Formally Enhanced Runtime Verification to Ensure NoC Functional Correctness

Ritesh Parikh
University of Michigan, Ann Arbor, MI
parikh@umich.edu

Valeria Bertacco
University of Michigan, Ann Arbor, MI
valeria@umich.edu

ABSTRACT

As silicon technology scales, modern processors and embedded systems are rapidly shifting towards complex chip multi-processor (CMP) and system-on-chip (SoC) designs, comprising several processor cores and IP components communicating via a network-on-chip (NoC). As a side-effect of this trend, ensuring their correctness has become increasingly problematic. In particular, the network-on-chip often includes complex features and components to support the required communication bandwidth among the nodes in the system. In this landscape, it is no wonder that design errors in the NoC may go undetected and escape into the final silicon, with potential detrimental impact on the overall system.

In this work, we propose ForEVeR, a solution that complements the use of formal methods and runtime verification to ensure functional correctness in NoCs. Formal verification, due to its scalability limitations, is used to verify the smaller modules, such as individual router components. We complete the protection against escaped design errors with a runtime technique, a network-level error detection and recovery solution, which monitors the traffic in the NoC and protects it against escaped functional bugs that affect the communication paths in the network. 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, a recovery mechanism delivers the in-flight data safely to the intended destination via the checker network. ForEVeR's experimental evaluation shows that it can recover from NoC design errors at only 4.8% area cost for an 8x8 mesh interconnect, with a recovery performance cost of less than 30K cycles per functional bug manifestation. Additionally, it incurs no performance overhead in the absence of errors.

Keywords

network-on-chip, NoC, functional correctness, formal verification, runtime verification

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 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.

Current methodologies for functional verification in the industry are heavily rooted on simulation-based validation and formal methods. Simulation techniques are inherently incomplete, since they cannot check all the possible execution scenarios of the system. Formal techniques, although they can theoretically provide complete guarantees of correctness, are in practice heavily limited by design complexity, and thus mostly applied only to small portions of the design. A recent trend in the research community has started to explore runtime verification solutions where the system's activity is monitored at runtime, after customer deployment, and checked for correctness. Runtime verification solutions can sidestep the negative impact of escaped design bugs by detecting their occurrence and intervening at runtime to prevent the corruption of network/processor state, loss of data and/or system failure. Their cost, however, includes i) silicon area dedicated to runtime monitoring and recovery, ii) dedicated design effort and often iii) a performance impact on the overall system due to continuous monitoring activities.

ForEVeR's approach is based on the insight that although formal methods do not scale to the complexity of an entire NoC, yet they can ensure component-level correctness which, in turn, could greatly reduce the need for 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 at the network-level. ForEVeR's runtime modules are de-

MICRO'11, December 3-7, 2011, Porto Alegre, Brazil

signed 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 the runtime verification hardware directly benefits from the designers' ability to formally verify.

1.1 Contributions

ForEVeR (**For**mally **E**nhanced **Ve**rification 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 a wide variety of functional errors in the interconnect and can ensure forward progress in the execution with no data corruptions. Finally, ForEVeR comes at a small area cost of 4.8% for an 8x8 mesh interconnect, while incurring a minimal performance impact only when an error manifests.

To the best of our knowledge ForEVeR is the first work to provide correctness guarantees in NoCs via complementary use of formal verification and runtime validation. Formal methods are employed to verify basic router functionality, ensuring that packet integrity is maintained within a router. The complementary runtime technique ensures correct forward progress in the overall network transfers. The runtime detection and recovery scheme, also enables designers to deliberately implement aggressive allocation schemes that might starve packets or complex routing and prioritization mechanisms that occasionally lead to deadlock or livelock. If starvation and deadlocks are rare, the overall performance improvement over conservative schemes outweighs the recovery overhead [2]. Together, formal verification and runtime validation guarantee that all data is eventually delivered to the correct destinations without being dropped or corrupted. ForEVeR achieves these goals by adding simple, verifiable and mostly decoupled hardware to the baseline interconnect. To enable runtime network-level detection and recovery, ForEVeR adds a lightweight checker network over the baseline NoC that can be guaranteed to operate correctly. For each data packet sent over the primary NoC, an advanced notification is transmitted over the checker network to flag a future packet delivery. Each destination maintains a count of expected packets and initiates recovery on anomalous behavior in the counters' values. During recovery, all packets in-flight in the primary NoC at the time of bug detection are reliably delivered to their intended destinations via the checker network.

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

2. RELATED WORK

Very few research works have proposed complementary use of formal and runtime techniques. Among them, Bayazik and Malik [4] suggested a hybrid verification methodology that leverages the use of hardware checkers in model checking to avoid state explosion by validating assumptions and abstractions at runtime. ForEVeR, on the other hand, uses formal methods and runtime verification in a hybrid methodology to provide complementary correctness goals for NoCs. Moreover, [4] is a generic methodology for verification of complex properties that cannot be directly applied to NoC correctness. Other works, such as [6], target the formal verification of an abstracted model of the NoC, and thus cannot guarantee correctness of the actual implementation.

Ensuring the runtime correctness of NoCs has been the subject of previous research, focusing on a variety of aspects. Several works [2, 11, 12, 17] target deadlock, prominent in adaptive routing, while others target a wider variety of errors through general end-to-end detection and recovery techniques [14]. Traditionally, the deadlock problem in NoCs is overcome by deadlock avoidance, or through detection and recovery, as in DISHA [2]. Other works [11, 12], propose more sophisticated deadlock detection mechanisms based on monitoring activity at the router channels. In contrast, ForEVeR safeguards against all kind of functional bugs, including deadlock and livelock. In addition, ForEVeR is an end-to-end solution leveraging hardware units mostly decoupled from the primary NoC, thus making minimal changes to the primary NoC design. Other end-to-end approaches for NoC runtime correctness have been surveyed in [14]. The most common error recovery scheme for NoCs is the acknowledgement-based retransmission technique [14], 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, otherwise the sender times out and re-transmits the locally-stored packet copy. Apart from large storage buffers and performance degradation due to the additional acknowledgement packets, this approach suffers from the obvious disadvantage of not being able to overcome errors such as deadlock and livelock. Moreover, since it uses the same untrusted network for re-transmission, it cannot guarantee packet delivery.

SafeNoC [1] is an alternative end-to-end runtime solution to detect and recover from functional errors in NoCs. It 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 also uses a secondary network to transfer notifications and for recovery of data packets, it has several advantages over SafeNoC. ForEVeR leverages complementary formal and runtime verification to provide a low overhead solution that guarantees NoC functional correctness under all execution scenarios. SafeNoC, on the other hand, has various points of failure. 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 is prohibitively expensive and does not guarantee completion.

Runtime verification solutions have focused so far on microprocessor designs [3, 13, 19]. In general, these solutions add hardware to verify 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. The idea of using runtime checkers has been proposed for various purposes [7, 16]. [7] champions the use of runtime monitors for post-silicon debug and in-field diagnosis, as a general design methodology, while ForEVeR leverages a set of specialized hardware monitors coupled with dedicated network recovery support.

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 providing complete formal verification of the individual network's routers. In addition, ForEVeR provides hardware additions to equip the NoC

with a runtime solution to monitor and correct the network execution at runtime. In the case that even individual network routers are too complex to be amenable to formal verification, ForEVeR proposes an additional runtime solution targeting specifically only those aspects of the routers' functionality that could not be verified during system development.

The purpose of formal verification efforts in ForEVeR is to verify local router properties that do not require any network-wide knowledge. Specifically, they are used to ensure that routers maintain packet integrity. The runtime components of ForEVeR operate to monitor the communication paths in the network, and assume correctness of operation internally to individual routers. In addition, they provide a mechanism of error recovery and forward progress once the monitors expose the occurrence of an anomaly. It is also possible that a NoC deploys complex routers designs that cannot be completely formally verified. In this scenario, additional local monitors deployed within each router track the correct operation of those aspects that could not be verified. Upon detection of a functional error occurrence, these monitors invoke the same network-level recovery mechanism already available in ForEVeR.

As discussed in [6], 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 1 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 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.

4. ROUTER CORRECTNESS

A correctly functioning router should ensure that packet's integrity is maintained while each packet transfers 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 describe our approach to router correctness by leveraging formal verification techniques, possibly augmented with runtime solutions, depending on the complexity of the router design. Our solution to provide router-correctness can be generalized to any router architecture, as discussed is Section 7. We discuss our ideas

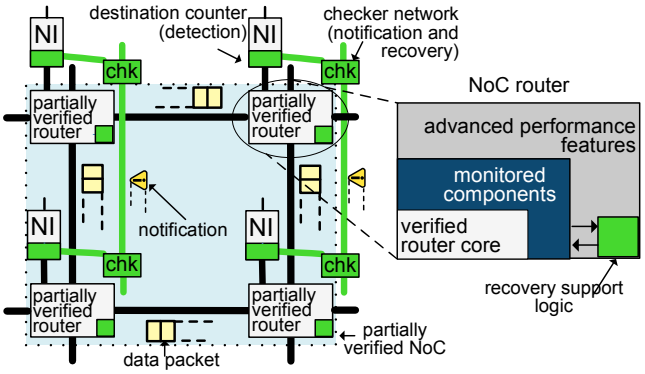


Figure 1: **High-level overview of ForEVeR.** A combination of router-level verification/runtime monitoring and network-level detection and recovery ensures correct NoC operation.

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 2a. 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 flow control manager. The control components manage the flow of packets and flits from input channels to output channels via input buffers and crossbar. The datapath components are fairly simple and can be completely verified at design-time. Verification of the control components presents a greater challenge and it is discussed in detail below. In the context of an individual router, the interactions between the virtual channels are handled by RC, VA and SA units. These units rely on information provided by the flow control mechanism, used to transmit buffer availability information among neighboring routers. Other control units, such as IVC and OVC, operate mostly on local data and hence can often be formally verified using traditional formal verification tools.

In the rest of 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 provide a runtime solution to monitor and correct any functional bug that may manifest in the incompletely verified functionality.

4.1 Formal Verification

The verification process can be efficiently partitioned into three sub-goals: 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 sub-goal, 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 sub-goal, 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 sub-goal 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 through 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 [18], a commercial formal verifica-

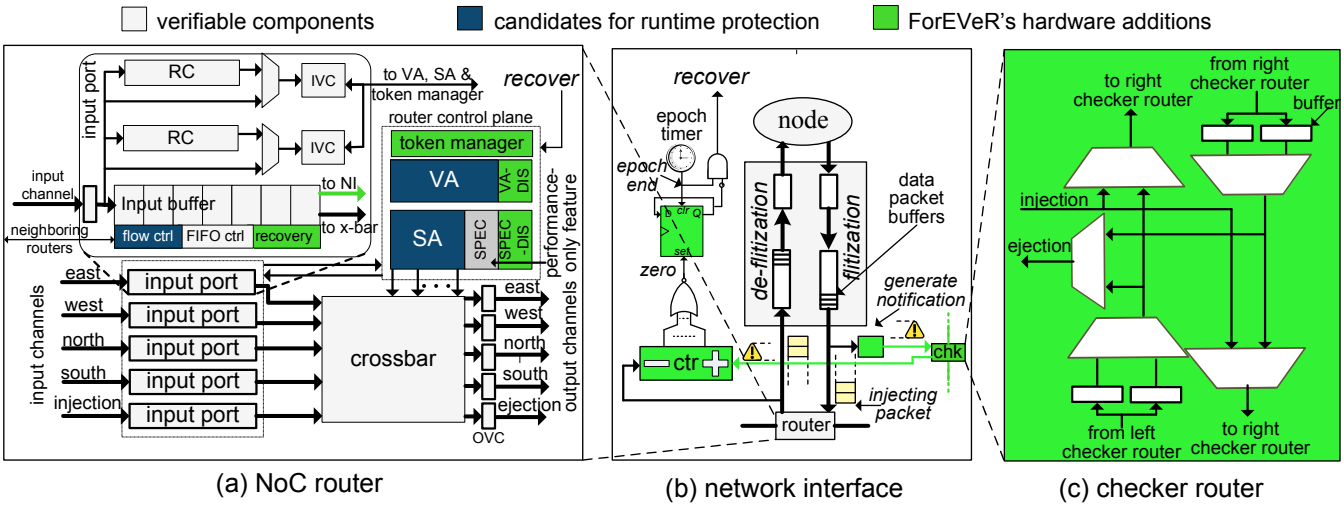


Figure 2: **ForEVeR's hardware overview.** (a) *Router additions:* VC allocator (VA), switch allocator (SA) and flow control units are monitored by runtime checkers. To implement recovery, each NoC router is augmented with VC and speculation disablers, along with a token manager and a recovery FIFO controller. (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.

Correctness goal	Property to be verified	Sub-properties	time(s)
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 (complex 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

Table 1: **Organization of router's formal verification.**

tion tool. Table 1 summarizes our sub-goals, 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. Components that are more likely to be too complex for design-time verification and thus may need to be protected by runtime techniques, when necessary, are those that control the interaction between multiple router activities [9]: VC allocator, switch allocator and flow control (as indicated in Figure 2a). We labeled these components separately in the Figure, because escaped bugs in these units may lead to critical functionality flaws, instead of just performance penalties. Note that the route control unit does not need to be protected, because its activity is monitored as part of our network-level solution.

If any of the router-level monitors detects the occurrence of a bug, we reconfigure the router to a barebone mode of operation that can be completely formally verified and performs recovery by transferring the packets while in this mode of operation, which can be guaranteed to be correct. After router-level recovery, a network-level recovery step, as discussed in section 5.2, is also performed.

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 benign, 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 a benign error as, in the worst case, it may only lead to misrouting or deadlock. Starvation is another example that needs no detection or remedy at the router level. Critical errors arise when an unreserved output VC is assigned to two input VCs, or an already reserved output VC is assigned to a requesting input VC. This situation will lead to 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 Comparator (AC) unit, a simplified version of a unit proposed in [16] 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 dur-

ing 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.

Flow control. Flow control used to manage buffer availability can lead to either buffer underflow or overflow errors. 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 re-transmitted 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 existing flow control mechanism. 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 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. To prevent the NoC routers from servicing new packets during recovery, all packet-level operations, such as route computation and VC allocation are disabled during recovery, as discussed in section 5.2. In addition, advanced "performance only" features, such as switch speculation and prioritizing mechanisms are disabled. Switch allocator and flow control 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, flow control is replaced by an acknowledgement-based control to prevent flit loss, as discussed in section 4.2.1. 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. Our checker network is extremely simple: organized in a ring topology and comprising single-cycle latency, packet-switched routers, as shown in Figure 2c. The router architecture we use is identical to the one proposed in [10]. In particular, we leverage its low latency property to consistently deliver notifications before the corresponding packets arrive through the primary network. Note that a notification carries no information other than the destination address, enabling us to design a lightweight checker network.

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 1 shows how the checker network interfaces with the primary network via the network interface units, that also include the logic for detection and recovery initiation.

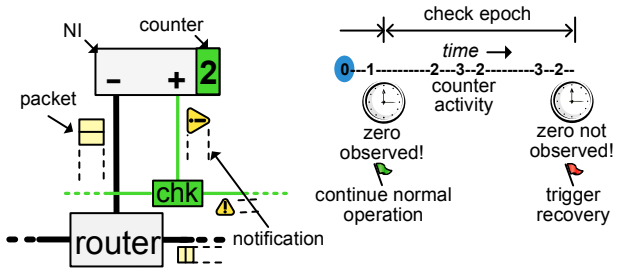
5.1 Detection

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. Figure 3a illustrates the hardware implementation and execution flow of our detection scheme and Figure 3b outlines the corresponding detection algorithm. 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 show in Section 6.2 how epoch length affects the accuracy of detection. Finally, note that design errors leading to misrouting of packets can be detected by analyzing the routing information in the header flit at the destination nodes.

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 4b. 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 positive detection, which can be caused, for instance, by a counter that does not reach zero within an *check epoch* because of high traffic. Note that false positives only impact the performance, not affecting system's 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 processing 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 4b. Since key router functionalities are still active in the



(a) Detection Implemenation

```
at each destination, at each cycle
If (received_notification_packet) then {counter++};
If (received_data_packet) then {counter--};
If (counter == 0) then {set zero.observed};
If (end_check_epoch) then {
  If (zero.observed) then {continue};
    else {trigger_recovery};
}
time++; next cycle;
```

(b) Detection Algorithm

Figure 3: **ForEVeR's network-level detection.** (a) *Detection hardware:* Each destination uses a counter and a timer, if any of the destinations do not observe a zero in the counter at least once during a *check epoch*, then recovery is initiated. (b)*Detection algorithm:* The counter is incremented or decremented upon notifications and packets arrivals, and monitored for the occurrence of a zero.

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 token loop through all routers, as certain packets may still remain in their respective buffers. Note that we do not drain a packet when its head flit is not at the front of the buffer on arrival of the token. Figure 4a shows the execution flow of the recovery phase, indicating how packet recovery (corresponding to one complete token loop) may be repeated if packets are still outstanding. The hardware components required to provide recovery are shown in Figure 2a and include: a *token manager*, *virtual channel allocation disabler* (VC-DIS) and *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. During recovery, only one router can be transmitting packets to a single destination at a time, greatly simplifying the dis-assembling/re-assembling process. To manage this, the checker network's channels include dedicated wires for head and tail indicators. The flit with head indicator carries the destination address and reserves an exclusive path between the source and the destination. All intermediate valid flits traversing the ring network are ejected at the same destination till a flit 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 our evaluation system with 64 nodes, the checker network channel is 8 bits wide (6-bit address, 2-bit head-tail indicators). Thus each 64-bit primary network flit takes 12 checker networks packets (1 head, 11 body/tail) to transfer.

5.3 Verification of 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.

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*). Table 2 reports the time required to prove these verification goals using the same formal verification setup as described in Section 4.1.

checker network correctness		**recovery operation correctness**	
verified property	time(sec)	verified property	time(sec)
injection	8	*fairness*	15
progress	156	*complete_packet*	10
ejection	86	*buffer_empty*	46
data_integrity	10	*valid_packet*	29

Table 2: **Formal verification of ForEVeR's network-level recovery operation.**

6. EXPERIMENTAL RESULTS

We evaluated ForEVeR by modeling a NoC system in Verilog HDL, as well as a cycle-accurate C++ simulator, both based on [8]. The baseline system is an 8x8 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 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 [18], a commercial formal verification tool. ForEVeR's area overhead was estimated using synthesis results from Synopsys' Design Compiler targeting the Artisan 45nm library. The C++ simulator was used to assess the accuracy of the network-level detection scheme and to evaluate the performance impact of recovering from functional bugs that we inserted in the baseline model to evaluate our solution. The framework was analyzed with two different types of workloads: directed random traffic (uniform), as well as applications from the PARSEC suite [5].

6.1 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, described in Table 3. Bugs 1-6 are errors that inhibit forward progress, bugs 7-8

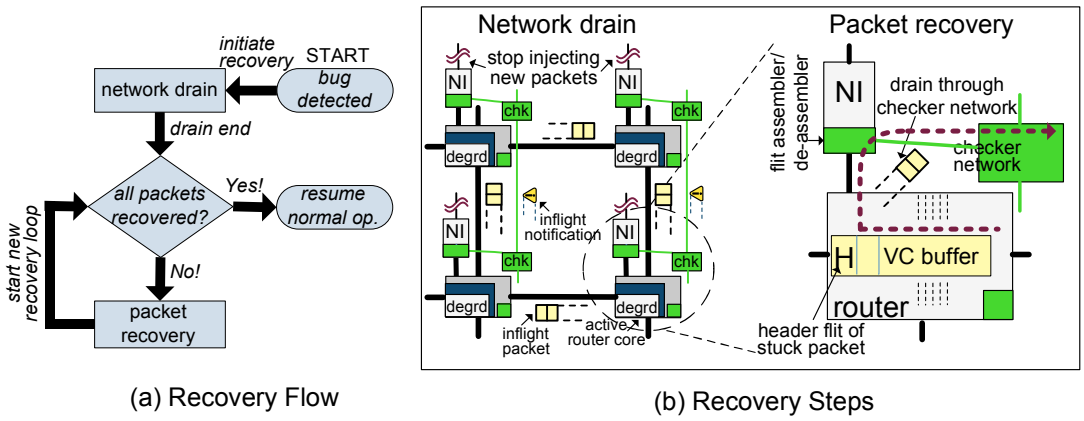


Figure 4: **ForEVeR recovery process.** (a) *Recovery flow:* Network drain is followed by packet recovery until all primary network packets are recovered. (b) *Recovery steps:* All packet injections are suspended during the network drain phase, while in-flight packets are recovered during the packet recovery phase.

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, 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 an execution overhead, due to *network drain* and *packet recovery*. During *network drain* the primary NoC was allowed to drain for a fixed period of 500 cycles, a parametric value that we set by simulating the draining of a congested network. In the rare case of network not draining completely within this time interval, an unnecessary initiation of *packet recovery* will occur, incurring a performance impact without affecting correctness. Table 3 reports the additional average *packet recovery* time incurred for each bug, averaged over all benchmarks, packet sizes, activation times and locations. It should be noted that, apart from forward progress errors that are detected at the network level, routing errors are quickly detected at erroneous destinations, whereas errors that affect router operation are detected immediately by the router's hardware monitors.

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

Table 3: **Functional bugs injected in ForEVeR and average packet recovery time.**

On average, ForEVeR spends approximately 2,633 cycles in *packet recovery* for each bug occurrence. This value is primarily affected by 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. For a network operating at 1 GHz, and exhibiting an error rate of one error every 5 minutes, this translates to a negligible performance penalty, less than one hundred millionth (10^{-8}). In practice, design errors that escape pre-silicon verification are quite infrequent.

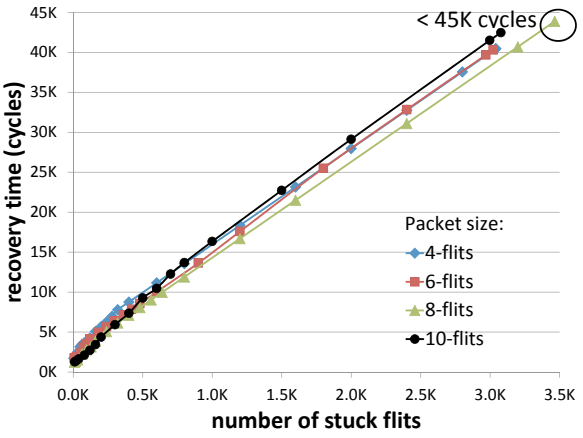


Figure 5: **ForEVeR's packet recovery time.** ForEVeR's recovery overhead increases almost linearly with the number of flits stuck in the primary NoC that must be transmitted reliably through the checker network.

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 to eject at the network interfaces. The network-level detection scheme flags an error due to un-accounted primary network packets at destinations, thus triggering *packet recovery*. Concurrently, we noted the time required to drain all stuck packets through the checker network. Figure 5 plots our results for varying packet sizes, reporting *packet recovery* time vs. number of extracted flits. Note that *packet recovery* time varies almost linearly with the number of stuck flits, requiring less than 45,000 cycles, even in the worst case.

6.2 Network-level Detection Accuracy

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

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

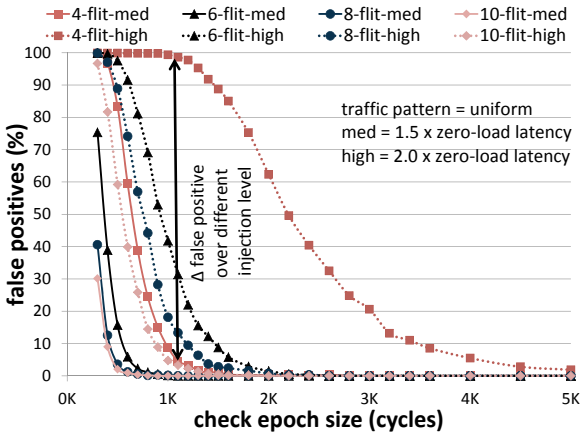


Figure 6: **ForEVeR's detection scheme under uniform random traffic.** The Figure plots the false positive rate vs. *check epoch* size, for various packet sizes. The false positive rate drops rapidly with larger *check epochs* and decreasing network load.

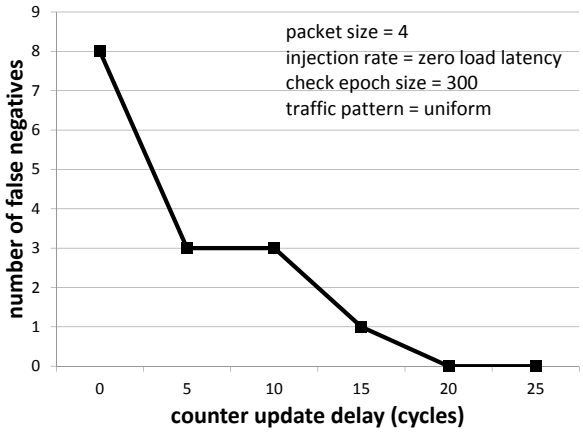


Figure 7: **False negatives vs. *primary network offset*.** The Figure plots false negatives with *primary network offset* under low latency uniform traffic with packet size of 4 flits and *check epoch* length of 300 cycles. False negatives decrease with increasing primary network offset.

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 affects 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* will reduce the false positive rate by allowing more time for packets to reach their destinations. Figure 6 shows the decrease in false positive rate with increasing *check epoch* size. The false positive rate drops to a negligible value beyond a certain *check epoch* size ($Epoch_{min}$), whose value depends on network load. Additionally, a heavily loaded network exhibits a higher false positive rate than a moderately loaded network, and hence a heavily loaded network requires a larger $Epoch_{min}$ to practically eliminate all false positives. Extensive simulations indicate that $Epoch_{min}$ rises to intolerable values only when the 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 [15].

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 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. However, to avoid false negatives in the detection scheme altogether, the checker network is constrained to deliver notifications before the corresponding data packets arrive via the primary network. If this is not the case for a baseline checker network design, this goal can still be achieved by considering design alternatives, such as bundling together multiple notifications before transmission, or using multiple checker networks, *etc.* In ForEVeR's evaluation system, our checker network almost always delivers notifications ahead of data packets, except for very low latency 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 updating of the monitor counters can be delayed by an amount determined by the maximum latency difference between primary and checker network at zero load (we call this value *counter_update_delay*). We ran low latency simulations using uniform traffic with a small packet size (4 flits) and a *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 7 plots the maximum number of false negatives observed over 10 different seeds for different *counter_update_delay* values. Note that the rate of false negatives falls quickly and are completely eliminated at a delay of 20 cycles or greater. It should be noted that the maximum latency difference between primary and checker network at zero-load was found via simulations to be 18 cycles.

Optimal epoch length. To calibrate the *check epoch*, we ran rigorous simulations using both uniform random traffic and PARSEC benchmarks. After operating ForEVeR normally for a preset length of time, a random primary network 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*.

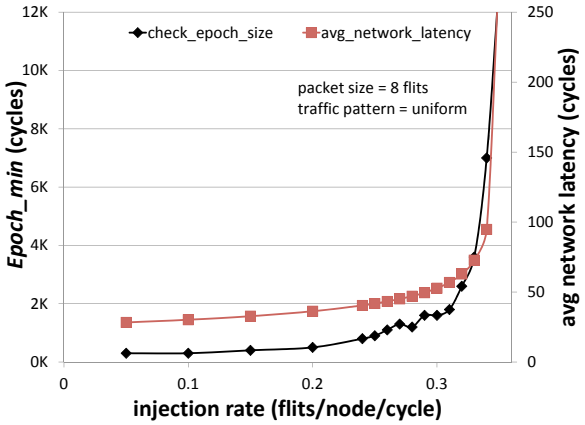


Figure 8: **ForEVeR's detection scheme under uniform traffic.** The Figure shows the variation of $Epoch_{min}$ and latency with increasing network load. $Epoch_{min}$ is within tolerable limits for all but deeply saturated networks.

Figure 8 plots $Epoch_{min}$ (necessary to minimize the false positive rate) and the average network latency as network load is varied, under uniform network traffic. $Epoch_{min}$ exhibits a slow increase

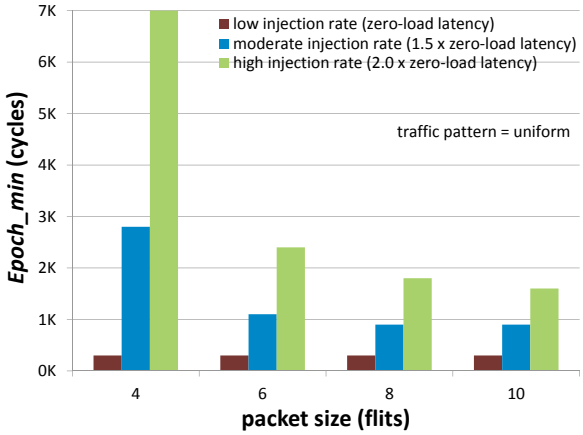


Figure 9: **ForEVeR's detection under uniform traffic.** The Figure shows the variation of $Epoch_{min}$ over different packet sizes, at low, moderate and high network load. $Epoch_{min}$ size decreases with increasing packet sizes.

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 9 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.

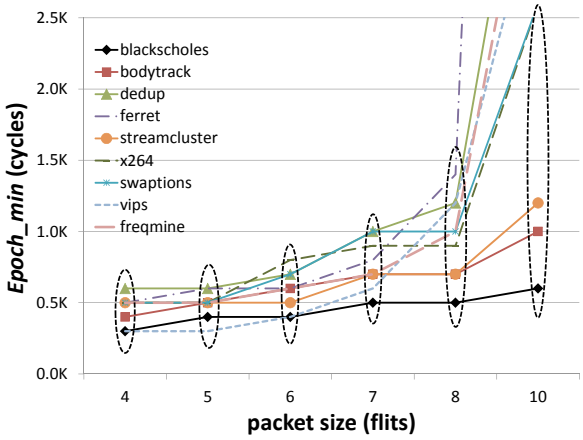


Figure 10: **Minimum required epoch length for varying packet sizes with PARSEC benchmarks**. The Figure plots $Epoch_{min}$ for different packet sizes for 9 PARSEC benchmarks. $Epoch_{min}$ is within 1,500 cycles for packets up to 8 flits (corresponding to an average network latency of 800 cycles).

PARSEC benchmark traces for evaluation of our detection and recovery scheme were extracted from a 64 core CMP system running PARSEC workloads, with our baseline NoC using 4-flit data packets. 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 network load. Using the same traces, NoCs with longer data packets (due to decreased channel width) are used to unrealistically stress the network during simulation. It should be noted that such high load scenarios (average network latency up to 1,600 cycles) should never arise in practice because of the self-throttling nature of the applications. Figure 10 plots $Epoch_{min}$ with different packet sizes for the PARSEC benchmarks, demonstrating that a zero false positives rate can be achieved with small *check epoch* sizes, even at high network loads.

6.3 Area Results

A central goal in designing ForEVeR is to keep silicon area 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. The bottom part of Table 4 reports additions for network-level correctness, indicating a 4.8% area overhead over a primary network router. The overhead is due to additions in each router, contributing 1.7%, and to each network interface and checker router, which, combined, are responsible for the remaining 3.1%.

The top part of Table 4 also reports the overhead for ForEVeR's router-level hardware monitors and reconfiguration hardware, accounting for 9.2% additional area over the baseline router. Flow control reconfiguration and extra storage required to avoid dropped flits costs 7.8%, whereas VA and SA checkers, along with a spare arbiter (Section 4.2.2) and other reconfiguration support, result in 1.4% overhead. In our framework, we were able to formally verify the baseline router completely, and hence we only incurred the network-level area cost (4.8%).

level	design	area (μm^2)	%
router	baseline router	77,723	100.00
	flow ctrl & extra storage	6,071	7.81
	VA & SA checker/reconfiguration	1,053	1.35
	router-level correctness overhead	7,124	**9.16**
network	token mgr & recovery support	1,300	1.67
	NI additions	1,550	1.99
	checker router	845	1.09
	network-level correctness overhead	3,695	**4.75**

Table 4: **ForEVeR area overhead.**

ForEVeR leverages formally verified components within the router to recover from design errors to keep the overhead low when compared to purely runtime verification techniques [1, 14]. Without these verified components there would be a need for a lot of extra hardware and added complexity. Specifically, in case of an error, a full runtime solution would require storage to duplicate all the in-flight data and retransmit that using the same unreliable network or transfer that over a secondary network that is guaranteed to be correct. For example, the popular re-transmission based runtime scheme uses the same untrusted network to re-send packets that have been dropped or corrupted. Experimental evaluations show that large duplicate-storage buffers alone result in 66% area overhead over the baseline network. In addition, this solution suffers from slowdown due to additional acknowledgement traffic.

7. GENERALIZATION

In this section, we discuss how ForEVeR's approach of complementary verification enables it to generalize to various current and future NoC and router designs.

Other NoC designs. Both detection and recovery schemes of ForEVeR can be generalized to any NoC design/architecture. ForEVeR is agnostic to NoC topology and the routing algorithm employed, as long as the checker network, used both during detection and recovery, can adapt to consistently deliver notifications ahead of time. To this end, checker network's performance can be tuned to the needs

of the baseline network by various mechanisms, such as increasing bandwidth or bundling notifications together before transmission. For our baseline design, a low bandwidth checker network sufficed to deliver notifications as required.

Other router designs. ForEVeR can be generalized to all mainstream router designs, as they have similar underlying structure, where control components manage the flow of data through the data-path components (channels, buffers, crossbars). All mainstream router micro-architectures involve buffering of data either at router inputs (input-queued) or outputs (output-queued). Buffers are managed by the flow control unit and can be allocated per flit or per packet. Similarly, access to resources like channels or crossbar links, is always determined by arbitration or allocation logic.

ForEVeR's hardware monitors for router-level detection provide protection to router components that handle complex interactions such as flow control and resource allocators. Although our baseline router implementation is completely formally verified, we designed these generalized hardware monitors (flow control checkers and allocation comparators) to be able to extend ForEVeR's detection scheme to more complex router designs that may be outside the scope of formal verification. These also enable designers to architect aggressive that are not guaranteed to be completely correct.

As for ForEVeR's recovery scheme, 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 type of features are disabled during recovery. Thus, our verification flow could be used to ensure bare-bone functionality for any router.

8. 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 keeps a count of 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 4.8% area cost for an 8x8 mesh NoC, requiring only up to 30K cycles to recover from errors.

Acknowledgments

This work was developed with partial support from the National Science Foundation and the Gigascale Systems Research Center.

9. REFERENCES

[1] R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco. Functional correctness for CMP interconnects. In *Proc. ICCD*, 2011.

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

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

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

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

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

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

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

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

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

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

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

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

[14] S. Murali, T. Theocharides, N. Vijaykrishnan, M. Irwin, L. Benini, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 22(5), 2005.

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

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

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

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

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