Compositional Performance Verification of NoC Designs

Daniel E. Holcomb
UC Berkeley

Alexander Gotmanov
Intel

Michael Kishinevsky
Intel

Sanjit A. Seshia
UC Berkeley

***Abstract*—We present a compositional approach to formally verify quality-of-service (QoS) properties of network-on-chip (NoC) designs. A major challenge to scalability is the need to verify latency bounds for hundreds to thousands of cycles, which are beyond the capacity of state-of-the-art model checkers. We address this challenge by a compositional form of k-induction. The overall latency bound problem is divided into a number of sub-problems, termed *latency lemmas*. Each latency lemma states that a packet spends a smaller number of cycles at a particular "stage" of progress. We present a partially-automated method of computing these stages based on the topology of the network and a subset of relevant state, and verify the latency lemmas using k-induction. The effectiveness of this compositional technique is demonstrated on illustrative examples as well as an industrial ring interconnection network.**

I. Introduction

Network on chip (NoC) is a paradigm for communication within large many-core system on chip (SoC) designs. An NoC architecture is a network of interconnected nodes, where each node has networking logic and is associated with a processor core, memory controller, specialized IP block, etc. Industrial examples include the Tilera TILE64^{TM} processor [1], STI Cell BE [2], and Intel Larrabee [3]. NoCs typically offer quality of service (QoS) guarantees on worst case latency, jitter, and throughput for some classes of network traffic. QoS violations are often similar to starvation and deadlock scenarios and can be easily missed in performance simulation.

High-level modeling of NoCs [4], [5] and automatic abstraction [6] help to hide unnecessary details, easing the way for formal analysis. Even so, verifying QoS properties can be challenging for industrial NoC designs, both due to the scale of the design and the property to be verified. Consider for instance the problem of proving an upper bound on the latency of sending a packet from one node in the network to another. In principle, this property can be expressed in linear temporal logic (LTL), and the problem can be solved using model checking. The LTL property expresses a *bounded liveness property*, written in English as "every packet from source A gets to its destination B within N cycles." Bounded liveness is essentially a safety assertion where one adds some extra logic to track the progress of time. One can use state-of-the-art model checking strategies such as k-induction [7], interpolation [8], and IC3/property-directed reachability (PDR) [9], [10] to verify this property. However, regardless of the strategy, it is necessary to analyze at least N consecutive cycles to either prove or disprove the latency bound, assuming that N is tight. Typical latency bounds for NoCs can be in hundreds or thousands of clock cycles. Unrolling of model transition relation to such depth is beyond the capacity of state-of-the-art model checking engines. Property-directed reachability [9], [10], while avoiding explicit unrolling of the transition relation, still does not scale past tens of clock cycles.

In order to address this challenge, in this paper we use a tried-and-tested approach in formal verification: *compositional reasoning*. In compositional reasoning, one breaks up the overall proof obligation (proving a latency bound of N cycles) into a number of "smaller" proof sub-goals, which are much easier to verify, such that if all of the sub-goals are proved, then so is the original property. The key is to devise a decomposition that is well-suited to the verification task at hand. A natural approach for latency bounds is to first prove smaller bounds on a packet's progress through the network; e.g., how long does it take to inject a packet into the network, how much time does it spend along a particular subpath, etc. We term these proof sub-goals *latency lemmas*. Methods to discover and apply them are the core contributions of this paper.

Specifically, we show that for some common network topologies, one can enumerate finitely many *stages* that a packet can go through. Each location in the network belongs to at least one stage at every time moment. Stages are arranged into a directed, acyclic *stage graph*, to capture the order in which they can be visited by a packet. A latency lemma bounds the number of cycles between when a packet is injected into the network and when the same packet exits some particular stage. By proving the latency lemmas for all stages, one proves the bounds corresponding to all paths through the network.

To summarize, we make the following novel contributions in this paper:

- A compositional approach to proving latency bound properties in NoCs by decomposition into latency lemmas.
- Methods of formulating latency lemmas using a *stage graph* based on the topology of the network and control state.
- Experimental results on two illustrative examples show that our approach can reduce the runtime of inductive verification of latency bounds by 4x-55x, and reduce the runtime of the state-of-the-art IC3/PDR technique by 2x. However using k-induction to verify latency with lemmas added gives a 8x-15x overall reduction in runtime relative to verifying the same property using PDR.
- Experimental results on an industrial-style ring interconnection network show that latency lemmas give a significant speedup, and in all cases allow latency bounds to be proved inductively. Several variants of the ring can only be verified within the allotted runtime when latency lemmas are used.

The rest of the paper is organized as follows. Sec. II introduces basic terminology and sketches our approach using a simple example. Sec. III describes our compositional approach in detail. Sec. IV describes our strategy for creating a stage graph. Results for illustrative examples are presented in Sec. V, and for the ring

network in Sec. VI. Related work is presented in Sec. VII and we conclude in Sec. VIII.

II. Preliminaries

A. Background

We describe NoC designs using a high-level modeling formalism called executable micro-architectural specifications (xMAS models) [4]. xMAS models are compositions of simple primitives, communicating over channels. Each channel c is a communication link between an initiator primitive and a target primitive, and comprises three signals $c.data$, $c.irdy$, and $c.trdy$. The initiator controls $c.data$ and $c.irdy$, while the target controls $c.trdy$. Data is transferred from initiator to target whenever $c.irdy$ and $c.trdy$ are both asserted during the same cycle. A channel c is said to be blocked (by the target) when $c.irdy$ is asserted and $c.trdy$ is not. A channel c obeys a liveness bound x if temporal logic formula $c.irdy \implies \mathbf{F}_{\leq x}\ c.trdy$ holds, where $\mathbf{F}$ is the temporal operator "Eventually". A liveness bound of $x = 0$ means that a channel never blocks. A channel c is persistent if $c.irdy$, once asserted, remains asserted until an eventual transfer occurs [11].

An xMAS model of an NoC $\mathcal{N}$ is a tuple $\langle \mathcal{I}, \mathcal{O}, \mathcal{B}, \mathcal{C}, Init \rangle$ where

- $\mathcal{I}$ is a finite set of input signals or *sources*;
- $\mathcal{O}$ is a finite set of output signals or *sinks*;
- $\mathcal{B}$ is a finite set of first-in first-out (FIFO) buffers;
- $\mathcal{C}$ is a finite set of arbitrary logic components, and
- *Init* is a set of initial states.

Each buffer $b \in \mathcal{B}$ comprises one or more buffer slots, and every slot in network $\mathcal{N}$ is indexed by a unique identifier i. The components $c \in \mathcal{C}$ are stateful or stateless xMAS primitives. Due to lack of space, we provide here only brief descriptions of the subset of xMAS components that we use. The interested reader is referred to [4].

1. *Queue:* parametrized by its number of slots k. Packets are read from the fixed head slot, and written to tail position that varies with the number of packets stored in the queue. When a packet is read from the head slot, all other packets in the queue advance to the next slot.
2. *Source:* a source non-deterministically attempts to send a packet through its output channel o. Alternatively, an *eager* source attempts to send a packet on every cycle. The data of the injected packet is also non-deterministic.
3. *Sink:* a sink non-deterministically consumes a packet from its input channel i. An eager sink attempts to consume a packet on every cycle. Finite latency bounds require that sinks cannot indefinitely block traffic. We therefore enforce that packet sinks guarantee their input channels to obey bounded liveness. A bounded live sink is parameterized by x, the liveness bound that it guarantees for its input channel i.[1]
4. *Fork:* parameterized by functions f and g, consumes a packet from i and produces both $a = f(i)$ and $b = g(i)$;
5. *Join:* parameterized by function h, consumes a packet from both inputs a and b and produces output $o = h(a,b)$;
6. *Switch:* parameterized by a switching function s, consumes a packet from i and produces it on a if $s(i) = \textbf{true}$, and on b otherwise;
7. *Merge:* arbitration primitive that consumes an input packet from either a or b, and produces the same packet on o. A state bit u stores the arbitration priority among the inputs, and its updation ensures local fairness.

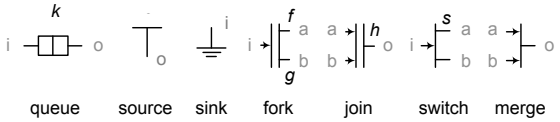

Figure 1: The set of XMAS primitives. Inputs and outputs are written in gray and the parameters of the primitives are written in black [4].

B. Sketch of Approach

A single state of the network includes a snapshot of the packets that may occupy each queue slot, and various bits of control state. From such a snapshot, our approach conjectures an upper bound on how long each packet has been in the network, based on its location in the network or designated state variables. These conjectures are termed latency lemmas, and described formally in Section III.

We sketch our approach using a very simple network: a queue of depth 5 between a non-deterministic source and an eager sink (Fig. 2). Our approach uses a conservative compositional technique to verify a latency bound of 6 cycles by using one latency lemma for each queue slot to assert the following conditions.

- Any packet in slot 1 is less than 2 cycles old.
- Any packet in slot 2 is less than 3 cycles old.
- Any packet in slot 3 is less than 4 cycles old.
- Any packet in slot 4 is less than 5 cycles old.
- Any packet in slot 5 is less than 6 cycles old.

The conjunction of the latency lemmas implies a global latency bound of 6 cycles because the lemmas cover all possible locations of a packet, and all of the bounds are less than or equal to 6 cycles[2].

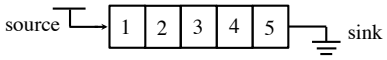

Figure 2: Example of a queue with 5 slots.

The efficiency of our approach comes from the implicit composability of the latency lemmas. If all lemmas hold in one cycle, then the transition relation of the network ensures that they will

[1] Sinks are assured of satisfying liveness bounds by their transition relations; if the current cycle would be the x^{th} consecutive cycle in which $i.irdy \wedge \neg i.trdy$, then the sink is forced to assert $i.trdy$.

[2] Note that the overall latency bound of 6 cycles is loose. The eager sink prevents the queue from ever filling more than one slot (slot 5), so the worst achievable latency is 1 cycle.

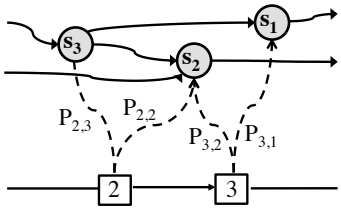

Figure 3: Correspondence between network $\mathcal{N}$ and stage graph G.

also hold in the next cycle. The conjunction of all latency lemmas comprises an inductive invariant for the age of all packets. As an inductive invariant, the property can be verified efficiently without a high degree of unrolling. While this toy example might be verified using (one-step) induction, in general we construct sets of composable lemmas that are verifiable with k-induction. The values of k are small on account of the latency lemmas each describing only a small increment of progress. Useful latency lemmas can be automatically discovered in many acyclic networks, and can be discovered with the help of manual insight in cyclic networks.

III. Formalism

As sketched in the previous section, we use a set of conjectured latency lemmas in order to efficiently verify a bound on the end-to-end latency from any source in the network to any sink.

The model being verified is xMAS model $\mathcal{N}$. We assign a unique index i to every queue slot in the network, and let variable q_i refer to the content of the i^{th} queue slot. The complete state of a network $\mathcal{N}$ with L total queue slots comprises variables $q_0, q_1, \ldots, q_{L-1}$ representing slot contents, and some additional control variables.

A. Mapping from $\mathcal{N}$ to G

Let graph $G = (S, E)$ be an acyclic digraph, with its vertices $s_0, s_1, \ldots, s_M \in S$ called stages. Each stage corresponds to a latency lemma. Queue slots in $\mathcal{N}$ map to stages in G depending on state; all sinks in $\mathcal{N}$ map to special stage s_0. If a slot i maps to stage j, then it means that slot i is subject to the j^{th} latency lemma. There exists an edge $e_{j,k} \in E$ if a packet can occupy slots in $\mathcal{N}$ that map to stages s_j and s_k in consecutive cycles.

In each state of execution, every queue slot in $\mathcal{N}$ that stores a packet is mapped to some stage in G (Fig. 3). A single queue slot can map to different stages depending on the state of certain variables, but always maps to exactly one stage unless the slot is empty. The mapping is defined using a set of formulas $p_{i,j}$. Each formula refers to a particular slot i in $\mathcal{N}$ and stage $s_j \in G$, and is **true** if the packet in slot i maps to stage s_j. In other words, $p_{i,j}$ is the Boolean function defining conditions under which slot i maps to stage j, and the function takes as inputs variables $q_i \in Q_i$ and $c \in C$.

Latency lemmas for a given slot are forbidden from considering the contents of any other slots so as to avoid the corresponding blow-up in the number of stages of G. This restriction on the form of latency lemmas can introduce looseness in the latency bounds by forcing each packet to always make conservative assumptions about other traffic.

$$p_{i,j} : Q_i \times C \mapsto \mathbb{B} \tag{1}$$

- Q_i is the set of states of queue slot i.
- C is the set of states of the control variables used to enforce fairness, including priority bits of merge primitives and reservation states of control logic.

A few special cases are worth mentioning. If slot i can never map to stage s_j then $p_{i,j} = \textbf{false}$ regardless of q_i and c. If slot i always maps to stage s_j, then $p_{i,j} = \textbf{true}$, regardless of q_i and c. We say some combination of slot i and states (q_i, c) are *covered* by stage s_j if formula $p_{i,j} = \textbf{true}$ for (q_i, c). Alternatively, we sometimes say that a *packet* is covered by s_j if it resides in the slot i at a time when $p_{i,j}$ is true.

All packets in all reachable states should be covered by some stage in G during every cycle. For each slot i, assume the existence of a specification variable $used_i$ that is **true** in every state where slot i stores a packet. A coverage property X is defined as **true** if every used slot i in $\mathcal{N}$ maps to at least one stage in G.

$$X := \bigwedge_{i \in [0,L-1]} \left(used_i \implies \left(\bigvee_{s_j \in S} p_{i,j} \right) \right)$$

B. Latency Lemmas to Imply Global Latency Bound

The xMAS model $\mathcal{N}$ is augmented with a global clock and packet timestamps to allow latency bounds to be evaluated as safety properties over single states. The current time clk and the injection timestamps $t(q_i)$ are specification variables in the xMAS model. Variable clk is the state of an n-bit counter that increments during every cycle. The timestamp is created by appending the current value of clk to every packet when it is first injected from a source into the network. When a packet occupies slot i, its timestamp $t(q_i)$ is part of the slot's state q_i. Each queue slot in $\mathcal{N}$ is therefore widened by n-bits to accommodate the packet timestamps.

The global latency bound property and the latency lemmas both assert claims about the *age* of a packet. The age of the packet in slot i (denoted $age(q_i)$), is the difference between the current time and the packet's timestamp (Eq. 2). Property Φ^G (Eq. 3) asserts that no packet has an age of T_{VER}. Latency lemma ϕ_j (Eq. 4) asserts that any packet in a slot i that satisfies $p_{i,j}$ (i.e. maps to stage s_j) has an age less than T_j. Property Φ^L (Eq 5) asserts that all latency lemmas hold.

$$age(q_i) := \quad (clk - t(q_i)) \mod 2^n \tag{2}$$

$$\Phi^G := \bigwedge_{i \in [0,L-1]} (used_i \implies age(q_i) < T_{VER}) \tag{3}$$

$$\phi_j := \bigwedge_i \left(p_{i,j} \implies age(q_i) < T_j \right) \tag{4}$$

$$\Phi^L := \bigwedge_j \phi_j \tag{5}$$

Latency lemmas are helpful because their composable nature allows them to be verified without a large number of unrollings. While the packet timestamps are ostensibly added to a packet when it is injected, the general initial state of induction allows for packets to exist without being injected[3]. Because the latency lemmas constrain the age of packets at intermediate stages of progress, it is not necessary for the verifier to unroll the circuit to a depth that is proportional to the total path latency. Each lemma can essentially ignore the prefix path, and assert an age bound building only upon the lemmas of direct predecessor stages. This simplification provides a large reduction in the number of unrollings required and in the solver runtime, but could introduce looseness into the verified bounds.

C. Automated Invariant Strengthening

An advantage of using the xMAS formalism is automated invariant strengthening. The automatically generated invariants are unrelated to QoS, but are useful to block the verifier from exploring unreachable states. The set of inferred invariants is denoted Ψ, and can include numeric invariants [5] and channel persistency invariants [11] among others. This work uses channel persistency invariants, non-blocking invariants on selected channels, and invariants to specify that the head and tail pointer of each queue are consistent with the number of items stored in it.

Including the automated invariants and latency lemmas, the total verification problem becomes $\mathcal{N} \vDash \Phi^L \wedge \Psi \wedge \Phi^G$. The approach is sound because the verified property is a strengthening of the original latency property Φ^G.

IV. Rules for Constructing Stage Graph

The $p_{i,j}$ formulas that define latency lemmas can be chosen to be more abstract or precise, and we do not offer a scheme for choosing an optimal level of abstraction. Two extreme cases must be avoided in defining these formulas. A mapping that is unnecessarily precise may induce too large a stage graph G. A mapping that is too abstract can induce a cyclic graph, or a graph where single stages are too large to serve as effective sub-goals. Our approach is to define the mapping as abstractly as possible using simple mechanical rules, and to refine the mapping by case-splitting on fairness variables whenever graph G is cyclic or its sub-goals are too large.

A. Propagating Liveness Bounds in $\mathcal{N}$

Latency depends largely on blocking caused by congestion. In an xMAS network, bounded liveness properties assert limits on the amount of blocking that can occur. Starting from the given liveness bounds of packet sinks in $\mathcal{N}$, simple rules back-propagate the liveness bounds across primitives to obtain finite liveness bounds for other channels.

For any persistent channel c, let $d_t(c)$ represent its liveness bound, or stated differently a bound on the number of consecutive cycles in which the channel can be blocked. A claim of $d_t(c) = 2$ is then just a compact notation for temporal logic formula $c.irdy \implies \mathbf{F}_{\leq 2} c.trdy$. For a non-blocking channel, $d_t(c) = 0$. The liveness bounds $d_t(c)$ are not explicitly verified, but are instead only used as a tool toward calculating the residence times of the stages in G. The rules to propagate liveness bounds through primitives to other channels are as follows:

1. *Sink:* For a sink with a liveness bound of x, $d_t(i) = x$
2. *Queue:* For a queue $d_t(i) = d_t(o)$, indicating that the queue itself does not add any backpressure beyond the backpressure from its output channel.[4]
3. *Merge:* In a fair merge primitive no input is ever blocked through a time period when the other merge input is granted twice, therefore $d_t(a) = d_t(b) = 2d_t(o)+1$.
4. *Switch:* Without considering which output a packet will be routed to, a conservative bound is given by $d_t(i) = \max(d_t(a), d_t(b))$.

B. Age Bounds of Stages in G

The liveness bounds of the previous subsection are a step toward discovering an age bound (T_j) for each latency lemma ϕ_j. Let the maximum number of consecutive cycles during which a packet can map to stage j be called the residence time of stage j, denoted by w_j. If a queue slot i is the head of a queue primitive, and is the only slot that stage j can cover, then w_j is one greater than the liveness bound on the output channel of the queue. Every stage covering another slot of this queue is assigned the same residence time, since non-head packets will advance within the queue whenever the head packet departs the queue (see Fig. 4). The age bound T_j for a packet in any stage j is one greater than the sum of all the residence times along the longest path to stage j (including j's own residence time w_j).

For topologically acyclic networks using the xMAS components listed above, the back-propagation scheme deduces liveness bounds for all channels and maximum delays for each queue slot. The age bounds T_j can be obtained by dynamic programming in this acyclic case, as demonstrated in the tree saturation example of Subsec. V-C.

In cyclic networks, dynamic programming generates infinite T_j for any stage j that is part of a cycle in G. The cycle can be eliminated by refining the stages using case-splitting on some fairness variables. A network that can guarantee finite latency may always have some refinement that produces an acyclic stage graph, but we make no claim about finding it automatically. In the ring example of Sec. VI, we find that the fairness mechanisms used to *ensure* finite latency are useful for *verifying* finite latency bounds. Automation of this refinement step is left to future work. The ring of Sec. VI is an example of a cyclic network that requires refinement. A similar refinement step can also be used to reduce the residence times of stages in acyclic networks, as is shown in the tree saturation example (Subsec. V-C).

V. Illustrative Examples

Several illustrative examples are used to highlight strengths and weaknesses of the proposed approach. The first example is a

[3]These packets would have been injected prior to the current k frames of unrolling

[4]A special case exists when a queue has size 1, disallows simultaneous read and write, and has an eager output channel. Under those conditions, $d_t(i) = 1$ and $d_t(o) = 0$.

single queue to introduce the approach, and the second shows how refinement can be used to reduce induction depth. In these two examples, T_{VER} can be obtained mechanically with no manual insight required. The ring network in Sec. VI will show an example where insight is required.

A. Experiment Methodology

The methodology used across all experiments is described here. The xMAS models are written in word-level Verilog, with parameterized modules implementing the xMAS primitives. The Verilog is bit-blasted into an and-inverter-graph[5] (AIG) using the VeriABC flow [12]. Verification is performed on the AIG using the bit-level model checker ABC [13][6] on a 2.4GHz Intel Core i5 processor with 4GB of RAM. Both k-induction[7] and property directed reachability[8] (PDR) [10] are performed by ABC. The networks used in experiments are available online in both Verilog and AIG format.[9]

Note that verification using k-induction consists of a base-case of k frames of bounded model checking and an inductive step of k frames. When attempting to verify a property with k-induction, we do not know in advance what value of k will be necessary. We therefore take the conservative approach of allowing the base case to explore a large number of frames k_{max}, and only learn k after the verification terminates. Note that in all cases $k_{max} \geq k$. To avoid letting the arbitrary choice of k_{max} skew the runtimes, we report only the runtime for the inductive step. The runtime of the inductive step is usually orders of magnitude larger than for the base case.

The tightness of T_{VER} in each example is quantified using bounded model checking (BMC). Let T_{CEX} be the largest latency for which BMC can find a counterexample within some allotted resource limits. The smallest latency that can possibly be a valid latency bound is then $T_{CEX}+1$. Therefore, the maximum amount by which T_{VER} over-approximates the tightest possible latency bound is $T_{VER}-(T_{CEX}+1)$. The determination of T_{CEX} for all examples is given in Table IX in the appendix.

B. Single Queue

A queue of depth 5 is shown in Fig. 4, along with its stage graph G. The sink obeys a liveness bound of 2 cycles, while the source is completely non-deterministic. Each queue slot maps to one stage (Table I), and the age bound (T_j) for each stage is 3 cycles larger than for the preceding stage.

The k-induction and PDR engines in ABC are both used to verify the latency (Tab. II). The first property is a conjunction of the latency lemmas (Φ^L), the automatically generated invariants (Ψ), and the global latency bound (Φ^G). K-induction verifies this property 12X faster than does PDR. When latency lemmas are removed from the property, the runtimes of both engines are increased. Regardless of the engine used, strengthening the global latency property using latency lemmas appears to create an easier verification problem.

[5]http://fmv.jku.at/aiger

[6]Rev. d0170182dbd6; at http://www.eecs.berkeley.edu/~alanmi/abc/

[7]ABC commands "read_aiger foo.aig; bmc3 -F k_{max}; orpos; ind -F k_{max};"

[8]ABC commands "read_aiger foo.aig; pdr -v;"

[9]http://www.eecs.berkeley.edu/~holcomb/memocode12_xmas.tar.gz

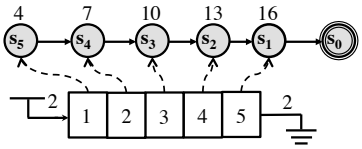

Figure 4: Queue with a sink obeying a liveness bound of 2 cycles, and corresponding stage graph G.

j	w_j	T_j	$p_{i,j}(q_i,c)$
1	3	16	$p_{5,1} :=$ **true**
2	3	13	$p_{4,2} :=$ **true**
3	3	10	$p_{3,3} :=$ **true**
4	3	7	$p_{2,4} :=$ **true**
5	3	4	$p_{1,5} :=$ **true**

Table I: Graph G from Fig 4. The index of each stage is j. The residence time of each stage is w_j. The age bound of each stage is T_j. Formulas $p_{i,j}$ define the mapping from $\mathscr{N}$ to G.

Runtime (s)	Frames	Proved	Engine	Property
1.24	5	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
4.50	17	Y	kind	$\Psi \wedge \Phi^G$
15.31	47	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
41.96	45	Y	pdr	$\Psi \wedge \Phi^G$
129.00	53	Y	pdr	Φ^G

Table II: Results for a queue of depth 5 and sink with liveness bound 2 (Fig. 4). In this example, $T_{VER}=16$ and $T_{CEX}=14$ (Tab. IX).

1) Varying Queue Depth: Next we repeat the verification of property $\Phi^L \wedge \Psi \wedge \Phi^G$ for different queue depths. In each case, the stage graph is modified according to the queue depth. The results are shown in Table III. Regardless of the queue depth, only 5 frames are required to verify the property using k-induction, because each latency lemma builds upon the lemmas for the other stages. The number of frames required by PDR generally increases with the queue depth.

	K-ind		PDR	
Depth	Frames	Run Time	Frames	Run Time
2	5	0.03	35	1.01
3	5	0.10	39	3.18
4	5	0.49	49	8.04
5	5	1.25	47	15.31
6	5	2.12	49	30.39
7	5	5.72	54	49.98
8	5	10.30	67	93.19

Table III: The runtime to verify $\mathscr{N} \models \Phi^L \wedge \Psi \wedge \Phi^G$ for different queue depths using k-induction and PDR.

2) Proving Lemmas in Isolation: A final experiment on a single queue is performed in order to investigate whether proving the

Property	Frames	Run Time	Proved
Φ^L	5	**1.12**	Y
ϕ_5	5	0.04	Y
ϕ_4	8	0.10	Y
ϕ_3	11	0.26	Y
ϕ_2	14	0.56	Y
ϕ_1	17	0.98	Y
		1.94	

Table IV: Using k-induction, the cumulative runtime for checking each latency lemma in isolation exceeds the runtime for checking the conjunction of all lemmas. 1.94 seconds is the cumulative runtime for the individual lemmas.

Runtime (s)	Frames	Proved	Engine	Property
2.72	9	Y	kind	$\Phi^{L'} \wedge \Psi \wedge \Phi^G$
4.13	13	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
227.86	32	Y	kind	$\Psi \wedge \Phi^G$
36.14	58	Y	pdr	$\Phi^{L'} \wedge \Psi \wedge \Phi^G$
35.27	58	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
73.91	72	Y	pdr	$\Psi \wedge \Phi^G$
88.45	74	Y	pdr	Φ^G

Table VI: Results for example of Fig. 5. In this example, $T_{VER} = 31$ and $T_{CEX} = 26$ (Tab. IX). Property $\Phi^{L'}$ asserts the refined latency lemmas. Refining the lemmas reduces the induction depth required to verify the bounds.

conjunction of latency lemmas (Φ^L) is easier than proving each one (ϕ_j) in isolation. The results are shown in Table IV. In each case, the lemmas are strengthened by Ψ. The cumulative runtime for verifying each lemma individually exceeds the runtime for verifying all lemmas together. The lemmas for later stages of progress require a larger number of frames to prove in isolation because they lack any information about the age of packets entering the stage. When all lemmas are verified together, the age of a packet entering the stage is bounded by the lemma of the previous stage.

C. Tree Saturation

Fig 5 shows a network that can suffer from tree saturation, even though both sinks are designated as non-blocking. The liveness bounds shown above each channel of the network are obtained using the propagation scheme of Subsec. IV-A. The channel liveness bounds and the residence times of corresponding stages grow quadratically with the number of merge primitives between the channel and the sink. The stage graph uncovered by this automated procedure has a one-to-one correspondence between queue slots in $\mathscr{N}$ and stages in G. The stage graph is shown above the network in the figure, with the age bounds (T_j) annotated above each stage. The stages (s_{10}, s_9) corresponding to two particular queue slots $(1,2)$ have residence times of 8 cycles each.

It is possible to reduce the largest residence times by refining the stage graph. A packet can only reside in slot 1 or 2 for 8 cycles if the merge primitive ahead of it transfers a packet from its other input during this time; the priority bit u of the merge primitive flips its state when this transfer occurs. We can therefore refine the mapping of packets in slot 2 based on u. A packet can reside in slot 2 for 4 cycles as the low-priority merge input, and 4 cycles as the high-priority merge input; the high-priority case indicates a more advanced stage of progress. The same type of refinement is applied to slot 1. A packet in slot 1 is in a more advanced stage of progress when the merge prioritizes its upper input, because it will then reach slot 2 within 4 cycles. The refined stage mappings are given in Table V and shown above the nominal stage graph in Fig. 5. Note that the stages are defined conservatively according to worst-case behaviors; a packet from a low priority merge input can be transferred if the other merge input is idle.

The results for verifying the latency of Fig. 5 are given in Table VI. Property $\Phi^{L'}$ asserts the latency lemmas of the refined stage graph. The induction depth needed is reduced by 4 when the stages with residence times of 8 cycles (s_{10}, s_9) are each replaced by multiple stages with residence times of 4 cycles $(s_{10''}, s_{10'}, s_{9''}, s_{9'})$. The use of latency lemmas reduces the runtime for both PDR and k-induction, and k-induction gives an 8x-15x speedup over PDR. When latency lemmas are not used, k-induction must resort to a larger number of unrollings and has 3x longer runtime than PDR.

VI. Non-Stallable Ring Interconnect

A ring network [14] is a topology used to route traffic among a number of agents. Each agent (Fig. 6) comprises arbitration logic, a ring buffer slot[10], and ingress buffer slots. Packets reach their destinations by moving around the ring until being granted access to the ingress buffer of their destination agent. The general ring network is configured by 3 parameters: the number of agents N, the ingress buffer depth I of each agent, and the liveness bound R on the sinks of each agent. Figure 6 shows a ring instance with $(N,I,R) = (3,2,1)$.

A packet being sent from agent i's source to agent m's sink is first injected into the ring buffer of agent i. After entering agent i's ring buffer, the packet moves around the ring, requesting entry to the ingress buffer whenever it arrives at agent m. If the request is not granted, the packet *bounces* back onto the ring to repeat the request next time it reaches agent m. The agents use unfair arbitration logic to prioritize traffic in the ring over traffic attempting to enter the ring. This ensures that traffic within the ring cannot be blocked, but does permit sources to be blocked indefinitely. A naïve ring implementation can have infinite latency even if all sinks obey bounded liveness. A single packet on the ring may never be granted access to the ingress of its destination, despite an unbounded number of other packets being granted access to the same ingress.

Receive reservations [15] are a type of mechanism to enforce fairness. When combined with bounded liveness of sinks, receive reservations guarantee that all packets have finite latency bounds. The particular receive reservation scheme used by all agents is described here:

1) If the agent's reservation token is available, then the agent issues it to any packet that is bounced (due to a full ingress). The next ingress slot to become free will be reserved for this packet.

[10]To avoid the extra cycle of backpressure that can be introduced by queues of depth 1 with non-blocking outputs, the ring buffer slot is allowed to be read and written during the same cycle.

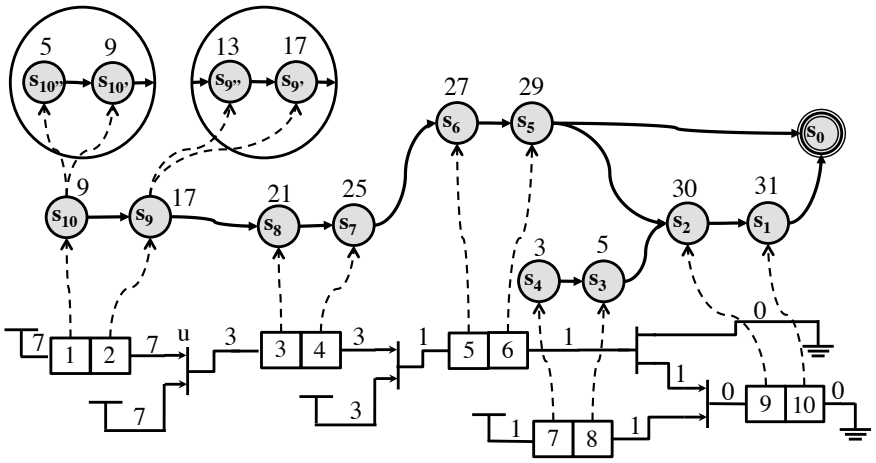

Figure 5: A network capable of tree saturation and the corresponding graph G. Channel annotations on $\mathscr{N}$ are liveness bounds, and stage annotations on G are the age bounds T_j.

j	w_j	T_j	$p_{i,j}(q_i, c)$
1	1	31	$p_{10,1} :=$ **true**
2	1	30	$p_{9,2} :=$ **true**
3	2	5	$p_{8,3} :=$ **true**
4	2	3	$p_{7,4} :=$ **true**
5	2	29	$p_{6,5} :=$ **true**
6	2	27	$p_{5,6} :=$ **true**
7	4	25	$p_{4,7} :=$ **true**
8	4	21	$p_{3,8} :=$ **true**
9’	4	17	$p_{1,9'} := u$
9”	4	13	$p_{1,9''} := \neg u$
10’	4	9	$p_{1,10'} := u$
10”	4	5	$p_{1,10''} := \neg u$

Table V: Details of refined graph G from Fig 5. The index of each stage is j. The residence time of each stage is w_j. The age bound of each stage is T_j. Formulas $p_{i,j}$ define the mapping from $\mathscr{N}$ to G.

2) If the agent’s reservation token is outstanding, packets that do not hold it are denied access to the ingress buffer, unless the ingress buffer has more than 1 free slot.
3) When a packet with a receive reservation token is granted access to the ingress buffer, the token is returned to the agent.

Receive reservations are fair with respect to packets in the ring. Whenever one packet returns the reservation token, the packet trailing it on the ring has a chance to make the reservation in the subsequent cycle. Each packet in the ring gets a turn at making a receive reservation in order.

The liveness bound of the sink is used as the basis for analytical upper bound on the latency of the ring. Because each sink’s liveness bound of R back-propagates to the ingress’ input channel, no receive reservation can be held by one packet for more than $N\lceil\frac{R}{N}\rceil$ cycles; if $R < N$ for example, then any receive reservation that is issued will be returned exactly N cycles later when the reserving packet returns. Given that there are N total slots in the ring, an analytical upper bound on the age of any packet in the ring is given by t_{ring} in Eq. 6. An upper bound on the age of any packet in the entire network is given by T_{VER} in Eq. 7. It is important to note that the analytical bound used for T_{VER} is derived from the high-level specification, but our approach allows it to be proved on the bit-level (Verilog) implementation of the network.

$$t_{ring} = 1 + N\left(1 + N\left\lceil\frac{R}{N}\right\rceil\right) \tag{6}$$

$$T_{VER} = I * (R+1) + t_{ring} \tag{7}$$

Send reservation mechanisms [16] are a counterpart to receive reservations for ensuring fairness in granting ring slots to sources, but are not addressed in this work.

A. Generating a useful stage graph G

The existence of cyclic paths in $\mathscr{N}$ complicates the construction of a stage graph, because the location of a packet within the ring is not a sufficiently precise indicator of progress. To find sub-goals for marking progress of packets circling the ring, we refine the latency lemmas by case splitting on the state of receive reservations. The refinement based on receive reservation priority in the ring is analogous to the refinement based on merge priority in Subsec. V-C.

Some notation is required for the mapping from $\mathscr{N}$ to G in the ring. For each agent $i \in [0, N-1]$, let $rsv_i \in \{\perp, 0, 1, \ldots, N-1\}$ indicate which of the N agents contains the ring slot holding the reservation token of agent i. $rsv_i = \perp$ indicates that the receive reservation is available. For each queue slot j in $\mathscr{N}$, let $dst(q_j)$ be the destination address of the packet. The destination address is chosen for each packet non-deterministically at the time of injection, and is stored with the packet in the same manner as the timestamp $t(q_j)$.

Using an $(N, I, R) = (3, 2, 1)$ ring as an example, the definition and explanation of the sub-goal for each stage in G is given in Table VII. We clarify notation by explaining the row of table VII that is denoted by $j = 4$; this row gives the age bounds and mapping conditions for stage 4. Only queue slot q_6 can ever map to stage 4, and it does so only when formula $p_{6,4}$ is true. Formula $p_{6,4}$ is true if the packet in slot q_6 has agent 2 for its destination and agent 2 has an available receive reservation. Stage 4 has a residence time of 1 cycle. The packet mapping to stage 4 must have an age of less than $4 + 2N\lceil\frac{R}{N}\rceil$.

B. Results

Our approach is evaluated on 4 different variants of the parameterized ring. The timeout for each verification experiment is 10k seconds. The results are shown in Table VIII. Strengthening the global latency property (Φ^G) by latency lemmas (Φ^L) and auxiliary invariants (Ψ) allows the global latency bound of T_{VER} to be proved by k-induction in all cases within 700 seconds using 13 frames or less. The inclusion of latency lemmas also reduces the runtime of PDR, but PDR is unable to verify 2 variants within the allotted time even with latency lemmas included. The bounds proved in all cases do not over-approximate the tightest possible bound by more

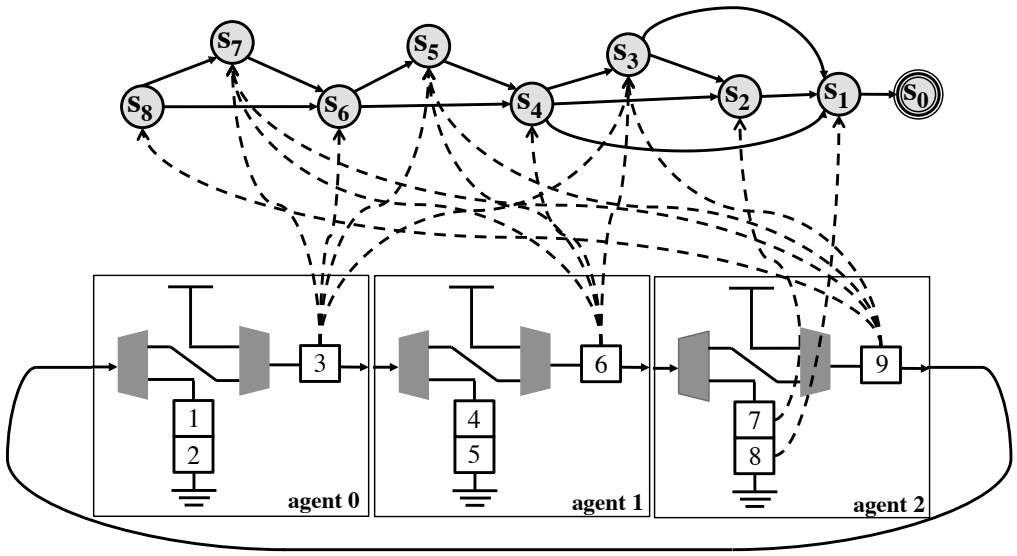

Figure 6: Ring network with $N = 3, I = 2$, and corresponding graph G. The dashed arrows drawn from $\mathscr{N}$ to G only represent the subset of all possible mappings for which the destination field $dst(q_i)$ of the slot is agent 2. Each arrow corresponds to one $p_{i,j}$ formula shown in Table VII. The shaded muxes and demuxes represent arbitration logic, including receive reservations; the logic for setting their control signals is not depicted.

j	w_j	T_j	$p_{i,j}(q_i, c)$	Meaning of Stage j
1	$R+1$	$2(R+1)+4+3N\lceil\frac{R}{N}\rceil$	$p_{8,1} :=$ **true**	head of ingress
2	$R+1$	$R+1+4+3N\lceil\frac{R}{N}\rceil$	$p_{7,2} :=$ **true**	tail of ingress
3	$N\lceil\frac{R}{N}\rceil$	$4+3N\lceil\frac{R}{N}\rceil$	$p_{9,3} := dst(q_9) = 2 \wedge rsv_2 = 2$ $p_{6,3} := dst(q_6) = 2 \wedge rsv_2 = 1$ $p_{3,3} := dst(q_3) = 2 \wedge rsv_2 = 0$	Holds reservation token
4	1	$4+2N\lceil\frac{R}{N}\rceil$	$p_{6,4} := dst(q_6) = 2 \wedge rsv_2 = \perp$	1^{st} priority for reservation token
5	$N\lceil\frac{R}{N}\rceil$	$3+2N\lceil\frac{R}{N}\rceil$	$p_{9,5} := dst(q_9) = 2 \wedge rsv_2 = 0$ $p_{6,5} := dst(q_6) = 2 \wedge rsv_2 = 2$ $p_{3,5} := dst(q_3) = 2 \wedge rsv_2 = 1$	2^{nd} priority for reservation token
6	1	$3+N\lceil\frac{R}{N}\rceil$	$p_{3,6} := dst(q_3) = 2 \wedge rsv_2 = \perp$	2^{nd} priority for reservation token
7	$N\lceil\frac{R}{N}\rceil$	$2+N\lceil\frac{R}{N}\rceil$	$p_{9,7} := dst(q_9) = 2 \wedge rsv_2 = 1$ $p_{6,7} := dst(q_6) = 2 \wedge rsv_2 = 0$ $p_{3,7} := dst(q_3) = 2 \wedge rsv_2 = 2$	3^{rd} priority for reservation token
8	1	2	$p_{9,8} := dst(q_9) = 2 \wedge rsv_2 = \perp$	3^{rd} priority for reservation token

Table VII: Graph G definitions for ring network with $N = 3, I = 2$, as shown in Fig 6. The index of each stage is j, and the residence time is w_j, and the age bound at each stage is T_j. The formulas that map $\mathscr{N}$ to G are $p_{i,j}(q_i, c)$. The formulas shown are only a subset of the formulas for each stage. In particular, they are the subset that can be true when the destination of a packet is agent 2.

than 4 cycles. Table IX shows the BMC results used to evaluate tightness (per methodology of Subsec. V-A).

- For the $(N,I,R) = (3,2,1)$ ring, the inclusion of latency lemmas gives a speedup in both PDR and k-induction. This small ring variant is the only variant for which PDR can prove the unstrengthened global latency.
- For the $(N,I,R) = (6,2,2)$ ring, T_{VER} is at most 1 cycle loose. Induction is able to prove the latency with and without latency lemmas, while PDR can only prove the latency with latency lemmas.
- For the $(N,I,R) = (6,3,4)$ Both engines require latency lemmas to prove the latency, and k-induction gives a 7x speedup over PDR.
- For the $(N,I,R) = (7,3,3)$ the only successful approach is k-induction with latency lemmas included.

VII. Related Work

One way of addressing QoS guarantees at the architectural level is to use resource reservation and contention-free routing [17]. Analysis can be performed manually, but formal verification is still useful for providing guarantees.

Network calculus [18] has been demonstrated as a useful tool for

(N,I,R)	T_{CEX}	T_{VER}	Runtime (s)	Frames	Proved	Engine	Property
$(3,2,1)$	12	17	4.89	7	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
			15.36	16	Y	kind	$\Psi \wedge \Phi^G$
			56.84	84	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
			99.20	82	Y	pdr	$\Psi \wedge \Phi^G$
			119.89	77	Y	pdr	Φ^G
$(6,2,2)$	47	49	193.53	11	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
			377.73	51	Y	kind	$\Psi \wedge \Phi^G$
			1874.63	152	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
			10000.00	77	-	pdr	$\Psi \wedge \Phi^G$
			10000.00	81	-	pdr	Φ^G
$(6,3,4)$	55	58	364.69	13	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
			4838.04	61	Y	kind	$\Psi \wedge \Phi^G$
			2555.98	149	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
			10000.00	70	-	pdr	$\Psi \wedge \Phi^G$
			10000.00	57	-	pdr	Φ^G
$(7,3,3)$	66	69	656.46	13	Y	kind	$\Phi^L \wedge \Psi \wedge \Phi^G$
			8573.73	71	Y	kind	$\Psi \wedge \Phi^G$
			5960.78	165	Y	pdr	$\Phi^L \wedge \Psi \wedge \Phi^G$
			10000.00	73	-	pdr	$\Psi \wedge \Phi^G$
			10000.00	70	-	pdr	Φ^G

Table VIII: Verifying global latency bound (Φ^G) with and without latency lemmas (Φ^L) and auxiliary invariants (Ψ) for four different variants of the parameterized ring. Values of T_{CEX} are from Tab. IX. The timeout is 10k seconds.

NoC performance analysis [19]. However, it has limited applicability and precision for networks with backpressure and complex circular message dependencies. Network calculus formalism relies on very high-level abstraction of arbiters, often modeling them as latency-rate servers. Recent abstraction-based formal approaches have been applied to NoC components [6], but they only address scalability problems arising from size of the network, rather than from proving a large latency bound, while the present work addresses the latter issue.

Several works have explored (unbounded) liveness verification of communication fabrics. The standard approach of verifying liveness using a liveness-to-safety transformation [20] does not scale to large networks in practice [11]. Alternative approaches include reducing deadlock conditions to a set of equations [11], [21], and proving liveness using the help of intermediate safety assertions [22].

Our verification approach is conceptually similar to ranking functions [23], i.e. numeric functions of model state that measure progress toward some goal. Typically, ranking functions are useful in proving termination or liveness properties, but they are also applicable for latency bounds. In fact, our stage graph can be viewed as a structural description of a ranking function for the model. Note, however, that stage graphs specify partial orders, rather than the linear orders that are typical for ranking functions.

VIII. Conclusion

This work presents a compositional approach to verifying latency bound properties of NoC designs. The key idea is to decompose the overall proof into a finite number of latency lemmas, based on the notion of stages that a packet can be in. The latency lemma approach is applied to illustrative examples and an industrial ring design. The approach facilitates inductive verification of latency bounds, and also yields significant speedup when using the PDR algorithm.

Acknowledgments. The authors thank Satrajit Chatterjee for technical discussions, and Bob Brayton and his research group for assistance with tools. This research was supported in part by the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

References

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

[2] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation–A performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[3] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerman, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *IEEE Micro*, vol. 29, no. 1, pp. 10–21, 2009.

[4] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "Quick Formal Modeling of Communication Fabrics to Enable Verification," in *High Level Design Validation and Test Workshop*, 2010.

[5] S. Chatterjee and M. Kishinevsky, "Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics," in *Computer Aided Verification*, 2010.

[6] D. E. Holcomb, B. A. Brady, and S. A. Seshia, "Abstraction-Based Performance Analysis of NoCs," *Design Automation Conference*, Jun. 2011.

[7] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Formal Methods in Computer-Aided Design*, 2000, pp. 108–125.

[8] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, 2003, pp. 1–13.

[9] A. R. Bradley, "SAT-Based Model Checking Without Unrolling," *Verification, Model Checking, and Abstract Interpretation*, 2011.

[10] N. Een and A. Mishchenko, "Efficient implementation of property directed reachability," in *Proceedings of International Workshop on Logic Synthesis*, 2011.

[11] A. Gotmanov, S. Chatterjee, and M. Kishinevsky, "Verifying Deadlock-Freedom of Communication Fabrics," *Verification, Model Checking, and Abstract Interpretation*, 2011.

[12] J. Long, S. Ray, B. Sterin, and A. Mishchenko, "Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models," *International Workshop on Design and Implementation of Formal Tools and Systems*, 2011.

[13] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verification Tool," *Computer Aided Verification*, 2010.

[14] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*.

[15] M. Mattina, G. Chrysos, and S. Felix, "Method and apparatus for synchronous unbuffered flow control of packets on a ring interconnect," *US Patent 7,539,141*, May 2009.

[16] M. Mattina, G. Chrysos, and Y. Choi, "Method and apparatus for preventing starvation in a slotted-ring network," *US Patent 7,733,898*, Jun. 2010.

[17] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: concepts, architectures, and implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 414–421, Sep. 2005.

[18] R. L. Cruz, "A calculus for network delay, part I. Network elements in isolation," *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 114–131, 1991.

[19] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit, "Modelling run-time arbitration by latency-rate servers in dataflow graphs," in *10th Workshop on Software & Compilers for Embedded Systems*. New York, NY, USA, 2007, pp. 11–22.

[20] A. Biere and C. Artho, "Liveness Checking as Safety Checking," *Electronic Notes in Theoretical Computer Science*, 2002.

[21] F. Verbeek and J. Schmaltz, "Hunting deadlocks efficiently in microarchitectural models of communication fabrics," *Formal Methods in Computer-Aided Design*, 2011.

[22] S. Ray and R. K. Brayton, "Scalable Progress Verification in Credit-Based Flow-Control Systems," *Design Automation and Test in Europe*, 2012.

[23] A. Turing, "Checking a large routine," *Conference on High Speed Automatic Calculating Machines*, 1949.

APPENDIX: BOUNDED MODEL CHECKING

The results of Table IX show the set of experiments conducted to obtain T_{CEX} for each network $\mathscr{N}$. For each $\mathscr{N}$, T_{CEX} is set to be the largest T (shown in bold) for which a counterexample is discovered by bounded model checking. All properties except for Φ^G are omitted in BMC. A linear search of T over the range $[0, T_{VER}]$ is performed; the linear search is chosen over binary search to minimize the number of trials that have a potentially expensive time-out. For each T that is tested, the BMC run ends when one of the following conditions occurs:

1) A counterexample is found to show that T is an invalid bound. The number of frames explored to find the counterexample is shown.
2) 100 Frames of BMC are completed in less than 10k seconds without finding a counterexample.
3) 10k Seconds have elapsed without finding a counterexample and without completing 100 frames of BMC. As 100 frames are not yet reached, the number of frames explored within the 10k seconds is shown.

	T	Found CEX	Runtime(s)	Frames
ex_queue.v	12	Y	0.76	18
	13	Y	0.96	19
	14	Y	1.66	21
	15	-	2285.31	100
	T	Found CEX	Runtime(s)	Frames
ex_tree_sat.v	24	Y	21.30	27
	25	Y	22.71	29
	26	Y	30.83	30
	27	-	2075.31	100
	T	Found CEX	Runtime(s)	Frames
ring_321.v	10	Y	0.93	12
	11	Y	0.61	13
	12	Y	0.68	14
	13	-	22.76	100
	T	Found CEX	Runtime(s)	Frames
ring_622.v	45	Y	47.97	48
	46	Y	37.67	49
	47	Y	44.12	50
	48	-	2253.03	100
	T	Found CEX	Runtime(s)	Frames
ring_634.v	53	Y	654.43	57
	54	Y	521.73	58
	55	Y	663.58	59
	56	-	10000.00	64
	T	Found CEX	Runtime(s)	Frames
ring_733.v	64	Y	4188.26	68
	65	Y	4174.52	69
	66	Y	3657.55	70
	67	-	10000.00	71

Table IX: Search with bounded model checking to discover T_{CEX} for each $\mathscr{N}$