between primary and checker network at zero-load was 18 cycles in our simulations.

*Optimal epoch length.* To calibrate the *check epoch* parameter, we ran rigorous simulations using both uniform random traffic and PARSEC benchmarks. After operating ForEVeR normally for a preset length of time, a random data packet is dropped to emulate the impact of an error in the primary network; we then calculate the false positive and negative rate for a range of *check epochs*.

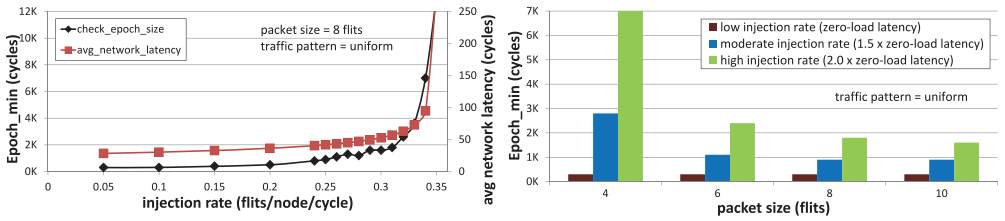



Fig. 12. *Impact of traffic load and packet size on* $Epoch_{min}$. a) $Epoch_{min}$ and network latency with increasing traffic load: $Epoch_{min}$ length is within tolerable limits for all but deeply saturated networks. b) $Epoch_{min}$ variation with packet size at different network loads: $Epoch_{min}$ decreases with larger packets.

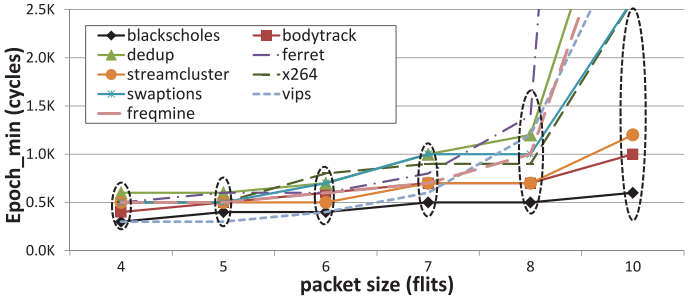



Fig. 13. *Epoch*$_{min}$ *for varying packet sizes with PARSEC benchmarks.* $Epoch_{min}$ is within 1,500 cycles for packets up to 8 flits long over all PARSEC benchmarks studied (corresponding to an average network latency of 800 cycles).

Figure 12(a) plots $Epoch_{min}$ (necessary to minimize the false positive rate) and the average network latency as network load is increased, under uniform network traffic. $Epoch_{min}$ exhibits a slow increase with rising injection rate up to network saturation, and a steep rise afterwards. From the plot, a worst case $Epoch_{min}$ 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 12(b) presents a similar study plotting $Epoch_{min}$ at low, moderate and high injection rates for four different packet sizes. The plot indicates that $Epoch_{min}$ decreases with increasing packet size. For similar loads, a network using larger packet sizes has fewer in-flight packets causing fewer notifications to accumulate at destinations, and hence lower $Epoch_{min}$ values.

PARSEC benchmark traces for evaluation of our detection and recovery scheme were extracted from a 64 core CMP system, using our baseline NoC using 4-flit data packets and running PARSEC workloads. 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, effectively increasing the packet length and network load. Using the same traces, we used these longer data packets (due to decreased channel width) to unrealistically stress the network during simulation. It should be noted that such high load scenarios (average network latency up to 1,600 cycles) should never arise in practice because of the self-throttling nature of the applications. Figure 13 plots $Epoch_{min}$ with different packet sizes for the PARSEC benchmarks, showing that, in practice, a zero false positives rate can be achieved with small *check epoch* length.

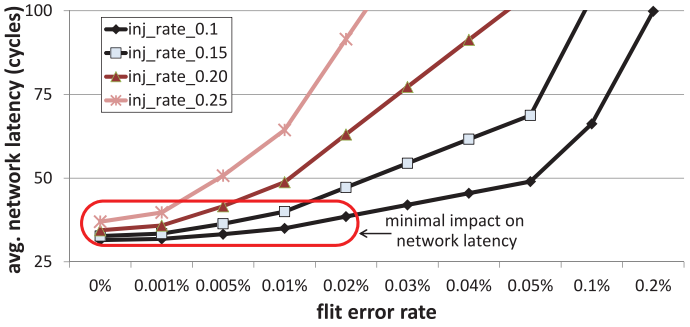



Fig. 14. *Average network latency vs. flit error rate* for varying injection rates. Minimal latency impact is observed up to the flit error rate of 0.01%. More flits are transferred via the reliable checker network on recovery initiation for high injection runs, and hence the latency degradation increases with injection rate.

### 7.3. Soft Error Protection

The performance of soft error recovery schemes depends on the soft-error rate (SER). The greater the SER, the larger the performance penalty on network operation. For NoC specific experiments, average network latency with varying flit error rates, is a well accepted metric to judge the quality of a recovery solution [Murali et al. 2005]. Flit error rate is defined as the probability of one or more errors occurring in a flit. Errors in a flit may be caused, either by soft error-induced malfunction of router control logic, or data bit-flips in router datapath/links. Assuming equal logic, memory and link SER, we make a simplifying estimate that the probability of a flit error being caused by control logic-malfunction is proportional to the silicon area percentage dedicated to control components. We conducted an area analysis of the components of our baseline router, and observed that approximately 14% of the area is dedicated to control components, while the rest is attributed to datapath and links. Further, since we rely on end-to-end correction of data corruptions occurring while traversing the links or datapath components, there is no network performance penalty for such errors. However, if an error in control logic is detected by router-level hardware monitors or the notification counting scheme, ForEVeR freezes the router pipeline and drains all the packets residing in the main network via the checker network. The corresponding recovery time is significant, and hence performance suffers on frequent recovery triggers.

We ran simulations injecting errors in flits with varying probability. The simulator classified the injected flit errors as *control-induced*, with 14% probability. The remainder were tagged as performance-neutral *datapath-induced* errors, as they are correctable by end-to-end ECC. However, the *control-induced* errors triggered a network-wide recovery, causing a performance hit. Figure 14 shows the average network latency with increasing flit error rate, for varying loads of uniform random traffic. Naturally, with the increasing flit error rate, the average network latency suffered. However, the effect on average network latency was tolerable (11% worse for 0.1 flits/cycle/node injection rate) up to the flit error rate of 0.01%, beyond which the latency degradation was quick. We also observed that the latency degradation was more drastic for higher injection rates. This is because more main network flits are transferred via the reliable checker network on recovery initiation for high injection runs. Finally, unlike retransmission schemes based on sending acknowledgement messages during normal operation [Murali et al. 2005; Aisopos and Peh 2011], our scheme does not introduce any extra traffic into the main network, and hence does not incur a performance hit in absence of errors. We conclude that ForEVeR's soft-error protection is suitable for networks operating at high load and expecting a flit error rate of less than 0.001%

Table III. ForEVeR Area and Power Overhead

(a) area: router level correctness

| design | area ($\mu m^2$) | % | NAND2 |
|---|---|---|---|
| baseline rtr (logic+link) | 138K (78K+60K) | 100 | 144K |
| flow ctrl & storage | 6,071 | 4.4 | 6.33K |
| VA & SA monitor | 1,053 | 0.8 | 1.10K |
| **overhead** | **7,124** | **5.2** | **7.44K** |

(b) area: network level correctness (per router)

| design | area ($\mu m^2$) | % | NAND2 |
|---|---|---|---|
| token mgr & rec. sup. | 1,300 | 1.0 | 1.35K |
| NI additions | 1,550 | 1.1 | 1.62K |
| chk rtr (logic+link) | 3.9K (0.3K+3.6K) | 2.8 | 4.07K |
| **overhead** | **6,748** | **4.9** | **7.05K** |

(c) area: soft error protection

| design | area ($\mu m^2$) | % | NAND2 |
|---|---|---|---|
| rel. chk rtr (logic+link) | 7K (1K+6K) | 5.2 | 7.39K |
| reused hardware | 8,974 | 6.5 | 9.37K |
| **overhead** | **16,049** | **11.6** | **16.76K** |

(d) power: checker network overhead

| design | power (mW) | % |
|---|---|---|
| baseline rtr (logic+link) | 219 (190+29) | 100 |
| chk rtr (logic+link) | 5.2 (3.0+2.2) | **2.4** |
| rel. chk rtr (logic+link) | 8.3 (4.6+3.7) | **3.9** |

(worst-case latency degradation of 7%), while ForEVeR can sustain a flit error rate of up to 0.01% for networks operating at low load.

### 7.4. Area and Power Results

A central goal in designing ForEVeR is to keep silicon area and power dissipation at a minimum. The amount of hardware required to implement router-level correctness varies with the designer's ability to verify different router components, as formally verified functionalities need no protection at runtime. Thus, we present the area overhead for network-level and router-level correctness separately. On the other hand, since ForEVeR's power dissipation is dominated by switching activity in the checker network, we only report power corresponding to the checker network. As a reference, we provide the number of NAND2-equivalent gates for each component, in addition to exact area and percentage overheads. Table III(b) reports additions for network-level correctness, summing up to a 4.9% area overhead over the primary network router. The overhead is due to additions in each router, contributing 1.0%, and to each network interface and checker router, which, combined, are responsible for the remaining 3.9%. Note that the link length between neighboring main network routers (and checker network routers) is estimated based on tile dimensions published in Vangal et al. [2008] and scaled to 45nm technology. Subsequently link area and power estimates were obtained from Orion2.0 [Kahng et al. 2009], assuming a switching activity factor of 0.5.

Table III(a) reports the overhead for ForEVeR's router-level hardware monitors and reconfiguration hardware, accounting for 5.2% additional area over the baseline router. Flow control reconfiguration and extra storage required to avoid dropped flits costs 4.4%, whereas VA and SA checkers, along with a spare arbiter (Section 4.2.2) and other reconfiguration support, result in 0.8% overhead. In our framework, we were able to formally verify the baseline router completely, and hence we only incurred the network-level area cost (4.9%).

We also modeled ForEVeR's soft error protection hardware in Verilog and synthesized the design to estimate the area overhead. Soft error protection additions to baseline ForEVeR were detailed in Section 6. Table III(c) shows the area overhead of soft-error specific components, as well as the comparison to baseline ForEVeR. The reliable checker router costs 5.2% of the baseline main network router area, in contrast to 2.8% for the baseline checker router. The additional area is predominantly attributed to the wiring overhead incurred in transmitting the codeword and TMR valid bits. Note that the reported checker router area accounts for the switch-to-switch ECC protection. The other overheads (6.5%) are shared between reliable and baseline ForEVeR designs. In summary, reliable ForEVeR leads to an area overhead of 11.6%, in contrast to 4.9%

Table IV. Comparison of ForEVeR Against Purely Runtime Solutions

| | type of design errors protected | | | area overhead | | performance overhead | | SEU protection | |
|---|---|---|---|---|---|---|---|---|---|
| solution | forward progress | duplicate/ drop pkt | correctness guarantee | storage buffers | link wires | normal operation | recovery operation | data path | control path |
| ForEVeR | yes | yes | yes | none | low | no impact | short | yes | yes |
| SafeNoC | yes | no | no | high | moderate | no impact | intractable | no | no |
| ACK-Ret | no | yes | no | very high | none | high impact | short | yes | yes |
| Park-06 | no | no | no | very high | none | low impact | low impact | yes | yes |

for network-level correctness in baseline ForEVeR. Note that an additional 5.2% area overhead is incurred for runtime router-level correctness if main network routers are not fully verified in baseline ForEVeR, making reliable ForEVeR only slightly more expensive than baseline ForEVeR (11.6% vs. 10.1%). The reported area numbers do not account for end-to-end ECC in the main network, assuming that the main network is already equipped with ECC units to overcome link cross-talk errors.

From the power analysis reported in Table III(d), we observe that the baseline checker router dissipates 5.2mW power on average, while the reliable checker router dissipates 8.3mW power on average. Thus, the checker router accounts for only 2.4% (3.9% for the reliable version) of the main network routers' power dissipation. We expect this overhead figure to be an order of magnitude less than other runtime techniques that employ large buffers [Murali et al. 2005; Park et al. 2006] or high-bandwidth secondary networks [Abdel-Khalek et al. 2011]. ForEVeR leverages formally verified components within the router to recover from design errors; thus keeping area and power overhead low when compared to purely runtime verification techniques [Murali et al. 2005; Abdel-Khalek et al. 2011]. Without leveraging the design-time verification effort, we would require much more additional hardware and complexity to provide similar functional correctness guarantees. In addition, ForEVeR's soft-error protection scheme mostly reuses the hardware required for runtime verification, to keep the area and power overhead at check.

### 7.5. Comparison against Purely Runtime Solutions

Table IV outlines the qualitative comparison of ForEVeR with other pure-runtime schemes. The comparison is with the following runtime solutions: i) SafeNoC [Abdel-Khalek et al. 2011], which uses a dedicated network to transmit packet checksums to detect and recover from design errors, ii) ACK-Retransmission [Murali et al. 2005], which maintains a fresh copy of injected data until it receives a positive acknowledgement from the destination, and iii) Park-06 [Park et al. 2006], which utilizes various microarchitectural modifications to overcome soft errors both in router datapath and control-logic. We compare the four schemes on four aspects: protection against design errors, area overhead, power overhead and resilience against soft errors. Only ForEVeR can guarantee complete correctness, while others fail to overcome either forward progress bugs or drop/duplicate packets or both. ACK-Retransmission, for instance, cannot recover from bugs like deadlock and livelock, and even for other bugs it uses the same untrusted network to retransmit data upon error detection, possibly incurring the error again and again. Therefore, the delivery of the retransmitted data in ACK-retransmitted cannot be guaranteed.

ForEVeR does not need to buffer main network packets to support its detection and recovery operation. However, storage of notification packets at injection ports is required in our checker network architecture. The notifications only store the destination address (6-bits), and hence are considerably small as compared to main network packets. In our simulations, the worst-case storage required at checker network injec-

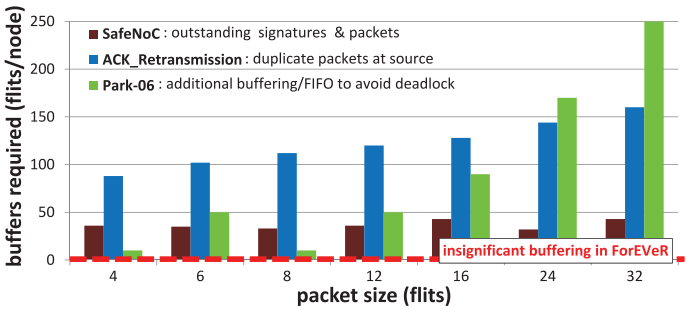



Fig. 15. *Storage requirement*. ForEVeR requires minimal storage only for notifications waiting injection. SafeNoC stores signatures and packets until their counterpart arrives. ACK-Retransmission maintains a backup copy, while Park-06 requires escape buffers at each router FIFO to break deadlock cycles.

tion port was 11 notifications when operated beyond saturation. As on-chip network do no operate beyond saturation [Nychis et al. 2010], this adds up to only 9 bytes of storage for the worst case, almost the same as one main network flit of 64 bits. Thus, the amount of notification storage required is insignificant.

In contrast, SafeNoC utilizes buffers at destination nodes to store checksums and packets waiting for their counterparts, while ACK-Retransmission maintains large source node buffers for all in-flight packets. Finally, Park-06 [Park et al. 2006], requires additional buffering at each router FIFO to overcome deadlocks caused by soft errors, such that the total buffer size is large enough to accommodate the remaining flits of a packet allocated to the FIFO and still have one empty slot. Naturally, the storage requirement grows with larger packets. Moreover, this additional buffering is required at each VC for each input port. We ran stress tests on the NoC using uniform traffic to calculate the maximum buffering required over time for the SafeNoC and ACK-Retransmission solution. For SafeNoC, we observed the maximum number of outstanding checksums and data packets at any time, and for ACK-Retransmission we noted the maximum number of data packets in-flight. Buffer requirements for Park-06 can be calculated using the analytical equations provided in Park et al. [2006]. Figure 15 shows the results of our study and plots the worst case buffering required at each node. Note that ForEVeR requires no additional buffers, while all others require substantial buffer space for proper operation. The buffer requirements for ACK-Retransmission and Park-06 grow substantially for large packet sizes, while SafeNoC requires buffering for ∼40 flits/node independent of packet size. Note that provisioning for buffers to store 50–100 flits at each node can be prohibitively expensive for a constrained NoC environment. For instance, for data packets of 16 flits, the buffers in SafeNoC, ACK-Retransmission and Park-06 alone result in 12.5%, 37% and 26% area overhead over our baseline NoC, respectively. ForEVeR, however, does use some additional wires on a separate checker network, but since the transmitted notifications contain no data, we can design an area-efficient checker network, as described in Section 5.3. In contrast, SafeNoC uses a similar checker network to transfer larger (16–32 bit) checksum packets, while the ACK-Retransmission and Park-06 solutions do not use a separate checker network to overcome errors.

On the performance front, neither ForEVeR nor SafeNoC has any impact during normal error-free NoC operation, whereas ACK-Retransmission and Park-06 create additional traffic due to end-to-end and switch-to-switch acknowledgements, respectively. During recovery, all techniques other than SafeNoC can quickly recover from anomalous behavior. SafeNoC runs a packet reconstruction algorithm in software, that can be exponential in the number of flits to be recovered and reassembled in the worst

case. Finally, ACK-Retransmission and Park-06 can provide the best protection against soft errors, while SafeNoC has no mechanism to overcome soft errors. In contrast, as demonstrated in Section 6, ForEVeR with a few modifications can be leveraged to overcome all kinds of soft errors. In summary, the low area and performance overhead, combined with the ability to protect against both design and soft errors, makes ForEVeR the most complete runtime solution.

## 8. GENERALIZING FOREVER

In this section, we discuss how ForEVeR's approach of complementary verification can be generalized to a number of current and future NoC and router designs.

### 8.1. Other NoC Designs

Both network-level detection and recovery schemes of ForEVeR can be generalized to any NoC design/architecture. At the most basic level, ForEVeR's network-level verification scheme checks if packets are delivered in time to the correct destinations via the main network: the defining property of any correctly functioning network. Further, ForEVeR's network-level correctness hardware is independent and mostly decoupled from the NoC design. Therefore, ForEVeR is agnostic to NoC topology, and the routing and congestion management schemes employed, as long as the checker network, used both during detection and recovery, can adapt to consistently deliver notifications ahead of main network packets. To this end, we leverage the observation that the checker network is simply just wires, muxes and registers with trivial control logic. A recent work [Krishna et al. 2013] reports that wires can carry signals up to 11 hops within a single main router clock cycle. The main router cycle period is usually determined by the complex allocation step and causes this imbalance in the router pipeline. Hence, if need be, the checker network without any complex control logic, can be clocked at a higher rate as compared to the main network router. In our evaluations, we used a high-performance main network with VCs, speculation and prioritization, and our checker network, clocked at the same frequency as the main network, sufficed to deliver notifications as required. Therefore, we anticipate that the basic checker network design would be sufficient for most NoC designs. However, if its performance lags behind, it can be tuned to the needs of the main network by supporting wider channels and multiple simultaneous transmission.

### 8.2. Other Router Designs

ForEVeR's router-level correctness ensures that the integrity of main network packets is maintained throughout the transfer. The *no_packet_drop*, *no_packet_create*, and *no_data_corruption* properties, proven over individual routers, are sufficient to ensure packet integrity. It should be noted that most hard-to-find bugs stall forward progress by causing starvation, deadlocks, etc.: bugs such as these are easily detected and corrected by our network-level recovery scheme. Therefore, formal verification alone, is adequate to prove the data integrity properties in most cases. For example, the formal verification plan developed in accordance to the guidelines of Section 4, was successful to verify our fairly complex baseline router. Although our baseline router implementation is completely formally verified, we further designed generalized hardware monitors to be able to extend ForEVeR's detection scheme to more complex router designs that may be outside the scope of formal verification. These hardware monitors for router-level detection along with some recovery hardware, provide protection to router components that handle complex interactions and the flow of data, such as the resource allocation and buffer management units. They also enable architects to deploy routers with aggressive and complex performance features that are not guaranteed to

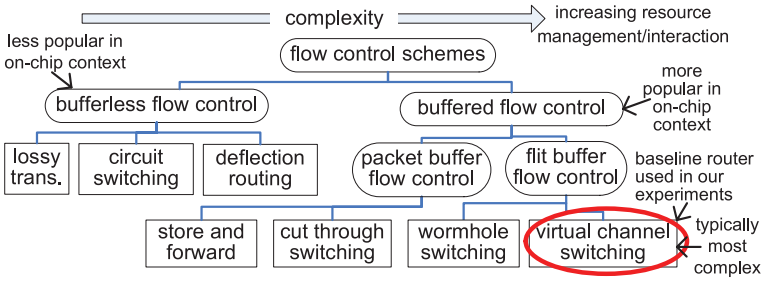



Fig. 16. *Mainstream router microarchitectures.* Generally, complexity increases moving left-to-right in the chart. We formally verified a VC router and designed runtime monitors to generalize to more complex router organizations.

be completely correct. Be it via formal verification or runtime monitors, our verification plan focuses on the movement of data through the datapath components, making sure data integrity is maintained at each move.

Data integrity, the objective of router-level verification, is harmed only when a datapath resource is assigned incorrectly to moving data. Arbiters and allocators, working on the information provided by the flow control units, manage the resources within the network. If data integrity properties are not fully verified via formal verification, arbiter/allocator decisions that can potentially jeopardize data integrity, are detected by the generic Allocation Checker (AC) unit, described in Section 4.2.1. If an error is detected, the allocation/arbitration decision is invalidated. However, due to the pipelined nature of routers, this might lead to loss of data. Emergency buffer space is reserved to avoid data loss due to invalidated control decisions. Based on the router pipeline, few emergency buffers suffice to ensure no data loss as recovery is initiated immediately after error detection.

ForEVeR can be generalized to all mainstream router designs, as they have a similar underlying structure, with control components managing the flow of data through the data-path components (channels, buffers, and crossbars). Router microarchitectures differ from each other in the manner they control the flow of data through each datapath component. Figure 16 shows most mainstream router microarchitecture classes and the associated complexity trends. Note that the baseline router used as a case-study throughout this paper, is typically the most complex. In this section, we will consider flow control options over each datapath component and will show that ForEVeR is equally applicable to all design choices. We try to cover a broad class of router designs in this study:

1. *Channels.* Allocation of channels can be done either at packet level, as in wormhole flow control or at a flit level, as in virtual channel flow control. Typically, verification of flit level allocation schemes is more challenging than packet level allocation schemes, as it increases the interactions between multiple data transmissions. We designed a successful formal verification plan for flit-level channel allocation and alternatively designed an AC unit to avoid over-subscription of channels. For channels allocated at packet level, an easier formal verification plan or a simpler hardware monitor would suffice to avoid over-subscription. Note that, for our baseline design, checking the validity of SA automatically ensures that at most one flit will travel via the output channel in any cycle. Other designs are also possible (output queued routers with VCs, or separate crossbar ports for VCs), where a separate arbiter controls the access to the channel. Fortunately, our generic AC unit can be used to check for over-subscription of the channel by this arbiter too.

2. *Buffers.* Similar to channels, correctness of buffer management schemes is easier to ensure for microarchitectures with packet-level flow control as compared to flit-level flow control. In our experiments, we again verified a flit-level buffer management policy, which is expected to be more challenging to verify than packet-level buffer management policy. Additionally, all mainstream buffer management policies (credit-based, ON-OFF, ACK-NACK) can be designed to fall-back to a safe interrouter transmission scheme described in Section 4.2.1, on error detection.

3. *Crossbar.* Access to the crossbar is exclusively managed by the Switch Allocation (SA) unit. We were able to formally verify that allocations issued by SA preserve data integrity, and additionally we designed an AC unit to check for violations to be able to generalize to SA designs beyond the scope of formal verification. As an example, the SA unit for a crossbar with a dedicated port for each input VC, manages increased request-grant combinations, and hence is more complex. If such a design is beyond the scope of formal verification, AC unit can detect violations at runtime.

Certain router design choices directly affect the ease of verification. For example, many router designs employ VCs to decouple the dependency between channels and buffers, allowing multiple interleaved packet flows through the same channel. A VC allocation (VA) unit is responsible for managing buffering resources for these independent packet flows. Our VA allocation unit was formally verified to preserve data integrity, and an AC unit was designed to generalize to more complex router designs, for instance, high radix routers with many VCs. Additional complexity is also introduced if packets are transferred through routers' buffers in a wormhole fashion. A correctly functioning wormhole routing architecture should ensure that all body flits follow the decisions that were made on the head flit. As described in Figure 2, this requirement can be broken down into easy-to-verify subproperties and then verified using available formal verification tools.

Finally, during *packet recovery*, data stored in the main routers' buffers is transmitted reliably over the checker network to the final destination. Therefore, we need to guarantee basic router functionality to safely salvage packets from the routers. To ensure this, basic router components (input ports, buffers, arbiters, crossbar) required for bare-bone functionality are formally verified. These components are common to all router architectures, while many of the features that are design-specific tend to be performance oriented: these types of features are disabled during recovery. Thus, our verification flow could be used to ensure bare-bone functionality for any router.

In summary, we show that ForEVeR can protect most major classes of router architectures against design errors. Additionally, by leveraging ForEVeR's guidelines to devise a formal verification plan and designing runtime monitors for resource management units, correctness guarantees can also be ensured for unconventional router designs, such as the flit reservation router [Peh and Dally 2000].

## 9. 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 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 counts expected packets, triggering recovery upon 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 types of design errors. The complete

scheme incurs only 4.9% area cost for an 8×8 mesh NoC, requiring only up to 30K cycles to recover from errors. ForEVeR hardware can also be leveraged for protecting the NoC against soft-errors. With an area cost of 11.6% over the baseline NoC, ForEVeR experiences minimal performance impact up to the flit error rate of 0.01% at low network loads.

## REFERENCES


R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco. 2011. Functional correctness for CMP interconnects. In *Proceedings of the IEEE International Conference on Computer Design*.

K. Aisopos and L.-S. Peh. 2011. A systematic methodology to develop resilient cache coherence protocols. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.

K. V. Anjan and Timothy Mark Pinkston. 1995. An efficient, fully adaptive deadlock recovery scheme: DISHA. In *Proceedings of the Annual International Symposium on Computer Architecture*.

T. M. Austin. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.

A. A. Bayazit and S. Malik. 2005. Complementary use of runtime validation and model checking. In *Proceedings of the IEEE International Conference on Computer-Aided Design*.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

D. Borrione, A. Helmy, L. Pierre, and J. Schmaltz. 2007. A generic model for formally verifying NoC communication architectures: A case study. In *Proceedings of the International Symposium on Networks-on-Chip*.

M. Boule, J.-S. Chenard, and Z. Zilic. 2007. Assertion checkers in verification, silicon debug and in-field diagnosis. In *Proceedings of the International Symposium on Quality Electronic Design*.

R. Brayton and A. Mishchenko. 2010. ABC: an academic industrial-strength verification tool. In *Proceedings of the International Conference on Computer Aided Verification*.

S. Chatterjee, M. Kishinevsky, and U. Ogras. 2012. xMAS: Quick formal modeling of communication fabrics to enable verification. *IEEE Des. Test Comput.* 29, 3.

W. Dally and B. Towles. 2003. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann.

W. J. Dally and C. L. Seitz. 1987. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.* 36, 5.

A. Dixit and A. Wood. 2011. The impact of new technology on soft error rates. In *Proceedings of the IEEE International Reliability Physics Symposium*.

A. Dutta and N. A. Touba. 2007. Reliable network-on-chip using a low cost unequal error protection code. In *Proceedings of the 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'07)*.

F. Fazzino, M. Palesi, and D. Patti. Noxim: Network-on-chip simulator. http://noxim.sourceforge.net/.

H. Foster, L. Loh, B. Rabii, and V. Singhal. 2006. Guidelines for creating a formal verification testplan. In *Proceedings of DVCon*.

O. Hammami, X. Li, and J.-M. Brault. 2012. NOCEVE: Network on chip emulation and verification environment. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*.

D. Holcomb, B. Brady, and S. A. Seshia. 2011. Abstraction-based performance analysis of NoCs. In *Proceedings of the IEEE/ACM Design Automation Conference*.

Intel. 2007. Intel Core2 Duo and Intel Core2 Solo Processor for Intel Centrino Duo Processor technology specification update.

Intel. 2010. Intel Core i7-900 Desktop processor series specification update.

A. Kahng, B. Li, L.-S. Peh, and K. Samadi. 2009. Orion 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*.

K. Kailas, V. Paruthi, and B. Monwai. 2009. Formal verification of correctness and performance of random prioritybased arbiters. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*.

J. Kim and H. Kim. 2009. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proceedings of the 2nd International Workshop on Network on Chip Architectures*.

T. Krishna, C.-H. O. Chen, W. C. Kwon, and L.-S. Peh. 2013. Breaking the on-chip latency barrier using SMART. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

X. Lin, P. McKinley, and L. Ni. 1994. Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. *IEEE Trans. Parallel Distrib. Syst.* 5, 8.

P. Lopez, J. M. Martínez, and J. Duato. 1998. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

J.-M. Martínez, P. Lopez, J. Duato, and T. Pinkston. 1997. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proceedings of the International Conference on Parallel Processing*.

A. Meixner, M. E. Bauer, and D. Sorin. 2007. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.

S. Murali, T. Theocharides, N. Vijaykrishnan, M. Irwin, L. Benini, and G. De Micheli. 2005. Analysis of error recovery schemes for networks on chips. *IEEE Des. Test Comput.* 22, 5.

E. B. Nightingale, J. R. Douceur, and V. Orgovan. 2011. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the 6th Conference on Computer Systems (EuroSys'11)*.

G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. 2010. Next generation on-chip networks: What kind of congestion control do we need? In *Proceedings of the ACM Workshop on Hot Topics in Networks*.

D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das. 2006. Exploring fault-tolerant network-onchip architectures. In *Proceedings of the International Conference on Dependable Systems and Networks*.

L.-S. Peh and W. Dally. 2000. Flit-reservation flow control. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

A. Prodromou, A. Panteli, C. Nicopoulos, and Y. Sazeides. 2012. NoCAlert: An on-line and real-time fault detection mechanism for network-on-chip architectures. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*.

S. Shamshiri, A. Ghofrani, and K.-T. Cheng. 2011. End-to-end error correction and online diagnosis for on-chip networks. In *Proceedings of the International Teletraffic Congress*.

D. Starobinski, M. Karpovsky, and L. A. Zakrevski. 2003. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Networking* 11, 3.

G. Tsiligiannis and L. Pierre. 2012. A mixed verification strategy tailored for networks on chip. In *Proceedings of the International Symposium on Networks-on-Chip*.

S. Vangal, J. Howard, G. Ruhl, et al. 2008. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE J. Solid-State Circuits*.

F. Verbeek and J. Schmaltz. 2011. Hunting deadlocks efficiently in microarchitectural models of communication fabrics. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*.

F. Verbeek and J. Schmaltz. 2012. Easy formal specification and validation of unbounded networks-on-chips architectures. *ACM Trans. Des. Autom. Electron. Syst.* 17, 1.

I. Wagner and V. Bertacco. 2007. Engineering trust with semantic guardians. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*.

D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. 2007. On-chip interconnection architecture of the tile processor. *IEEE Micro* 27, 5.

Received December 2012; revised May 2013, August 2013; accepted August 2013