UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Formal Verification and Synthesis for Quality-of-Service in On-Chip Networks

Permalink
https://escholarship.org/uc/item/38h7039c

Author
Holcomb, Daniel Edward

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Formal Verification and Synthesis for Quality-of-Service in On-Chip Networks

by

Daniel Edward Holcomb

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Andreas Kuehlmann
Professor Lee Schruben

Fall 2013

Formal Verification and Synthesis for Quality-of-Service in On-Chip Networks

Copyright 2013
by
Daniel Edward Holcomb

Abstract

Formal Verification and Synthesis for Quality-of-Service in On-Chip Networks

by

Daniel Edward Holcomb

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Quality-of-service (QoS) in on-chip communication networks has a tremendous impact on overall system performance in today's era of ever-increasing core counts. Yet, Networks-on-Chip (NoCs) are still designed and analyzed using RTL simulation, or analysis of highly abstracted models. The formal techniques that are used in core components do not find use in QoS verification due to the scale of the problems of interest, such as verifying latency bounds of hundreds of cycles.

This dissertation presents my recent work toward leveraging formal methods for NoC design and QoS verification. In particular, it addresses the problems of (1) verifying end-to-end latency bounds in a mesh network using abstraction; (2) scalable latency verification using compositional inductive proofs; and (3) optimal buffer sizing based on bounded model checking.

Contents

Contents **i**

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**
1.1 Thesis Statement 1
1.2 Thesis Contributions 2
1.3 Thesis Overview 3

I Background **4**

2 On-chip Communication Networks **5**
2.1 Motivation for NoCs 5
2.2 Defining Characteristics of an NoC 6
2.3 Quality of Service 9
2.4 Modeling NoC 9

3 Formal Verification **13**
3.1 Linear Temporal Logic (LTL) 13
3.2 Model Checking 15

4 Formal NoC Modeling **20**
4.1 Executable Microarchitectural Specifications (xMAS) 20
4.2 Implementation of Queues 27
4.3 Model of Router Core 29
4.4 Checking Latency Bound Properties on Network Model 31
4.5 Formal Traffic Model 33

II Scalable Latency Verification **39**

5 Compositional Reasoning using Traffic Abstraction **41**
5.1 Introduction 41
5.2 Preliminaries 43
5.3 Traffic Model Inference 45
5.4 Experimental Results 48
5.5 Related Work 53
5.6 Conclusion and Discussion 54

6 Compositional Proofs using Induction **58**
6.1 Introduction 58
6.2 Preliminaries 60
6.3 Formalism 63
6.4 Latency Lemmas 65
6.5 Experimental Methodology 75
6.6 Illustrative Examples 76
6.7 Non-Stallable Ring Interconnect 83
6.8 Related Work 90
6.9 Conclusion 91

III NoC Parameter Synthesis **93**

7 Buffer Sizing **94**
7.1 Introduction 94
7.2 Formal Model and Problem Definition 95
7.3 The CEBUS Approach 98
7.4 Experimental Buffer Sizing Results 101
7.5 Related Work 107
7.6 Conclusion 110

8 Conclusions and Future Work **111**

Bibliography **113**

List of Figures

2.1 Types of Interconnection Networks. 6
2.2 Arrival and Service Curves in Network Calculus. 12

4.1 The Signals Comprising an xMAS Channel. 21
4.2 The xMAS Kernel Primitives. 23
4.3 State Machine for Bounded Non-deterministic Sink Primitive. 24
4.4 Router Core. 30
4.5 Two Encodings of Flit Ages. 32
4.6 Block Diagram for a Traffic Model $\mathscr{T}$. 34
4.7 Non-deterministic FSA for Generating Consistent Flit Sequences. 36
4.8 Traffic Regulators. 38

5.1 Challenge of Performance Verification. 42
5.2 Traffic Model and Router Model to Verify. 46
5.3 Rate Constraints Imposed by Regulator 49
5.4 Convergence of Inferred Traffic Models. 50
5.5 SMT Solver Runtime versus Checked Latency Bound. 53

6.1 Credit Loop Model $\mathscr{N}$ with Stage Graph $\mathscr{G}$. 62
6.2 Recurrence Relations for Future Readiness of xMAS Primitives. 70
6.3 Comparison of Stopwatch Age Encoding and Timestamp Age Encoding. 76
6.4 Queue network with Stage Graph $\mathscr{G}$ Shown Above. 77
6.5 Runtime versus Problem Size in Queue. 78
6.6 Runtime versus Problem Size in Credit Loop. 79
6.7 Verification Runtime versus Proved Latency Bound in Credit Loop. 81
6.8 Virtual Channel Network $\mathscr{N}$ and Stage Graph $\mathscr{G}$. 82
6.9 Runtime versus Proved Latency Bound in Virtual Channel. 84
6.10 Token Bucket Traffic Metering Circuit. 84
6.11 Verification Runtime versus Proved Latency Bound in Token Bucket. 85
6.12 Parameterized Ring Network. 86
6.13 Receive Reservation State Machine. 87
6.14 Product Automaton of Receive Reservation and Occupied Ring Slot. 88
6.15 Stage Graph $\mathscr{G}$ for the 3-agent Ring Network. 88

7.1 CEBUS Procedure for Optimal Buffer Size Synthesis. 98
7.2 Modeling Element for Non-pipelined Delay. 101
7.3 Credit Logic Model for Buffer Sizing Experiment. 102
7.4 CMP Network Model for Buffer Sizing Experiment. 105

List of Tables

3.1 LTL operators 14

4.1 Data Fields of a Typical Flit. 22

5.1 Allowed Turns in XY-routing. 44
5.2 Example of a Simulation Trace. 47
5.3 Maximum Packet Density Constraints. 47
5.4 Proving Latency Bounds using TITAN Models with UCLID. 51
5.5 Proving latency bounds using TITAN models and 30-cycle BMC. 52

6.1 Comparing Verification Engines on Credit Loop. 80
6.2 Comparing Verification Engines on Virtual Channel. 83
6.3 Age Lemmas for 3-agent Ring. 89
6.4 Latency Verification Runtimes for 3-agent Ring. 90
6.5 Latency Verification Runtimes for 8-agent Ring. 91

7.1 Credit Loop Buffer Sizing Results using Circular Buffer Queue Style. 103
7.2 Credit Loop Buffer Sizing Results using Record-based Queue Style. 103
7.3 CMP Buffer Sizing Results using Circular Buffer Queue Style. 106
7.4 CMP Buffer Sizing Results using Record-based Queue Style. 107
7.5 Counterexample Traffic Patterns from BSV in CEBUS on CMP Design. 108
7.6 Buffer Sizes Produced by BSS in CEBUS for CMP Design. 109

Acknowledgments

I wish to thank my family, and especially my Mom and Dad. It would have been impossible to write this dissertation without their love and their procreation. I am lucky to have shared lunches and holidays with my sister Christine during our four common years in Berkeley, and fortunate for Julie being so gracious and understanding about deadlines. I also wish to thank my uncle William Valentine for his encouragement.

I am indebted to the many people who have taught me, and helped me to learn. Professor Sanjit A. Seshia has always been a great source of knowledge, wisdom, inspiration, and encouragement; working in his lab has been a wonderful experience. I am thankful to my dissertation committee members Professor Lee Schruben and Professor Andreas Kuehlmann for their input and support.

Many people outside of our immediate lab have helped and supported my research. The research group of Professor Li-Shiuan Peh provided the PARSEC benchmark traces that are used in Chapter 5. Professor Bob Brayton and his research group provided much assistance with tools; in particular, Alan Mishchenko for supporting ABC, and Jiang Long for supporting VeriABC. I am also indebted to many friends at Intel Strategic CAD Labs for their contributions to my research and for two immensely fun internships in Hillsboro Oregon.

I am thankful to all of my friends in the lab; to Bryan Brady, Susmit Jha, Jonathan Kotker, Alexandre Donze, Indranil Saha, and Rhishikesh Limaye for leading the way; Wenchao Li for taking the journey with me, and Daniel Fremont, Garvit Juniwal, Dorsa Sadigh, Rohit Sinha, Wei Yang Tan, Nishant Totla, and Zach Wasson for chasing me out.

I appreciate the financial support that made this work possible. This work was partially supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering (grant CCF-1139138). Additionally, 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.

Chapter 1

Introduction

Multi-processor systems historically meant multi-chip systems with high-performance links connecting them, but that has changed in the last 15 years. Technology scaling continues to enable the fulfillment of Moore's Law, with ever-increasing numbers of components integrated onto a die. This thesis topic was chosen when 45nm CMOS technology was being rolled out, and industry is now on the cusp of 14nm. Personal computers now routinely contain 8 high-performance processing cores integrated onto a single chip, and much higher core counts are found elsewhere. Inter-processor communication is no longer occurring over high-performance inter-chip links, but now occurs on-die.

While a great deal of research has gone into automated design methodologies for on-chip communication networks to connect specialized IP blocks, the design methodologies for multiprocessor networks has not kept pace. This thesis aims to advance the state-of-the art in design methodologies for on-chip communication networks in multiprocessor systems. The two specific problems that this dissertation addresses are performance verification, and synthesis of buffer sizes that guarantee performance goals are met; both of these problems are addressed using state-of-the-art formal techniques. Addressing these problems in a formal way requires work toward finding and applying appropriate models for networks and traffic, and choosing appropriate verification techniques for the chosen models.

1.1 Thesis Statement

The thesis explored in this dissertation is:

> Scalable formal methods based on abstraction and compositionality, adapted to the domain of communication networks, allow performance properties to be addressed using model checking.

The formal techniques that are used in core components have not traditionally found use in performance verification due to the scale of the problems of interest, such as verifying latency bounds

of hundreds of cycles. In this dissertation, scalable model checking of communication networks is achieved using techniques such as abstraction and invariant strengthening, but applied in a way that is well suited to the problems that arise in networks. Bringing model checking into performance analysis avoids the use of highly abstracted models, the creation of which is inherently error prone. Through the application of model checking, performance properties can be proved with the same highest level of assurance as formally verified functional properties are today.

1.2 Thesis Contributions

The contributions of this work are in devising domain-specific techniques that allow formal verification to scale up to the problems arising in NoC performance verification. The novelty comes not from the techniques, but instead from their adaptation to the problems at hand. Experimental data is provided to evaluate each approach, and to highlight promising future directions and apparent dead ends. Among performance properties verified, latency bounds play a role in all contributions of the dissertation.

The verification of NoC latency bounds is an important problem to solve because network latency can be a bottleneck for overall system performance. For example, consider the performance impact of traffic sent from a processing core to a memory controller and back. A computation on the processor may be stalled while waiting for data from memory, and the NoC is responsible for delivering the request to the memory controller and returning data to the core. A large latency for this traffic significantly degrades overall performance. Proving a worst-case bound on latency enables designers to decide whether the bound is acceptable or whether the design must be modified for a smaller latency. However, the myriad types of traffic in a CMP make it challenging to identify the conditions that induce worst-case latency, and this limits the confidence of any bound determined through simulation alone. Formal verification addresses this problem by providing guaranteed worst-case latency bounds.

Latency bounds are verified in this dissertation using model checking. Unfortunately, current model checking engines are unable to scale up to handle problems as large as those arising in naïve formulations of latency verification. Two factors challenge scalability of model checking latency properties. Firstly, in a SAT-based model checker, the number of variables generally scales as the product of the number of unrollings and the number of state bits in the model. Secondly, the number of unrollings is proportional to the worst-case latency, and this is unacceptably large. Chapter 5 addresses these challenges using abstraction, while Chapter 6 uses invariant strengthening for inductive verification, and demonstrates that the technique reduces verification runtime by 2 orders-of-magnitude. An additional contribution of this dissertation is the use of model checking for synthesizing latency-guaranteeing buffer sizes in Chapter 7.

1.3 Thesis Overview

This dissertation is split into two parts: Part I gives motivation and background material about techniques and models used; Part II gives two approaches for latency verification; Part III gives an approach for synthesis, and more specifically for buffer sizing. The chapter topics are sketched here, with a description of how myself and collaborators contributed to the novel material presented in Chapters 5, 6, and 7:

- Part 1 (Background)
  - **Chapter 2:** introduces networks-on-chip, why they are of interest, the need for quality of service, and typical approaches for modeling and analysis.
  - **Chapter 3:** presents an overview of the current state-of-the art in formal verification.
  - **Chapter 4:** introduces the modeling approach that is employed for the novel contributions of the subsequent chapters.
- Part 2 (Scalable Latency Verification)
  - **Chapter 5:** presents work on traffic abstraction for latency verification. This work was done in collaboration with Bryan Brady and Sanjit Seshia at UC Berkeley, and published at the 2011 Design Automation Conference (DAC) [50].
  - **Chapter 6:** presents work on compositional verification using invariant strengthening. This work was a collaboration with Sanjit, myself, and many people at Intel Strategic CAD Lab including Alexander Gotmanov, Mike Kishinevsky, Satrajit Chatterjee, and Yuriy Viktorov. An early version of this chapter was published in the 2012 Formal Methods and Models for Codesign Conference (MEMOCODE) [51] with Alexander, Mike, and Sanjit as co-authors. While the ideas in this chapter were developed both at Berkeley and during an internship at Intel, the published data and that appearing in this dissertation are from my implementation of the work while at UC Berkeley. The work appearing in Chapter 6 is extended and improved relative to the version that appeared at MEMOCODE, and is currently under review as a journal paper.
- Part 3 (NoC Parameter Synthesis)
  - **Chapter 7:** presents work on buffer sizing done in collaboration with Bryan Brady and Sanjit Seshia; this work was originally published at the 2011 Design Automation and Test in Europe Conference (DATE) [14]. Bryan did early work on the implementation, and I later reimplemented it; Bryan and I were joint first authors on the paper.

Part I

Background

Chapter 2

On-chip Communication Networks

According to Moore's law, technology scaling enables an integration of an ever-increasing number of components onto each chip. With the increasing number of functional blocks on a chip, the importance of the communication infrastructure that connects them rivals that of the individual blocks themselves. On chip networks emerged in the early 2000s as an efficient mechanism for communication between a large number of components. Network-on-chip (NoC) is a term introduced by Dally and Towles [37] to describe this paradigm for communication within large many-core, system-on-a-chip designs.

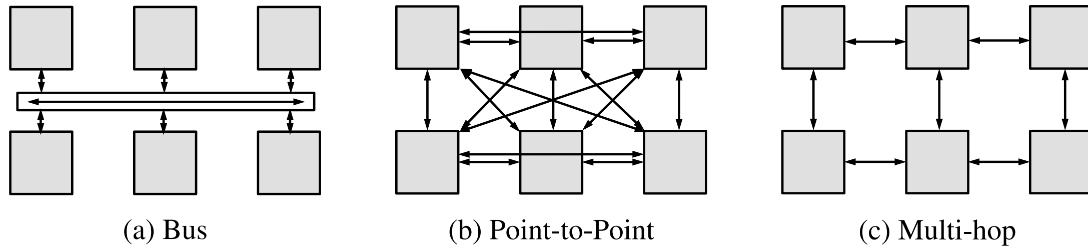
2.1 Motivation for NoCs

Different types of communication fabrics can be used to interconnect a number of processing elements, where each element is a processor core, memory controller, or other functional block found in a modern VLSI design. The appropriate choice of communication fabric for a given design depends on a variety of factors including bandwidth and latency requirements. The advantages and disadvantages of various interconnection schemes are outlined in the following paragraphs. Whereas buses or point-to-point links were used historically, the dominant paradigm is now multi-hop communication using a series of on chip routers. A detailed quantitative comparison of these interconnection schemes is given by Lee *et al.* [65].

Bus: as shown in Fig.2.1a, a bus is a communication network one in which all agents communicate over a single shared resource (the bus). Each agent has a interface to the bus, and through this interface it can inject and receive traffic. Area cost in a bus is linear in the number of connected agents n. The significant drawback of a bus interconnect is performance. Due to the global sharing of the bus resource, only two agents can communicate concurrently. Bus variants exists which can support multiple concurrent communicating agents, but such schemes are essentially just multiple redundant buses and do not avoid the problem of shared global resources which make buses undesirable for a system with a large number of agents.

Point-to-point: as shown in Fig.2.1b, a point-to-point network, has a dedicated link between every pair of communicating agents. Each of the n agents arbitrates among n inputs. Area cost in a point-to-point network is quadratic in n because all pairs of agents have dedicated connections. Furthermore, the long dedicated links are not simple metal wires, since combinational repeaters are required to linearize the delay of the long wires, or else sequential repeaters to pipeline the link for increased bandwidth. Having such a high number of links constitutes an inefficient use of resources, since it is not possible to use all of the links simultaneously. Point-to-point networks with properly designed links perform very well, but are infeasible due to the high area costs.

Multi-hop: as shown Fig.2.1c, a multi-hop network is a configuration in which traffic travels from one agent to another by making a series of hops along a path of intermediate routers. Nearly all NoCs are multi-hop networks. Arbitration is distributed and simple, as each router only arbitrates between a small number of neighboring routers. The area cost is linear in n because additional agent requires exactly one additional corresponding router. Yet, contrary to the bus, there is no single globally shared link that constrains the overall communication. NoCs are therefore a sort of compromise between buses and point-to-point networks, where area is linear in n but performance is not constrained by a single bus.



(a) Bus (b) Point-to-Point (c) Multi-hop

Figure 2.1: **Types of Interconnection Networks.** Each shaded block is a an agent comprising a processor core or memory controller, along with associated logic for interfacing it to communication links which are drawn as arrows.

NoCs for heterogeneous designs tend to be parameterized architectures that are customized according to the communication needs of each IP block, whereas NoCs for chip-multiprocessors (CMPs) tend to be uniform on account of the homogeneity of processing elements. Examples of highly-customizable heterogeneous NoCs include ASOC [66], AEthereal [41], Xpipes [33], NOSTRUM [72], SPIN [49], and QNoC [10]. Examples of CMP NoCs include the Tilera TILE64 [105], STI Cell BE [25], Intel Larrabee [92], and Intel Terascale [52].

2.2 Defining Characteristics of an NoC

Nearly all NoCs share a set of common characteristics and are based on multi-hop routing to avoid both the quadratic cost of dedicated point-to-point links and the bandwidth limitations of globally shared buses. All NoC architectures consist of a network of interconnected agents, where each agent comprises a router, and a processing element such as a processor core, memory controller, or

other functional unit. Each router interfaces to its processing element through a network interface port, and to the routers of a small subset of other routers. Packet-switching is used to route data packets among routers from a source node to a destination node. At the physical level, a packet is transferred across a channel as some number of "flits", or flow-control-units. A head flit carries information about a packet's destination address, and is followed by some number of payload-carrying body flits and a single tail flit. Despite the many significant commonalities of all modern NoCs, they also vary in a few significant ways. The following subsections highlight briefly a few characteristics that vary across NoC designs.

Topology

A defining characteristic of an NoC is its topology. In heterogeneous networks, where bandwidth requirements vary widely for each node, specialized topologies can be synthesized from communication requirements [82]. Chip multiprocessor NoCs, which tend to have a large number of identical processing cores and a few memory controllers, tend to use a variety of more regular topologies.

A mesh is a topology in which each router exchanges traffic with its neighbors. In an n-dimensional mesh, each router connects to $2n$ neighbors. Because integrated circuits are planar, 2-dimensional meshes are common, with each node connecting to its 4 neighbors, except for the edge nodes which connect to 3 neighbors, and the corners which connect to 2 neighbors. Examples of industrial 2-dimensional meshes are the Intel Terascale [52] and Tilera TILE64^{TM} processors [105].

A torus topology is similar to a mesh, except that it lacks edges. A 2-dimensional torus can be constructed from a 2-dimensional mesh by connecting the left edge to the right edge, and bottom edge to top edge. Hence, in a 2-dimensional torus every node has 4 neighbors, and in a 3-dimensional torus every node has 6 neighbors (i.e. N,S,E,W planar neighbors as well as the neighbors above and below). A torus is a common topology for communication between the many separate chips in supercomputers; for example, IBM Blue Gene uses a 3-dimensional torus [2]. However, torus networks are not well-suited to single-chip many-core designs because of the long wires that connect the nodes on opposite edges of the die. A folded torus is proposed to solve this problem for NoCs [35, 79], but does not appear to be used in any common industrial designs.

Rings are another common topology for NoCs. Rings are used in designs with relatively modest core counts such the 48-core Intel Larrabee processor [92] or 8-core STI Cell BE [25]. To scale beyond tens of rings, hierarchical rings can be used. Various configurations of hierarchical rings are competitive with meshes up to around 128 [88] or 256 [44] cores.

Flow Control

Flow control describes mechanisms for deciding how to allocate channels and buffers to packets. Flow control has a huge performance impact, and a wide variety of optimizations exist to maximize performance. A thorough discussion of flow control options in NoCs is given in the dissertation of Peh [81], and in this section only basic mechanisms are described.

At a high level, flow control can be classified as either store-and-forward or wormhole. Store and forward networks wait until all flits of a packet are received before forwarding any to the next node, while wormhole networks forward the individual flits of a packet as they arrive. Wormhole networks can have lower latency and use smaller buffers than store and forward [74], but a single packet can be strung out across the network blocking the forward progress of many other traffic flows. This type of blocking is known as head-of-line blocking, and performance analysis of head-of-line blocking is a difficult problem [3]. In some cases, head-of-line blocking can lead to deadlocks.

Virtual channels are a mechanism for avoiding head-of-line blocking. Virtual channels were initially introduced as a mechanism for deadlock avoidance in wormhole networks [36], and later also shown to have desirable performance properties [34]. With virtual channels a single physical channel is shared between multiple logical (i.e. virtual) channels. Each of the virtual channels has its own buffer, but only one virtual channel can be active in a given cycle on account of their sharing of a physical channel. When one of the virtual channels is blocked, another virtual channel can proceed, thus eliminating head-of-line blocking. Some additional logic is required to control how the physical channel is multiplexed to the virtual channels, but virtual channels are ubiquitous in CMP NoCs.

A router will only send traffic across a link to a neighbor if that neighbor is ready to accept the traffic. This readiness is determined either through the use of a credit system or on/off signaling. With credits, the sending router keeps track of the number of open buffer slots in the ingress queue of the receiving router, and halts when the supply of credits is exhausted, as this indicates a full queue. The router only begins sending flits again once it receives a credit to indicate that a slot has now become available. An alternative to credit-based flow control is on/off signaling, where a router does not keep track of the number of free slots, but instead simply keeps sending packets until the receiving router sends a signal to stop.

Routing

Routing in a mesh NoC can be classified as oblivious or adaptive. Oblivious routers use deterministic paths for packets that are independent of congestion, whereas adaptive routing steers packets around congestion. The examples in this dissertation use oblivious routing, and more specifically XY (dimension-order) routing. In XY routing, a packet reaches its destination by first being routed to the appropriate column along the x-dimension, and then being routed to the appropriate row in the y-dimension. XY-routing is guaranteed not to introduce deadlock or livelock [38].

Architectural Parameters

Once the topology, flow control and routing of an NoC are decided, remaining parameters such as buffer depths and channel widths remain to be assigned. The depth of each physical or virtual channel buffer is the maximum number of flits that can be stored in it; deeper buffers improve performance, but increase the area cost of the NoC [53][55]. Similarly, wider channels (and hence

wider flits) allow more data to be transferred per cycle across a channel, but require wider buffers to store the flits and therefore also lead to increased area costs. An overview of these considerations in NoC design is given by Ogras *et al.* [77].

2.3 Quality of Service

A correctly-designed NoC should ensure that performance properties are satisfied. First and foremost, the NoC should never deadlock or livelock. But guaranteeing that packets reach their destinations is not a sufficient indicator of correctness in an NoC. To ensure that programs executing on an NoC will perform well, the network must give guarantees on performance of network traffic.

Quality-of-Service (QoS) refers to performance guarantees that are provided by the NoC to users. Some QoS metrics of interest are minimum throughput offered, end-to-end latency bounds, and maximum jitter. Minimum throughput is a lower bound on the number of bytes per second that a source can inject under arbitrary achievable network conditions. Maximum latency is the largest allowable time between when a packet is injected at the network interface of its source agent, and when it is ejected from the network interface of its destination. Jitter refers to irregularity in delay and is especially problematic in streaming traffic, where the existing of jitter may require more conservative buffering. This dissertation focuses primarily on latency bounds. Satisfaction of QoS requirements depends on spatial and temporal properties of injected traffic, so in some cases QoS guarantees are conditioned on traffic constraints.

Packets traveling through an NoC are used for many different purposes; for example, packets in an NoC may be urgent control data, real-time data with deadline requirements, reads or writes between processors and memory controllers, block transfers of large data, and so on. Some NoCs therefore provide differentiated service classes, where high priority traffic is given guaranteed service, and other traffic is given best-effort service. Best effort service refers to the manner in which packets without service guarantees are handled, and the performance of such traffic depends significantly on the amount of competing traffic in the network. The focus in this dissertation is on QoS bounds in NoCs with undifferentiated service, where all traffic can be viewed as best-effort.

2.4 Modeling NoC

There are many different ways to model an NoC, and the right model depends on the purpose for which it will be used. While modeling approaches such as GeNoC are tailored to proving functional correctness and liveness [101, 103], the focus in this dissertation is on models more suitable for performance verification. This section broadly splits modeling into cycle-accurate hardware descriptions, and the variety of analytical models. Hardware description languages provide highest accuracy, but can be more detailed than is necessary for performance analysis because they don't hide any irrelevant details of the network behavior. Furthermore, full hardware descriptions may not exist until late stages of design, potentially making them infeasible for design space exploration. Analytical models are fast and typically admit closed-form solutions, but may not capture

all the intricacies of real system behavior, and their applicability often only holds under specific assumptions.

Hardware Description Languages

Before fabrication, every digital circuit first exists as a register transfer level (RTL) model in some hardware description language such as Verilog or VHDL. To minimize the number of logic bugs that reach silicon, extensive functional validation through simulation and formal verification are performed on the RTL model. Performance verification is accomplished by monitoring the timing in simulation of interesting events such as packet injection and ejection times. Clearly the performance observations depend closely on the inputs that are applied, and so finding appropriate inputs is a significant challenge.

To perform simulations, the inputs to the RTL model of the system are driven from a testbench, and the state and output are monitored for violated assertions or unexpected outputs. The execution of the model can be either cycle-driven or event-driven. In a cycle-driven simulator, all state variables are updated in each cycle, and in event-driven simulation state variables are only updated on-demand when one of their inputs changes.

Register Transfer Level Model

The canonical example of a simulatable system model of an NoC is the RTL description of the hardware. An RTL description of a digital circuit is system of Boolean state variables often grouped into bit-vector words, and transition relations. Each state variable is computed using a deterministic function over the previous state and the inputs. The RTL representation of a NoC is highly accurate, and captures all logic-level details of a design. Electrical interactions, timing slack, and other non-logical details are abstracted away in RTL.

Testbench Environment Model

When evaluating performance using a simulatable model, the inputs supplied by the testbench can have a significant impact on the measured performance. In choosing input patterns to apply there exists a tradeoff between having models that are highly general or highly representative of the traffic produced by certain applications of interest.

The most general inputs are those coming from synthetic traffic generators. Synthetic generators can create sequences of non-repeating inputs according some basic assumptions on the destination of packets and the rate at which they are injected. Different distributions such as uniform random and hot-spot are commonly used [38]. Results of performance analysis may vary wildly across different types of synthetic traffic generators [60], and therefore to obtain meaningful performance measurements it is important to use highly representative traffic.

A highly representative set of traffic patterns can be obtained by running applications of interest on architectural simulators and recording the inter-process communication [80]. This provides

a trace of highly relevant inputs that can be simulated. However, the process is very involved compared to using synthetic traffic generators, the finite traces generated in this manner are not very comprehensive.

A compromise between synthetic traffic and architecture-level simulation traces is to use synthetic traffic generators with parameters that are tuned to produce traffic resembling the traces; this synthetic traffic can then be viewed as a generalization of the original traces. Varatkar and Marculescu propose a model for MPEG-2 traffic that models self-similarity using a Hurst parameter [100]. Later, Soteriou *et al.* [96] show that a three parameter model leads to accurate representation of a variety of applications; the three parameters used are a Hurst exponent for temporal burstiness, a spatial hop distribution parameter, and a spatial injection distribution parameter.

Analytical Models

Instead of cycle-accurate RTL models, a second way to reason about performance is through the use of analytical models [78]. Analytical models typically lack state variables, transition relations and synchronous semantics. They instead aim to describe various performance metrics as direct functions of traffic and configuration parameters.

Network Calculus

Network calculus is an analytical technique for reasoning about classes of discrete event systems using max-plus algebra. Network calculus was first introduced by Cruz for modeling both traffic and network elements [31][32], and relies on bounding envelopes for traffic. Incoming traffic to an element such as a queue is upper bounded by an envelop function known as an arrival curve. If function $R(t)$ represents the number of packets to arrive before time t, then R is bounded by arrival curve a if $R(t) - R(s) \leq a(t-s)$ for all t and s. A common arrival curve is that of a leaky-bucket [31], with arrival curve $a(x) = \sigma + \rho x$ having a burst-size of σ and long-time average of ρ packets per cycle. The draining of packets through an element is lower bounded by a service curve, such as a latency-rate service curve [97] in which the number of packet served from a backlog of waiting packets in any time window of duration x is lower bounded by $s(x) = (x - \gamma) * \kappa$. Various network elements transform and shape arrival and service curves. If arriving and served traffic are bounded by some arrival and service curves a and s, then closed-form analysis can be used to compute from a and s the upper bounds on queue occupancy and latency (Fig. 2.2).

While the advantage of network calculus is the existence of closed-form analysis, it is not well suited to many types of traffic. Burst-sizes in best effort NoC traffic can be very large, and therefore only enveloped by arrival curves that are too conservative to yield meaningful results. Furthermore, self-similar traffic cannot be bounded by any deterministic arrival curve. Work by Qian *et al.* [87] shows that it is possible to derive from self similar traffic an arrival curve that envelopes no less than some known percentage of traffic, but not all of it. From this, probabilistic latency bounds can be obtained using the closed-form network calculus formulation

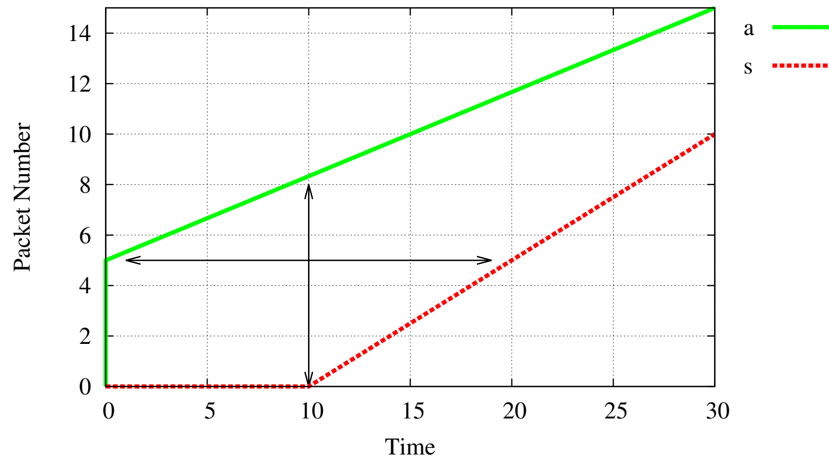


Figure 2.2: **Arrival and Service Curves in Network Calculus.** Arrival curve is a leaky-bucket with $(\sigma, \rho) = (5, 1/3)$. Service curve is latency-rate server with $(\gamma, \kappa) = (10, 1/2)$. For a queue with this arrival curve on input, and service curve at output, the horizontal arrow shows the maximum latency of 20 through the queue, and vertical arrow shows the maximum queue occupancy of 9.

Queueing Theory

Queueing theory provides another analytical framework for reasoning about QoS. Whereas network calculus reasons about worst-case behaviors, queueing theory applies to average-case. Under the assumption of Poisson arrival of header flits [76] in each buffer, queueing theory gives closed-form expressions for average buffer utilization, average packet latency per flow, and maximum network throughput. A criticism of queueing theory models for NoCs is given by Bogdan *et al.* [9].

Preview of xMAS Formalism

This dissertation uses extensively a modeling formalism known as xMAS. Full description of xMAS is delayed until Chapter 4, but for now it suffices to remark that xMAS is akin to a simplified form of RTL. While not analytical models in the sense of the previous section, xMAS models are simple enough enough to be reasoned about formally.

Chapter 3

Formal Verification

All designed systems are created with the goal of correctly implementing designer intent, but correctness is more important in some systems than in others. In modern VLSI circuits, the costs of the photolithographic masks for a design are in the millions of dollars, and therefore significant effort is spent to ensure correctness before fabrication. In such situations, formal verification techniques are used to prove that a model of a system satisfies its intended specifications.

An obvious but weak assurance of correctness is obtained from simulation. In simulation, selected inputs are applied, the system model is executed, and properties over state variables and outputs are monitored to check for violation of specifications. Simulation is a sound bug finder, meaning that if a property fails, then the system model is indeed incorrect. However, simulation is woefully incomplete, meaning that no guarantee of correctness can be given if the simulation is free of failures. If no failures are observed in simulation, it is still possible that the model can fail, but that the inputs applied were not the right ones to cause the failure.

Formal verification techniques, in contrast to simulation, can guarantee that a model always satisfies some desirable property. The two main formal verification techniques in use today are model checking and theorem proving; there are significant overlaps between these two techniques, and they are often used in concert. The applicability of formal verification is limited by its cost, as significant human intuition and computationally expensive algorithms are generally required to formally verify a model. However, the cost of formal verification is justified in applications such as VLSI circuits where the cost of failure is also high.

3.1 Linear Temporal Logic (LTL)

Correctness of a system model is always defined in terms of properties that the system must satisfy, or properties that it must not satisfy. One way to specify such properties is using Linear Temporal Logic (LTL). Linear Temporal Logic (LTL) was first used for reasoning about systems by Pneuli in 1977 [84], and has found wide application in the time since. Properties in LTL are specified with respect to execution paths, where an execution path π is a sequence of states $s_0, s_1, s_2, \ldots$.

true	positive literal
false	negative literal
p	atomic proposition
¬ p	negation
p ∧ q	conjunction
p ∨ q	disjunction
p $\implies$ q	implication
F	Future
G	Globally
X	Next
U	Until

Table 3.1: **LTL operators** LTL combines propositional logic operators with temporal operators.

The basic LTL operators are given in Tab. 3.1. In each state along a path, an atomic proposition p evaluates to **true** or **false**; a state s_i where p holds can be written as $p(s_i)$, and a state where it does not hold as $\neg p(s_i)$. The logical operators $\neg, \wedge, \vee, \implies$ are applied over propositions to produce **true** or **false** in each state. The temporal operators **F**, **G**, **X**, and **U** evaluate to **true** or **false** for a state in consideration of the path through the state.

Safety and Liveness Properties

A safety property asserts that some bad condition (e.g. p) must never happen (e.g. $\mathbf{GF}\neg p$), whereas a liveness property asserts that something good (e.g. q) must eventually happen (e.g. $\mathbf{GF}q$). A safety property is disproved by a finite length witness (counterexample) that ends in a non-p (bad) state. A liveness property has no finite-length witness because any finite length trace that does not reach a p state may yet reach a p state in the future. A liveness checking problem can be encoded as safety checking [6], but this causes a blow-up in the model.

Bounded Liveness Properties

A liveness property (e.g. $\mathbf{GF}p$) specifies only that p must hold in some state subsequent to the current state on a path, but does not specify how many other states are visited before a state satisfying p. This makes liveness insufficient for specifying a QoS latency property, where it is necessary to specify that some action (e.g. packet reaches destination), occurs not just in the future, but within a specific number of cycles in the future.

Bounded liveness properties are better suited to checking latencies. A bounded liveness property is a liveness property where a proposition must hold in the future within a specified number of states. A bounded liveness property is written as $\mathbf{F}^{<x}p$ to specify that p must occur no more than x states into the future. A bounded liveness property is really a safety property, and is equivalent to

Eq. 3.1, where the final property in the disjunction has n nestings of the **X** operator. Kupferman *et al.* [59] describe a system where all **F** properties are bounded as a "prompt" system.

$$\mathbf{F}^{<n} p \equiv p \vee \mathbf{X}p \vee \mathbf{X}(\mathbf{X}p) \vee \mathbf{X}(\mathbf{X}(\mathbf{X}p)) \vee \dots \quad (3.1)$$

3.2 Model Checking

Model checking performs verification by exhaustively checking whether a state-transition graph (a model) satisfies a property. The model is $M = (S, I, R, L)$. The set of states of the model are S, the initial states are $I \subseteq S$, the transition relation $R \subseteq S \times S$, and a labeling $L : S \mapsto 2^p$, where p is the number of propositions that evaluate to **true** or **false** in each state of the model.

Model checking suffers from the so-called state explosion problem. This state explosion problem refers to the fact that the number of model states is exponential in the number of Boolean variables. Explicit state model checkers are based on graph-traversal of the model states, and must keep track of visited states. This quickly becomes infeasible as the size of the model grows. Straightforward implementations of explicit-state model checking are therefore unable to scale to models with millions of states or more, as can arise from just 20 or so Boolean variables. Symbolic model checking overcomes state explosion by avoiding explicit graph traversal.

Initial attempts at symbolic model checking [19] are based on Binary Decision Diagrams (BDDs) [17]. Binary decision diagrams are compact representations of sets, and in symbolic model checking BDDs are used to efficiently encode the Boolean formulas for sets of reachable states, and for the model transition relation. A BDD representation of the states that are immediately reachable from a set of current states is computed by direct manipulation of the BDDs for the current states and transition relation. BDD-based symbolic model checking enabled for the first time the checking of properties on systems with up to 10^{20} states [19]. BDD-based model checking, while still used, is limited by the fact that the compactness of BDD-based representation of formulas depends on the variable ordering used in their construction.

SAT-based symbolic model checking has supplanted BDD-based as the dominant approach. SAT-based symbolic model checking avoids variable ordering issues because there is no significance to variable ordering in the conjunctive normal form used by SAT-solvers. Due to the nature of SAT-solvers, most time is spent on the important parts of the problem, where solvers work to resolve conflicts. SAT-based model checking is able to handle larger problems than BDD-based model checking. SAT-based formulation of various model checking algorithms are given in the following sections, and a comprehensive survey of SAT-based model checking is given by Prasad *et al.* [86].

Model checking can be applied to both Boolean formulas and formulas in various theories; this dissertation makes use of both of these variants of model checking. Chapters 5 and 7 use SMT-based model checking as will be explained in Sec. 3.2, and Chapter 6 uses SAT-based model checking.

Boolean Satisfiability Problem

A significant advantage to SAT-based symbolic model checking is that it leverages the tremendous performance improvements in SAT-solvers over the last decade. The Boolean satisfiability problem is, given an expression using $\wedge, \vee, \neg$ over Boolean variables, to find an assignment to the variables that causes the expression to evaluate to **true**, or else to determine with certainty that no such assignment exists. Boolean satisfiability problems are typically written in conjunctive normal form (CNF), as a conjunction of disjunctive clauses; 3-SAT refers SAT where each disjunctive clause has up to 3 variables. The 3-SAT problem is NP-complete [30], meaning that SAT-solvers can be used to solve the variety of interesting problems that are NP-complete [56]; for this reason SAT solving has been the subject of a large amount of research.

SAT-solvers are now routinely able to solve SAT problems with thousands or millions of variables. Most SAT-solvers are based on the DPLL procedure [40, 39] for resolution and backtracking. The capabilities of SAT-solvers showed a dramatic improvement in the last 15 years. The solver GRASP [69] advanced conflict analysis to enable non-chronological backtracking. The later solver Chaff [73] sped up constraint propagation using watched literals. Most modern solvers make use of the techniques pioneered by GRASP and Chaff. This dissertation uses extensively the SAT-solver MiniSat [43].

Bounded Model Checking

SAT-based bounded model checking (BMC) was first introduced in 1999 by Biere *et al.* [7]. Given a model M, BMC addresses the problem of whether it is possible within k frames from the initial state, to falsify a property p. BMC attempts to falsify p by finding a counterexample, which is an input assignment that causes Eq. 3.2 to be satisfied. BMC finds the shortest possible counterexample by first using a small value of k and then for as long as Eq. 3.2 cannot be satisfied, unrolling the problem for an additional cycle by increasing k, until the formula is satisfied (i.e. the property is disproved) or some preset resource limit (e.g. memory or runtime) is exceeded.

In general, if a safety property cannot be disproved by BMC within allotted resource limits, then the validity of the property is unknown. Given that the model has finitely many states, it is of course possible for BMC to be complete when k is sufficiently large, and the value of k required for this is known at the completeness threshold [27]. In general there is no assumption of completeness when using BMC. Due to its incompleteness, BMC is used primarily for bug-finding, and not for proving properties.

$$BMC(M,k,p) \equiv I(s_0) \wedge \left(\bigwedge_{i=0 \ldots k-1} R(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=0 \ldots k} \neg p(s_i) \right) \tag{3.2}$$

Induction

Informally, induction tries to prove a property by assuming that it has always held previously, and then using that fact and the model transition relation to show that it holds in all reachable next states. If it can be shown that no transition exists from state satisfying p to one that does not satisfy p, then p is proved as long as it holds in the initial state. A property p that can be proved in this way is called an inductive invariant.

Simple Induction

Inductive proof of a safety property p is a two-step process comprising a base case and an inductive step. The base case checks that the property holds for all initial states (denoted s_0). The inductive step checks that, if the property holds in some arbitrary state s_k, then it must hold in all s_{k+1} that are reachable next states. If INDBASE and INDSTEP (Eq. 3.3) are both unsatisfiable, then all initial states satisfy p, and all states reachable from p states are also p states; this constitutes a deductive proof that $M \models p$.

Induction is attractive proof method because it does not require an unrolling of the transition relation past two frames. This translates to a small SAT problem, and ultimately one that is likely to have a fast runtime for a SAT-solver. Unfortunately many properties cannot be proved using simple induction, due largely to the general initial state in INDSTEP. The inductive step begins from an arbitrary state s_k, with the only constraint being that s_k is a p state. This allows for the inclusion as s_k of many unreachable states that happen to be p states, and progression of the model from unreachable p states is often erratic because it is anyway not a meaningful model trajectory.

$$\begin{aligned} IndBase(M,p) &\equiv \quad I(s_0) \wedge \neg p(s_0) \\ IndStep(M,p) &\equiv \quad p(s_k) \wedge R(s_k, s_{k+1}) \wedge \neg p(s_{k+1}) \end{aligned} \tag{3.3}$$

K-Induction

Properties that cannot be proved using simple induction require strengthening, and one way of strengthening an inductive property is to use k-induction [8, 94]. K-induction is stronger than simple induction due to its handling of the inductive step. Where the inductive step in simple induction assumes that a property holds in a single state, the equivalent step in k-induction assumes that a property holds in k consecutive states. The base case in k-induction (KINDBASE) is equivalent to $BMC(M,k,p)$. Property p is proved using k-induction if KINDBASE and KINDSTEP (Eq. 3.4) are both unsatisfiable.

$$KIndBase(M,k,p) \equiv \quad I(s_0) \wedge \left(\bigwedge_{i=0...k-1} R(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=0...k} \neg p(s_i) \right)$$

$$KIndStep(M,k,p) \equiv \quad \left(\bigwedge_{i=0...k} R(s_i, s_{i+1}) \right) \wedge \left(\bigwedge_{i=0...k} p(s_i) \right) \wedge \neg p(s_{k+1}) \tag{3.4}$$

If unique states are enforced in k-induction, then the approach constitutes a complete proof engine if the value of k is not restricted. The completeness comes from the fact that ultimately there are finitely many unique p states to visit, and once the induction depth is sufficient to reach all p states, if still no transition is possible to a non-p state, then p is proved. However, the depth required to reach all p states can be large, and therefore k-induction is not typically used for its completeness.

Invariant Strengthening

Invariant strengthening plays an important role in inductive and k-inductive proofs. The idea behind invariant strengthening is to prove a property p by proving a stronger property p' that implies p but is easier to prove. Consider for example a model comprising a 10-bit variable $x[9:0]$ where the transition relation increments x in every cycle, except that x=10'd15[1] is followed by x=10'd0 instead of x=10'd16. Now assume that the goal is to prove a property $p := x[9:0] \neq$ 10'd500. All states are p states except for the single state where x=10'd500. The state x=10'd17 is a p state, and the next 485 states that follow are all p states, until reaching the non p state x=10'd500. So the value of k that would be needed to prove p would be at least 485. However, if the property $p' := x[9:4] =$6'd0 is proved, then the induction depth required to prove it will be greatly reduced. Since $p' \implies p$, proving p' provides an easier way to prove the original property p.

IC3 / Property-Directed Reachability

The IC3 algorithm developed by Aaron Bradley is considered a recent breakthrough in model checking [12]. An efficient AIG-based implementation of IC3 that is used in this dissertation is referred to as property directed reachability (PDR) [42]. IC3 is based on automatically finding and refuting candidate incremental inductive invariants. Ultimately, IC3 terminates by finding invariants that over-approximate reachable states but are precise enough to show that non-p states are unreachable.

Model Checking using Satisfiability Modulo Theories

An alternative to verifying Boolean formulas with SAT solvers is to use Satisfiability Modulo Theories (SMT) solvers. SMT solvers extend SAT solvers by replacing Boolean variables with predicates in some underlying theory or combination of theories.

[1] In notation borrowed from Verilog, 10'd15 is used here to represent a bit-vector of width 10 that has the decimal equivalent of 15.

SMT solvers can be categorized as eager or lazy. An eager solver [62, 93, 4] encodes an SMT problem into an equisatisfiable SAT instance; all inference is done by the SAT solver in the problem's Boolean representation. A lazy SMT solver is based on a generalized DPLL procedure [75], with propagation and learning according to the underlying theory.

Some theories used that are relevant to this work are bit-vectors and linear arithmetic. Bit-vectors are interesting because they match hardware implementations which are naturally restricted to finite-size data. Bit-vector SMT problems can be mapped to Boolean SAT problems by bit-blasting, a process in which each n-bit bit-vector variable is replaced by n Boolean variables.

The majority of SMT solving in this dissertation is performed using the UCLID verification system [18, 62]. UCLID decides properties in first order logic that contain equality, uninterpreted functions, constrained lambda expressions, integer linear arithmetic, and bit-vector arithmetic. The lambda expressions in UCLID allow it to model memories, and infinite queues or queues of bounded size. UCLID has been previously been used to verify hardware systems [63, 61, 13], software systems [26, 45] and discrete-time hybrid systems [54]. Although UCLID can be used for inductive verification, in this dissertation it is used for bounded model checking. From a given initial state, UCLID performs the necessary unrolling of the transition relation to a specified depth k, and then either outputs an SMT problem in SMT-lib format, or eagerly encodes the problem as a decision procedure and dispatches it to a SAT-solver. If the problem is output in SMT-lib format, then any available SMT solver can be used to check the property.

Chapter 4

Formal NoC Modeling

This dissertation uses finite-state NoC models so that QoS problems can be solved using model checking. To minimize complexity (e.g. the number of states in model), it is therefore desirable to use a model that is as simple as possible while still capturing with sufficient precision the details that determine QoS. The approach for minimizing complexity is to use, to the maximum extent possible, a simple modeling language called xMAS to represent NoC designs. As the xMAS modeling language is not well-suited to certain aspects of control logic, xMAS models are supplemented with arbitrary combinational and sequential logic where needed. The modeling methodology presented in this chapter serves as the basis for all contributions in subsequent chapters of the dissertation.

This chapter is structured as follows. The xMAS modeling language is presented in Sec. 4.1. Two different implementations of xMAS queues are presented in Sec. 4.2. Modeling of NoC router cores is given in Sec. 4.3. Approaches for checking latency bounds as simple safety properties are given in Sec. 4.4, and traffic modeling is in Sec. 4.5. Among the sections of this chapter, Sec. 4.3, Sec. 4.4, and Sec. 4.5 contain novel modeling contributions.

4.1 Executable Microarchitectural Specifications (xMAS)

As mentioned in the chapter introduction, the modeling language used heavily in this dissertation is xMAS – standing for executable microarchitectural specifications. The xMAS formalism was developed by researchers at Intel's Strategic CAD Lab as a way to formally model communication fabrics [22]. The development of xMAS was motivated by the observation that communication fabrics are essentially compositions of queues, arbiters, and routing logic, connected and interfaced to each other with a large amount of ad-hoc glue logic. The authors pose the following question in the first paper on xMAS [22]:

> Do we need this ad hoc glue logic? Instead can we identify a set of primitives that is rich enough to permit a purely structural description of interesting systems?

In response to this question, xMAS is a way to model networks that obviates the need for ad-hoc glue logic by having all primitives adhere to a uniform interface that allows components to be connected directly without glue logic. By providing a clean formal model of NoCs, xMAS makes it possible to verify NoC properties using model checking [21, 23, 89] and other means [48].

Every xMAS model $\mathcal{N}$ is a finite state model composed of modeling elements drawn from a library of eight simple parameterized kernel primitives (Fig. 4.2). All primitives communicate over channels using a common handshaking protocol. This provides a simple way to create network models by simply wiring together primitives with channels. xMAS models use synchronous semantics, and each stateful primitive updates its state variables on the same edge of an implicit global clock that is omitted from diagrams of network models. An xMAS model is easily translated into the UCLID modeling language (Chapters 5 and 7) or into synthesizable Verilog (Chapter 6) by mapping each primitive into the appropriate syntax. This provides a convenient tool flow for verifying properties of xMAS models using UCLID or RTL model checking.

Communication over Channels

Communication in xMAS networks occurs between two primitives over a channel. The primitive that is initiating the communication is referred to as the initiator of the channel, and the primitive that is receiving the communication is referred to as the target. Each channel, drawn as a single directed edge in xMAS network diagrams, comprises three signals that are shared between the initiator and target (Fig. 4.1). For any channel c, the initiator primitive controls Boolean signal $c.irdy$ (initiator ready) and $c.data$, while the target controls $c.trdy$ (target ready). The value of $c.data$ is transferred from initiator to target on any cycle when $c.irdy$ and $c.trdy$ are both asserted.

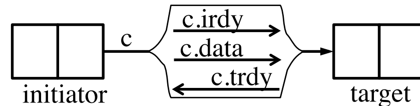


Figure 4.1: **The Signals Comprising an xMAS Channel.** The upstream primitive, denoted the initiator of channel c, controls $c.data$ and the Boolean signal $c.irdy$. The downstream primitive, denoted the target, controls $c.trdy$. The value of $c.data$ is transferred from initiator to target when $c.irdy$ and $c.trdy$ are **true** in the same cycle.

The communication protocol used in xMAS is an instance of latency-insensitive design [20], where functional correctness is independent of channel latencies. The channels in xMAS networks obey forward (Eq. 4.1) and backward (Eq. 4.2) persistency. Persistency means that a channel that is ready to initiate or receive data will remain so until a transfer occurs over the channel. The behavior of each primitive ensures that it will always obey forward and backward persistency, provided that all primitives it interfaces to also obey forward and backward persistency.

$$c.irdy \wedge \neg c.trdy \implies \mathbf{X} c.irdy \tag{4.1}$$

$$c.trdy \wedge \neg c.irdy \implies \mathbf{X} c.trdy \tag{4.2}$$

Data Types in xMAS

Each channel has a designated type, in reference to the data signal on the channel. By convention, each channel in this dissertation uses one of two data types: the first type is flit data, and the second type is token data. In xMAS network diagrams with both types, a bold line is used to represent flit data, and a thinner line is used for tokens.

Flit Data

Flit data are bit-vectors that represent the individual flits of a packet (in Chapters 5 and 7) that are communicated across links in an NoC. In Chapter 6, flit data is replaced by packet data that represents all of the collective flits of a packet as one single unit of transfer. Flits and packets are grouped together here into the single type "flit" because of their similarity. Using data to represent an entire packet is a higher level of abstraction than representing individual flits, but ultimately the choice of whether to model data at the packet level or flit level depends on the network model. In chapters 5 and 7, flit-level modeling is used to capture the granting of output channels to head flits, and releasing of output channels by tail flits. In Chapter 6, it suffices to model traffic at the packet level.

Flit bit-vectors encode multiple logical fields. The values stored in certain positions of the data bit-vector represent logical fields such as destination addresses and timing information (discussed further in Sec. 4.4). Examples of these fields are shown in Tab. 4.1.

<table>
<tr><td></td><td colspan="24">bits of data</td></tr>
<tr><td></td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>head flit:</td><td colspan="14">timestamp / age</td><td colspan="4">x dest</td><td colspan="4">y dest</td><td colspan="2">01</td></tr>
<tr><td>body flit:</td><td colspan="14">timestamp / age</td><td colspan="8">payload</td><td colspan="2">10</td></tr>
<tr><td>tail flit:</td><td colspan="14">timestamp / age</td><td colspan="8">payload</td><td colspan="2">11</td></tr>
</table>

Table 4.1: **Data Fields of a Typical Flit.** Head, body, and tail flits all store a timestamp or age in data[24:10]. The bits data[9:2] represent destination information in head flits and arbitrary payload for body and tail flits. Bits data[1:0] represent the flit type.

Token Data

Tokens channels are channels where the data value is null. The information communicated over such a channel is only an agreement by the initiator and target that a transfer has occurred, and the data value is insignificant. Token channels are used to model the transfer of credits in NoCs.

xMAS Primitives

All xMAS networks are compositions of eight simple kernel primitives (Fig. 4.2). The inputs and outputs drawn for each primitive represent channels, but at a more detailed level the inputs and

outputs to each primitive are the signals within the channels. More precisely, the input signals to each primitive are (1) the *irdy* and *data* signals on each channel of which the primitive is the target; and (2) the *trdy* signals on each channel of which the primitive is the initiator . The output signals of each primitive are all the channel signals going in the opposite direction of the input signals.

Primitives can be either stateful or stateless. The stateful primitives update their state variables on each clock edge based on their current state and inputs, and also set outputs based on current state and inputs. The stateless primitives set output signals based directly upon their input signals. Among the eight kernel primitives, the source, sink, queue, and merge primitives have state, while the join, fork, switch, and function primitives are stateless.

The primitives as defined in the following paragraphs differ from the original xMAS paper [22] in the type of data transformations that are used. The original xMAS definition allows join primitives to receive two flits as inputs and produce a single output with a data value that is a function of both inputs. The convention is adopted here that the data transformations performed by the primitives are always unary; such primitives have previously been referred to as restricted primitives [22]. Since all data transformations are unary, it suffices to use the function primitive for all data transformations. All other primitives do not modify flits when propagating them from input channel to output channel.

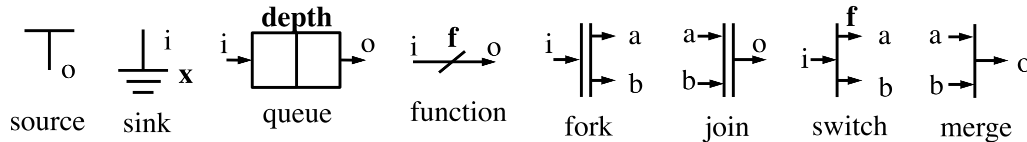


Figure 4.2: **The xMAS Kernel Primitives.** All xMAS networks are created by instantiating combinations of these eight primitives, and connecting them using channels.

Source Primitive

A source primitive is an interface between the environment and network, and generates traffic that it injects into the network through its output channel o. A flit source primitive has a single Boolean state variable $blocked := o.irdy \land \neg o.trdy$ that is **true** when a source is blocked from injecting. This state variable is used to ensure that the source satisfies forward persistency (Eq. 4.1) on its output channel, by continuing to retry a blocked transfer until succeeding. Source primitives are used for various purposes, and a few applications are given here:

- An *Eager Token Source* eagerly attempts to inject tokens in every cycle (i.e. $o.irdy :=$ **true**), and succeeds whenever the target of its output channel is ready to receive.
- An *Periodic Token Source* with a period of x uses a counter that increments in every cycle and rolls over to 0 after $x-1$. It injects a token on channel o whenever the count is $x-1$

- A *Non-deterministic Flit Source* non-deterministically decides when to inject, and the data of said packet is also non-deterministic:

$$\begin{aligned} o.irdy &:= pre(blocked) \wedge oracle \\ o.data &:= oracle \in \mathbb{B}^{|i.data|} \end{aligned} \tag{4.3}$$

- A *Flit Source in a Traffic Model* produces a constrained subset of the behaviors of a non-deterministic source. This is explained in Sec. 4.5.

Sink Primitive

A sink primitive is an interface between the environment and network, and attempts to consume data from a channel i. Finite latencies can only be achieved if sinks are not permitted to block traffic indefinitely, so sinks are not allowed to be fully non-deterministic. The two types of sinks that are used are eager sinks and bounded non-deterministic sinks.

- An *eager sink* is always ready to accept traffic. All token sinks are eager (i.e. $i.trdy :=$ **true**), and flit sinks are eager in Chapters 5 and 7.

- A *bounded non-deterministic sink* uses a non-deterministic oracle to decide when to consume a waiting flit within x cycles, where x is a parameter of the sink. Data sinks are assured of satisfying liveness bounds by their transition relations per the state machine of Fig. 4.3; if the blocking condition $i.irdy \wedge \neg i.trdy$ has been **true** for the past x cycles, then the sink is forced to assert $i.trdy$ in the current cycle.

$$\begin{aligned} waiting &:= i.trdy \wedge \neg i.irdy \\ i.trdy &:= pre(waiting) \wedge oracle \wedge force \end{aligned} \tag{4.4}$$

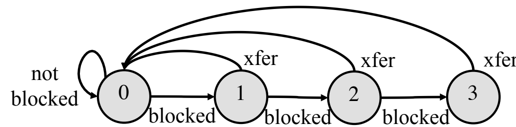


Figure 4.3: **State Machine for Bounded Non-deterministic Sink Primitive.** The state machine shown is for a sink with a bound of $x = 3$. Variable $force$ is **true** when in state 3 and **false** otherwise. Labels $xfer$ and *blocked* are transitions guards, and determine the next state based upon whether or not a transfer occurs for the blocked packet at the sink input. Together with Eq. 4.4, this ensures that data is not blocked for more than 3 cycles before being transferred to the sink and out of the network.

Queue Primitive

A queue primitive is a FIFO buffer that stores flits or tokens. It is parametrized by its depth (or number of slots), which is the maximum number of items that can be stored in it. The most prominent state variables of the queue are for its data storage, but the the queue primitive also has bit-vector state variables to store the current head position, current tail position, and number of items stored in the queue. These variables orchestrate reading from, and writing to, the queue. Data is read from the head position, and written to the tail position.

A queue primitive that holds flits is a model for a typical data-storing NoC buffer such as an ingress or egress queue of a router. A queue primitive that holds tokens represents a credit counter, and is equivalent to a binary counter with xMAS channel interfaces to control the incrementing (i.e. writing) and decrementing (i.e. reading) of the stored count.

Two different implementations of the internal data storage of a queue are given in Sec. 4.2. In either implementation, a queue is ready to receive data (i.e. a write operation can be performed) when the number of items stored in the queue is not equal to the queue depth. A queue is ready to output data (i.e. a read operation can be performed) when the number of items stored in the queue is not 0.

Function Primitive

The function primitive is stateless and maps input data on i to output data on o. The mapping from $i.data$ to $o.data$ is performed according to a deterministic function f that is a parameter of the primitive. The function primitive directly connects the $irdy$ and $trdy$ signals from its input channel and output channel, and is therefore transparent with respect to timing.

$$\begin{aligned} i.trdy &:= o.trdy \\ o.irdy &:= i.irdy \\ o.data &:= f(i.data) \end{aligned} \tag{4.5}$$

Fork Primitive

A fork is a stateless primitive for synchronization. A transfer occurs through the fork when the upstream primitive is ready to initiate a transfer and both downstream primitives are ready to receive a transfer. A typical use for a fork is credit return, where a flit at the fork input is transferred through to output b while a token is generated at output a to represent a credit being returned.

$$
\begin{aligned}
i.trdy &:= a.trdy \wedge b.trdy \\
a.irdy &:= i.irdy \wedge b.trdy \\
a.data &:= null \\
b.irdy &:= i.irdy \wedge b.trdy \\
b.data &:= i.data
\end{aligned} \tag{4.6}
$$

Join Primitive

A join is a stateless primitive for synchronizing data with tokens. A flit channel typically connects to input *b* of the join, and a token channel to input *a*. The flit is then forced to wait for a token before propagating on through the join to the output channel. A transfer through a join occurs when both upstream primitives are ready to initiate and the downstream primitive is ready to receive data.

$$
\begin{aligned}
a.trdy &:= o.trdy \wedge b.irdy \\
b.trdy &:= o.trdy \wedge a.irdy \\
o.irdy &:= a.irdy \wedge b.irdy \\
o.data &:= b.data
\end{aligned} \tag{4.7}
$$

Switch Primitive

The switch is a stateless primitive that performs all of the routing in xMAS networks. The input channel to a switch is always a flit channel. The switch routes the incoming flit to output *a* or output *b* depending on the data value of the input channel ($i.data$). Each switch is parameterized by a switching function $f : \mathbb{B}^N \mapsto \mathbb{B}$, where N is the width of the data bit-vector (e.g. 24 for the flit data shown in Tab. 4.1). Data consumed from input i is therefore routed to output a if $f(i.data) = \textbf{true}$ and to b otherwise. Note that the input data is always produced on both outputs, but only the output channel that is selected by the routing function will have the *irdy* signal asserted to initiate the transfer.

$$
\begin{aligned}
a.irdy &:= i.irdy \wedge f(i.data) \\
b.irdy &:= i.irdy \wedge \neg f(i.data) \\
i.trdy &:= (a.trdy \wedge f(i.data)) \vee (b.trdy \wedge \neg f(i.data)) \\
a.data &:= i.data \\
b.data &:= i.data
\end{aligned} \tag{4.8}
$$

Merge Primitive

The merge is a stateful primitive that performs arbitration in xMAS models. It receives inputs on channels a and b, consumes data from at most one of them, and produces data on output o. A Boolean state variable u represents the current priority among the inputs. A merge can use any priority scheme, such as round robin priority, where the value of u is negated whenever the high priority input transfers a packet through the merge.

$$\begin{aligned} o.irdy &:= a.irdy \vee b.irdy \\ a.trdy &:= o.trdy \wedge \ \ u \\ b.trdy &:= o.trdy \wedge \neg u \end{aligned} \tag{4.9}$$

4.2 Implementation of Queues

The xMAS components that make up the vast majority of network state are the queues. Regardless of whether the xMAS models are implemented in the UCLID verification system [1] or in RTL (for conversion to AIG to verify with ABC), the stateful components other than queues are implemented in a straightforward way as finite state models. However, there is a choice to make in modeling queues when using the UCLID verifier. Queues of a fixed depth can be modeled as circular buffers, but can also be implemented as an ordered data structure using lambda notation. The two implementations are both given in the following paragraphs. Recall from the previous section that in addition to the data storage variables, each queue uses three additional bit-vector state variables to keep track of the head and tail position and the number of items stored. Both queue implementations assume the availability of these variables. An experimental comparison of the two queue implementations in given in Chapter 7.

Queues as Circular Buffers

In a circular buffer queue implementation, the storage element of queue with depth n is implemented using n bit-vector variables `queue0`,`queue1`,..., where each such variable represents the data value stored in one queue slot. State update of each queue slot is performed using case statements. If the tail position matches the index of the slot (i.e. the number x in variable name `queuex`), and a write operation is being performed, then the state variable for the slot is updated with the value `i_data` that is being written to the queue over its input channel. Otherwise, the state variable for the queue slot remains unchanged from its previous value.

The data (`o_data`) that is outputted from the queue on a read operation is also accomplished with a case statement. One case exists for each queue slot, and if a read is being performed and the head points to that slot, then the value from the slot is outputted onto the channel via the data bit-vector for the output channel.

The relevant part of the UCLID model for the circular buffer queue implementation is given below. This queue implementation is also used in the RTL implementation of xMAS networks, as used in the model checking flow of Chapter 6.

```
1   VAR
2
3    queue0 : BITVEC[24];
4    queue1 : BITVEC[24];
5    (* add bit-vector variable for each slot up to depth-1 *)
6
7   DEFINE
8
9    o_data := case
10      head = d0   : queue0;
11      head = d1   : queue1;
12      (* add case for each slot up to depth-1 *)
13      default     : d0;
14   esac;
15
16  ASSIGN
17
18   init[queue0] := d0;
19   next[queue0] := case
20      (tail = d0) & i_irdy & ~i_trdy : i_data; (* write to slot 0 *)
21      default                        : queue0; (* slot unchanged *)
22   esac;
23
24   init[queue1] := d0;
25   next[queue1] := case
26      (tail = d1) & i_irdy & ~i_trdy : i_data; (* write to slot 1 *)
27      default                        : queue1; (* slot unchanged *)
28   esac;
29
30   (* repeated for each queue slot up to depth-1 *)
```

Queues as Records

The second model for FIFO buffers in UCLID follows the memory model presented by Bryant *et al.* [18]. A brief description is given here, and interested readers can find a thorough discussion in [18]. In this queue implementation, the queue storage is modeled by a function expression that maps queue positions to the data stored at those positions. In principle this allows the queue to have arbitrary finite size.

Head and tail pointers are implemented using bit-vector storage variables as in the circular buffer queue implementation. When a write occurs, the tail position is mapped to the data from the input channel (`i_data`) and the tail position is then incremented. When a read occurs, the data value on the output channel (`o_data`) is the value that the head position maps to in the function expression of the queue, and the head pointer is subsequently incremented. The relevant part of UCLID code for the record-based queue implementation is shown below.

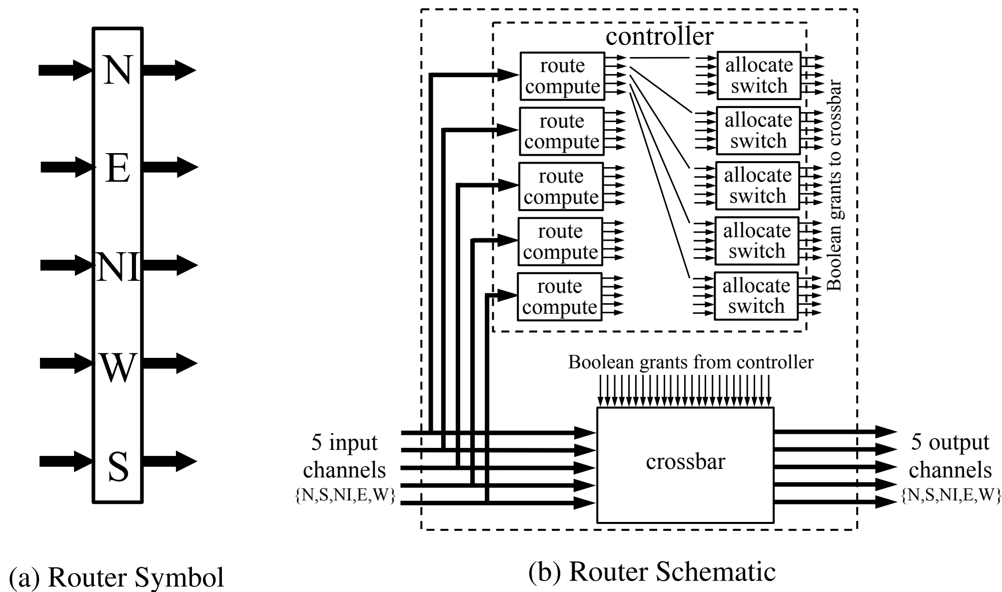
```
VAR

 queue : BITVECFUNC[24,1]; (* Queue/FIFO related signals *)

CONST

 a : BITVEC[24];

DEFINE

 o_data := queue(head); (* Queue function maps head to data *)

ASSIGN

 (* update queue *)
 init[queue] := Lambda(a).d0;
 next[queue] := Lambda(a). case
    a = tail & i_irdy & i_trdy : i_data; (* a now maps to written data *)
    default                    : queue(a);
 esac;
```

4.3 Model of Router Core

Chapters 5 and 7 both use models of NoC routers with five input channels and five output channels, as would be found in a typical two-dimensional mesh network. Those chapters share in common the router core shown in Fig. 4.4. The router core is implemented in UCLID and augments the basic xMAS primitives with arbitrary combinational and sequential logic to implement control logic (comprising route computation and switch allocation) and a crossbar, as shown in Fig. 4.4b. The route computation, switch allocation, and crossbar components are each described in the following paragraphs; along with the description of each component, a fragment of UCLID code shows the most relevant details of its implementation.

Route Compute

Route computation determines which of the five output channels should be requested for incoming head flits. The appropriate request to make depends on three fields extracted from incoming flit data (see Tab. 4.1); these fields are the flit type (f_type), and the x and y components of the destination address (f_addr_y and f_addr_y respectively). For head flits, based upon the extracted destination address fields and the router's own address (r_addr_y and r_addr_x), the request for the appropriate output channel is asserted. These requests are the output of the route compute block and inputs to the switch allocation logic. Note that the route computation implements XY-routing, and data is not routed to the north or south outputs until reaching its destination column.



(a) Router Symbol

(b) Router Schematic

Figure 4.4: **Router Core.** Fig. 4.4a gives the symbol representing a router core. Fig. 4.4b shows that the router core comprises a controller and a crossbar. The controller contains combinational route compute blocks for each input channel, and a sequential switch allocator for each output channel.

```
1   hf       := d1#[1:0];       (* 2-bit constant value for head flits *)
2   f_type   := data#[1:0];     (* flit type, i.e. head/tail *)
3   f_addr_x := data#[9:6];     (* x destination address *)
4   f_addr_y := data#[5:2];     (* y destination address *)
5
6   req_w  := (f_type = hf) & (f_addr_x <u r_addr_x);
7   req_e  := (f_type = hf) & (f_addr_x >u r_addr_x);
8   req_s  := (f_type = hf) & (f_addr_x =  r_addr_x) & (f_addr_y <u r_addr_y);
9   req_n  := (f_type = hf) & (f_addr_x =  r_addr_x) & (f_addr_y >u r_addr_y);
10  req_ni := (f_type = hf) & (f_addr_x =  r_addr_x) & (f_addr_y =  r_addr_y);
```

Switch Allocation

A switch allocator is used for each of the five output channels of the router to control which input channel is granted use of it. The input signals to the switch allocator are the Boolean requests made by each input channel via the route compute. The switch allocator is the only sequential component in the router core. The state variables of each switch allocator are a bit-vector (gnt) to track which input channel is currently granted, and 10 Boolean state variables to store the current priority. Each of these 10 Boolean variables signifies the current priority among a pair of input channels (e.g. pns is the priority ordering between north and south inputs), and the 10 Boolean variables collectively represent a total order of priority among the five input channels of the router.

The switch allocator grants an output, when available, to the requesting input with the highest priority. The state variable gnt is then updated to indicate that the channel is granted, and the priority variables are updated to give the granted input lowest priority going forward. The grant is released when the tail flit is transferred across the input channel.

```
1  init[gnt] := none;
2  next[gnt] := case
3   (gnt = none) & rn & (~rs|pns) & (~re|pne) & (~rw|pnw) & (~rp|pnp) : north;
4   (gnt = north) & ~tail_n : north; (* don't release output *)
5   (gnt = north) &  tail_n : none;  (* release output *)
6   (* repeated cases for other 4 inputs in same style as shown for north *)
7   default : idle;
8  esac;
9
10 (* update priority of n vs s inputs, repeat for other nine pairings *)
11 init[pns] := true;
12 next[pns] := case
13  (state = n_sending) : false;
14  (state = s_sending) : true;
15  default: pns;
16 esac;
```

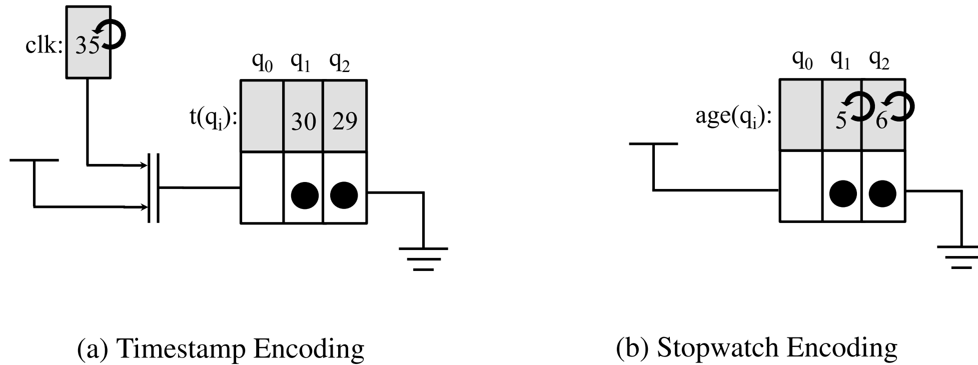
Crossbar

The crossbar is the means through which the non-xMAS router core interfaces to standard xMAS primitives that use channels. The crossbar is a stateless primitive that makes connections between the five input channels and five output channels of the router. Boolean signals from the switch allocator control which input-output channel pairings are connected by the crossbar. When a particular input channel is logically connected to some output channel by the crossbar, the *irdy* and *data* signals of the output are driven from the input, and the *trdy* signal of the input is driven from the output. Note that the crossbar cannot be implemented using the traditional xMAS switch primitive because the switch primitive routes flits based on their value. In the crossbar, body and tail flits do not contain any destination information and are routed according the value of a head flit that preceded them, information that is stored in the state of the switch allocators.

4.4 Checking Latency Bound Properties on Network Model

While any performance property that can be formulated as a safety property is of relevance in this work, the focus is on *latency* bounds. The three major components of this dissertation (Chapters 5, 6, and 7) all concern verifying latency bound properties. The common approach in all these chapters is to compute latencies by tracking the age of flits. Checking the validity of a latency bound of x cycles is then simply a matter of checking the validity of a simple safety property stating that no flit ever has an age greater than or equal to x. Two different simple safety encodings are used, termed "timestamp encoding" and "stopwatch encoding".

All flits within the network occupy a slot of some queue in each cycle. Checking the age of flit data is then a matter of checking properties on queue slots. Let q_i represent the state of the ith queue slot in the network. Assuming the queue implementation of Sec. 4.2, this queue slot is a bit-vector comprising a number of different logical fields as explained in Tab. 4.1, and checking latency makes use of the field that stores the timing information. The notation $age(q_i)$ is used to signify the age of a flit occupying the ith queue slot in the network. A latency bound of x cycles is therefore checked on the ith queue slot using a simple safety property of the form $\phi := used_i \wedge age(q_i) < x$, where $used_i$ is **true** if slot i is non-empty.



(a) Timestamp Encoding (b) Stopwatch Encoding

Figure 4.5: **Two Encodings of Flit Ages.** Both encodings show flits with ages of 5 and 6 cycles. Timestamp encoding stores a static timestamp with each flit and updates a global variable *clk* to advance the age. Stopwatch encoding stores the age directly, and updates each age *in-situ*.

Timestamp Encoding

Timestamp encoding uses an injection timestamp, denoted $t(q_i)$ for the ith slot, and computes $age(q_i)$ as the difference between a global clock (*clk*) and the injection timestamp. To use timestamp encoding, a network model is augmented with a single bit-vector state variable *clk* that implements a global counter from 0 to $t_{max} - 1$, where t_{max} is some number that exceeds the largest age bound that is being checked. In every cycle of execution, *clk* is incremented by 1 (mod t_{max}). The timestamp field of each flit is assigned a static copy of the value of *clk* on the cycle when it is injected into the network, and this timestamp propagates from source to sink unchanged as a part of the flit. Because *clk* is incremented in every cycle to advance the age, the age of the data in slot i is computed as $age(q_i) = clk - t(q_i) \mod (t_{max})$. The overhead for timestamp encoding is the inclusion of the *clk* state variable, and the fact that at least $log_2(t_{max})$ bits of each queue slot's state variable are reserved to store the timestamp as part of the data.

A drawback of timestamp encoding is that ages are not represented canonically. There are t_{max} equivalent representations of the same data ages. More specifically, given any state of the network model, equivalent ages are formed by adding (mod t_{max}) a common offset $x \in [0, t_{max} - 1]$ to *clk* and all $t(q_i)$ timestamps. To show that some particular age cannot be reached by flits, the verifier

must show that all equivalent representations of the age are unreachable. This is a burden for the verifier when using induction as a proof engine, and is exacerbated by larger t_{max}.

Stopwatch Encoding

Stopwatch encoding is denoted as such because it stores within each flit data an age counter that functions like a stopwatch for measuring latency. This variable is $age(q_i)$ for each slot i, and it travels with the flit through the network model. In contrast to the static timestamp in timestamp encoding, the age field of each flit begins at 0 when injected and then is incremented in every cycle until the flit is ejected. Therefore, the value of the stored variable $age(q_i)$ is itself the age of the flit occupying slot i, and there is no need for a global counter. The variable $age(q_i)$ is considered a part of the flit just as is $t(q_i)$ in the timestamp encoding, and it similarly requires each queue slot in the network to reserve at least $log_2(t_{max})$ bits to store it. Because standard xMAS queues store data without modifying it, some extra logic is added to the queue primitives for incrementing the ages in each cycle. For the flit that will be stored in each queue slot i, the age field $age(q_i)$ of the stored data is incremented by 1 (mod t_{max}) relative to the value that would be stored by a normal xMAS queue. This ensures that the ages are incremented both for flits remaining in slot i from the previous cycle, and for flits being newly written into slot i over the queue's input channel.

Compared to timestamp encoding, stopwatch encoding requires a more complex transition relation (incrementing values in all queue slots instead of incrementing a single counter), but represents ages canonically. In stopwatch encoding, an age of x for a flit stored in slot i is unambiguously represented by $age(q_i) = x$.

Chapters 5 and 7 use timestamp encoding; Chapter 6 uses stopwatch encoding, and also presents a comparison of the two encodings and shows stopwatch encoding to perform better when using induction as the proof engine. It is unclear whether the work of Chapters 5 and 7 might also benefit from stopwatch encoding, as this was not evaluated. It is believed that stopwatch encoding would not offer significant benefit in those chapter because they use BMC as the proof engine and would therefore have less benefit from a more canonical representation of bad states.

4.5 Formal Traffic Model

A formal traffic model injects traffic into a network while avoiding unrealistic traffic such ill-formed flits that should never be injected, or flits injected at an overly adversarial rate. The traffic model, denoted $\mathscr{T}$, serves as a formal environment model for an NoC model $\mathscr{N}$. It is constructed as a modeling element that uses constrained non-determinism to generate a large finite set of traffic patterns that meet certain qualifications. Before going into detail about traffic models, notation regarding traffic patterns is introduced.

A concrete traffic pattern, denoted p, is a trace of *what* data is injected and *when*. A traffic pattern can refer to either a single source or collection of sources. For simplicity of notation, the description of a traffic pattern assumes a single source with channel data width of L. Over N cy-

cles, traffic pattern p is a sequence of pairs $\langle(irdy_1, data_1), \ldots, (irdy_N, data_N)\rangle$, where $irdy_i \in \mathbb{B}$ and $data_i \in \mathbb{B}^L$. A source that injects pattern p sends data into the network for all cycles i such that $irdy_i$ is **true**, and the value injected on cycle i is $data_i$. The value of $data_i$ is irrelevant when $irdy_i =$ **false**. A traffic pattern for a collection of M sources is similar to that of a single source, except that each scalar $irdy_i$ value is replaced by a vector of width M, and each $data_i$ of width L is replaced by a vector of M data values that are each L-bit wide.

Using the definition of traffic pattern, a traffic model $\mathscr{T}$ is then a component that non-deterministically generates a pattern p_i from a finite set of possible patterns $P = \{p_0, p_1, \ldots\}$. Note that the set of patterns P for a traffic model $\mathscr{T}$ are an implicit set and never explicitly enumerated. For a traffic model $\mathscr{T}$ that generates L-bit wide traffic to inject on M channels over N cycles, the set of patterns P that can be generated is a subset of $\mathbb{B}^{NM(L+1)}$. Note that the $L+1$ term arises because for each channel, L-bit $data$ values are generated along with a Boolean $irdy$ signal.

The constraints enforced by the traffic model are of two types: data constraints and rate constraints. Corresponding to the two types of constraints, the traffic model $\mathscr{T}$ logically comprises two stages (Fig. 4.6). Stage 1 of $\mathscr{T}$ is tasked with using non-determinism to generate a candidate traffic pattern, and stage 2 imposes deterministic rate constraints on the candidate traffic from stage 1. For every output channel of $\mathscr{T}$, there is therefore a corresponding channel from stage 1 to stage 2 that carries the candidate traffic. In the following sections, a generic instance of such a channel is denoted as channel x, and so its channel signals are denoted $x.irdy$, $x.data$, and $x.trdy$.

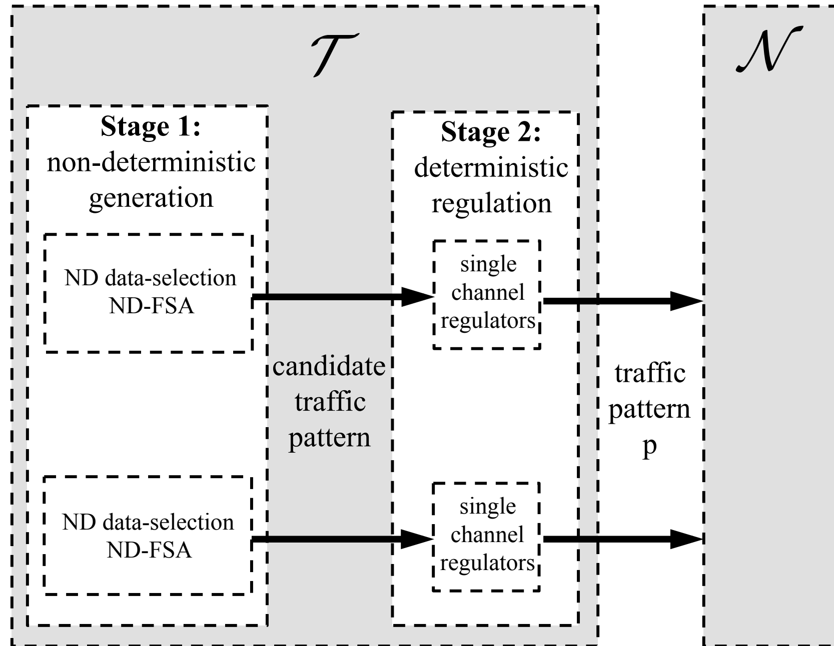


Figure 4.6: **Block Diagram for a Traffic Model $\mathscr{T}$.** Traffic model uses a two-stage process to generate the traffic patterns that are injected into the network model $\mathscr{N}$. Stage 1 generates candidate traffic, and stage 2 regulates the rate of the candidate traffic.

Stage 1 of Traffic Model: Generating Candidate Patterns

A traffic model must not inject flits with fully non-deterministic data across a channel into a network, as the network assumes incoming traffic to satisfy certain conditions. These traffic conditions arise because the different logical fields of the flit data (Tab. 4.1) have defined meanings, and the injected values must be sensible with respect to those meanings. To motivate the need for constrained data, consider the following two examples in which unrealistic flit data could foil attempts at verification:

1. A head flit that is injected without a subsequent tail flit may cause a deadlock because a reservation is made for the head flit but never released by the tail flit.
2. If 3 bits of data are used to encode the five different destination addresses 0 through 4, then injected data with a destination address of 5 may never be routed to any destination.

Bounded non-deterministic data precludes these two types of unrealistic behaviors. Different mechanisms within stage 1 of traffic model $\mathscr{T}$ are used for handling each of the two cases above. While each field such as flit type or destination is generated independently using the appropriate mechanism, ultimately the fields are simply concatenated to generate the overall data that is the output of stage 1 of the traffic model $\mathscr{T}$.

Non-deterministic Finite State Automata

A non-deterministic finite state automaton (ND-FSA) solves the problem of generating appropriately ordered head, body, and tail flits. An example of such an ND-FSA is shown in Fig. 4.7, and implemented in the UCLID modeling language. The state of the ND-FSA determines the single-bit value for $x.irdy$ and the 2-bit value representing the flit-type field of candidate data value $x.data$; the values on the states in Fig. 4.7 represent $x.irdy$ and the flit-type field of $x.data$.

When the ND-FSA is in the idle state, it is non-deterministically chosen whether or not to start sending the flits of a packet by transitioning to the head state. Once in the head state, $x.irdy$ is **true**, meaning that an attempt is made to send to data, and $x.data[1:0]$ is assigned value 01 to indicate a head flit. The ND-FSA remains in the head state until the head flit is transferred through stage 2 of $\mathscr{T}$, and then transitions to the body state to send the body flit, and finally the tail state to send the tail flit. Once the tail flit is sent, a non-deterministic choice is made to send the head flit of the next packet, or else go to the idle state.

One or more ND-FSA form part of the traffic models $\mathscr{T}$ in Chapters 5 and 7. The ND-FSA shown in Fig. 4.7 is exactly the one used in Chapter 5, and the one in Chapter 7 omits the body state, as each packet in that chapter comprises only a head and tail flit. The traffic models of Chapter 6 do not require an ND-FSA, because there each data independently represents an entire packet, and there is hence no need to enforce consistency of flit types.

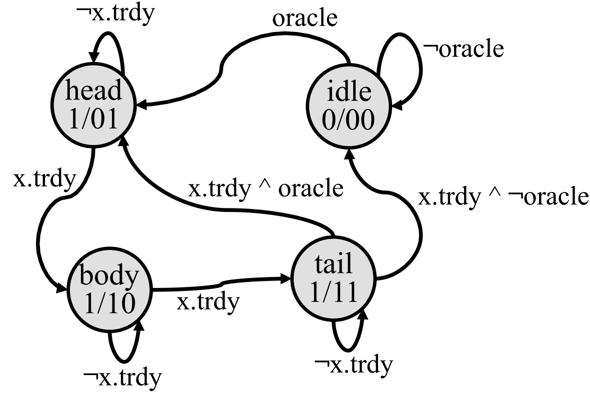


Figure 4.7: **Non-deterministic FSA for Generating Consistent Flit Sequences.** Variations of this ND-FSA are part of the traffic models $\mathscr{T}$ in Chapters 5 and 7. The values on each state represent the values used, when in that state, for *x.irdy* and the bits of *x.data* that encode the flit type. The ND-FSA non-deterministically transitions from the idle state to the state where a head flit is being sent, and only advances from the head, body, or tail states when *x.trdy* is asserted. Because *x.irdy* is always asserted from those three states, *x.trdy* always corresponds to a flit being successfully injected from the traffic model into the network model.

Non-deterministic Data Selection

The injection of flawed data values, as in the motivating example of the incorrect destination addresses, is prevented using non-deterministic choice among enumerated legal values. A multiplexer controlled by a non-deterministic oracle selects between the values. Because every oracle value selects a legal output value, there is no possibility of bad data values.

Stage 2 of Traffic Model: Deterministic Rate Constraints

Rate constraints in $\mathscr{T}$ prevent unrealistic behaviors such as, for example, every source injecting flits on every cycle. The rate constraining mechanisms receive candidate traffic from stage 1 of the traffic model, and produce the traffic model outputs that are the inputs to the network model. The specific rate constraints implemented by traffic models in this dissertation are upper bounds on burst size of traffic, and upper bounds on average injection rate. The constraints are enforced using token bucket regulators built from xMAS primitives, as defined in the following paragraphs. The token bucket regulators are similar to those described by Dally and Towles [38] among others, but coupling the regulators to non-deterministic candidate traffic allows for exploration of "bounded" adversarial traffic patterns in a manner suitable for formal verification.

The structures within dotted boxes in Fig. 4.8 are xMAS implementations of token bucket regulators. Each regulator comprises a join primitive, a token buffer and a periodic token source. The buffer has depth σ and is filled with tokens in the initial state (e.g. in BMC), and a token is consumed each time data is injected through the join primitive from the output of stage 1 of the traffic model. Token source src_ρ adds a token to the buffer every ρ^{th} cycle, unless the buffer is already

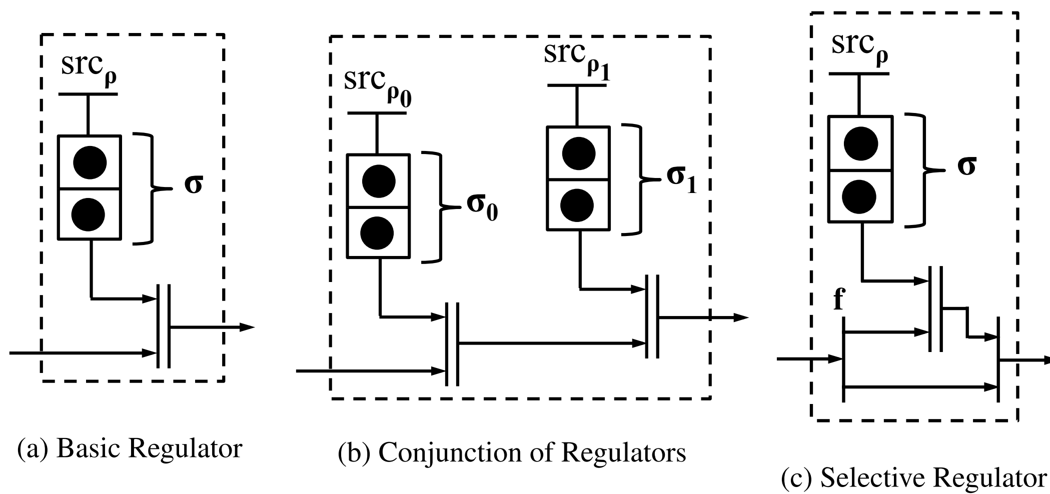
full. When a token buffer is empty, no flits can be propagated through the regulator until more tokens are added.

The token bucket regulators impose the rate constraints on the transition relation of traffic model $\mathscr{T}$, so every output sequence of $\mathscr{T}$ inherently satisfies the regulator constraint. Since at most σ tokens can accumulate in the buffer, and each injected flit consumes one token, σ constrains the burst size of traffic through the join. Also, in each regulator, the maximum number of packets sent through the join in N cycles (after a possible initial burst) is $\lceil \frac{N}{\rho} \rceil$. In this dissertation the relation $\sigma < \rho$ always holds, and therefore the maximum burst size is $\sigma + 1$, as at most one new token is added to the token bucket during the σ cycles when a burst is being sent. The following paragraphs give three different applications of token buckets, corresponding to the three token bucket regulators shown in Fig. 4.8.

Basic Regulator: Fig. 4.8a shows a basic token bucket regulator. When the token buffer is empty, no candidate traffic from stage 1 of the traffic model $\mathscr{T}$ can be sent through the join to reach the output of $\mathscr{T}$. Applying the regulator prevents candidate traffic from being injected at an overly-adversarial rate.

Conjunction of Regulators: Fig. 4.8b shows multiple regulators applied in sequence. The ith regulator constrains burstiness and rate of traffic according to parameters σ_i and ρ_i. The traffic at the output of the overall regulator satisfies all of the burstiness and rate constraints of the individual token buckets. A more detailed discussion of this style of regulator is given in Chapter 5 where it is used.

Selective Regulator: Fig. 4.8c shows a regulator combined with a switch parameterized by a switching formula f. This regulator constrains the rate of all flits with data satisfying f. In Chapters 5 and 7, formula f is set to be **true** for head flits and **false** otherwise; this allows for the rate of head flits to be constrained while allowing body and tail flits to flow unimpeded after their head flits. Otherwise, reservations may be blocked by the regulator for artificially large numbers of cycles while waiting for tail flits to follow their corresponding head flits.



(a) Basic Regulator

(b) Conjunction of Regulators

(c) Selective Regulator

Figure 4.8: **Traffic Regulators.** Within each dotted line is a type of token bucket regulator that receives on its input channel candidate traffic , and produces on its output channel rate-constrained traffic for injection into a network. Each regulator constrains the burstiness of traffic using σ parameters, and the average rate of traffic using ρ parameters. Fig. 4.8a bounds the rate of all flits on a channel. Fig. 4.8b bounds the rate by of all flits on a channel by conjunction of constraints. Fig. 4.8c bounds the rate of all flits on a channel that satisfy formula f.

Part II

Scalable Latency Verification

Compositional reasoning enables scalable model checking of latency properties, and the two chapters in this part of the dissertation present two different approaches for compositional reasoning. The approach in Chapter 5 reduces both the number of variables and the number of unrollings by decomposing both the model and the latency bound. The approach in Chapter 6 keeps the model and property whole, but strengthens the property such that it can be proved inductively, thus reducing the number of unrollings required, and achieving scalability.

Chapter 5

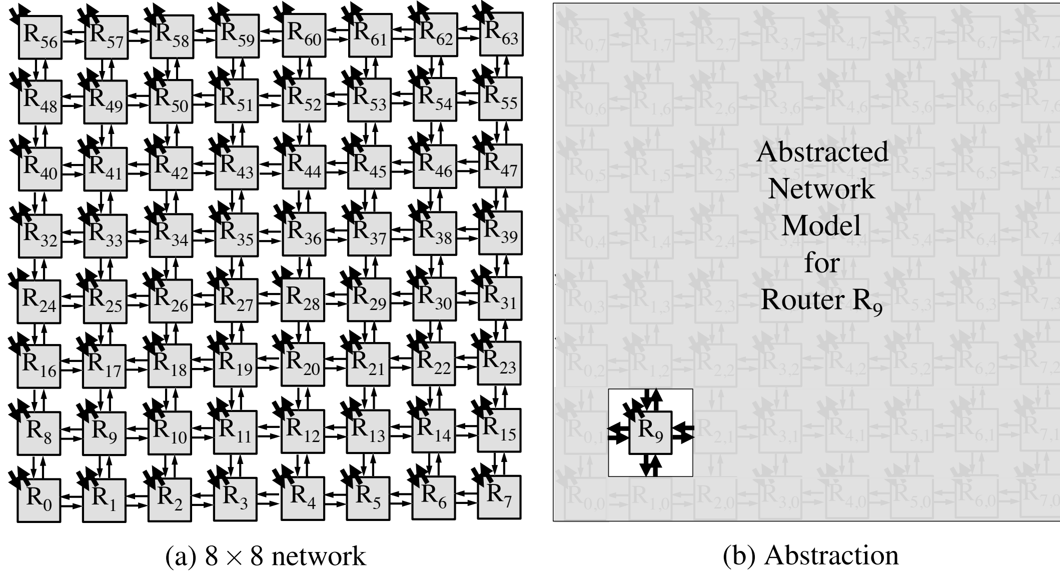
Compositional Reasoning using Traffic Abstraction

This chapter presents an approach to formally analyze latency properties of large NoC designs. Industrial-scale designs are tackled using an abstraction-based approach, where only the routers of interest in the network are modeled precisely and the rest of the network is abstracted away as sources and sinks of traffic. An automatic technique is given for inferring formal traffic models of sources based on simulation traces from software benchmarks. Experimental results demonstrate that the inferred models generalize well and demonstrate that the approach can be applied to industrial-scale models. The significant limitations of this approach as implemented are also given.

5.1 Introduction

Verifying performance (QoS) properties on NoC designs is challenging. This point is illustrated using the 8×8 mesh of interconnected routers shown in Fig. 5.1a. Each router has connections to a processing element (shown by bold arrows), and connections to neighboring routers. Consider the specific router in the mesh labeled R_9, and suppose that one wants to verify the property that every packet traveling through R_9 spends no more than 42 cycles within it. One option is to perform simulations, perhaps derived from software benchmarks. While software-derived testbenches are a good way to characterize workloads for NoCs, simulations can only prove the presence of performance bugs, not their absence. An alternative approach is to use a formal verification method such as model checking [29]. The property above can be formalized in temporal logic, and the overall network can be modeled as a synchronous composition of 64 finite-state machines, one for each router. However, even a simple router has over 1000 state variables, and a model of the entire mesh has tens of thousands of state variables and is beyond the capacity of current formal verification tools.

The obvious approach for applying formal methods to this problem is to use *abstraction*. As depicted in Fig. 5.1b, one can abstract all nodes in the network other than R_9 into an environ-



(a) 8 × 8 network (b) Abstraction

Figure 5.1: **Challenge of Performance Verification.** The goal of this chapter is to verify latency bounds through individual routers, such as R_9. When the entire network is modeled, there are 64 inputs and outputs representing connections between each router and its associated processing element (which is not depicted in figure). When modeling a single router and abstracting away the rest of the network, there are five inputs and outputs representing connections to the router's single processing element and four neighboring routers.

ment model. The traditional approach to generating an environment abstraction is to liberally use *non-determinism*: for example, the environment model can be a state machine that non-deterministically decides at each cycle whether to send a packet to R_9. Such non-deterministic behavior is often restricted using additional constraints. However, while such modeling is useful for verifying "logical correctness" properties such as absence of deadlock, as typically used it does not work well for proving performance properties.

Consider again proving a latency property through router R_9. With a completely non-deterministic environment model that makes no assumptions about typical traffic patterns, it is not possible to prove a useful latency bound. In the worst case, the environment will inject flits on inputs channels of R_9 in all cycles, and furthermore will configure all packets through R_9 to require the same output port. This unrealistic traffic would only allow overly pessimistic bounds to be proved for R_9. The question is: what sorts of constraints on the environment model would be reasonable?

This chapter presents TITAN, a new abstraction-based approach for performance verification of NoC designs. TITAN[1] leverages the presence of testbenches derived from software benchmarks by generating formal models of network traffic from simulation traces. The inferred traffic models are guaranteed to be over-approximations of the simulation traces they are derived from, meaning that they can generate not only the packet sequences in the simulation traces, but also other

[1]TITAN stands for Trace-Inferred Traffic-Abstractions for Network-on-chip verification

sequences that have similar traffic characteristics. The generated traffic models have several applications: (i) they enable the use of formal techniques like model checking to prove QoS properties of NoCs; (ii) they formalize assumptions about traffic patterns under which QoS guarantees hold; and (iii) they can be used for diagnosing the cause of poor performance observed on new software benchmarks. To summarize, the main contributions of this chapter are:

- A technique for inferring traffic models and their associated configuration parameters from simulation traces (Sec. 5.3)
- Experimental results demonstrating formal verification using the inferred traffic models (Sec. 5.4). The generated results provide tighter bounds than traditional non-deterministic models with equal efficiency, but have some limitations (Sec. 5.6). Results are presented for an industrial-scale NoC design [81] using simulation data derived from PARSEC benchmarks [80], with verification by UCLID [18], which uses model checking based on SMT solving [4].

The remainder of this chapter is organized as follows: basic terminology and preliminaries are covered in Sec. 5.2, traffic model inference is presented in Sec. 5.3, experimental results are given in Sec. 5.4, related work is discussed in Sec. 5.5, and conclusion and discussion of limitations are in Sec. 5.6.

5.2 Preliminaries

Basic Definitions

In this chapter, a model of an NoC $\mathscr{N}$ is a tuple $\langle \mathscr{I}, \mathscr{O}, \mathscr{C} \rangle$ where

- $\mathscr{I}$ is a finite set of input *sources*;
- $\mathscr{O}$ is a finite set of output *sinks*;
- $\mathscr{C}$ is a finite set combinational or sequential of NoC components.

Consider the NoC design of the 8×8 mesh in Fig. 5.1a. If the entire mesh is used as a network model $\mathscr{N}$, then $\mathscr{I}$ is the set of 64 sources that inject traffic into the network interface port of each router, and $\mathscr{O}$ is the set of 64 sinks that receive traffic going in the opposite direction; these inputs and outputs are shown as bold arrows in Fig 5.1a.

However, the entire mesh network is too large to formally verify if represented precisely in the model. The size of the model is reduced by considering $\mathscr{N}$ to be a single router instead of the entire mesh of 64 routers. A model of each single router in the network is denoted $\mathscr{N}$ with no differentiation because all routers are identical. If the design to verify is the single router R_9, located at position $(1,1)$ in the grid of Fig. 5.1a, then it can be represented by a model $\mathscr{N}$ with five sources and sinks; one source/sink pair is for the network interface port associated with R_9, and

	Router Output				
Router Input	N	S	E	W	NI
N		✓			✓
S	✓				✓
E	✓	✓		✓	✓
W	✓	✓	✓		✓
NI	✓	✓	✓	✓	

Table 5.1: **Allowed Turns in XY-routing.** For a mesh network of routers that use XY-routing, only the input-output combinations with check marks are possible.

four source/sink pairs are for the channels connecting R_9 with neighboring routers. The source and sink for the network interface are an input and output for the entire network, but the connections to neighboring routers R_1,R_8,R_{10}, and R_{17} are only sources and sinks because the remainder of the mesh network other than R_9 is abstracted away. Analyzing R_9 in isolation therefore requires a model for the behavior of these sources and sinks. This model is the abstract environment model for R_9, as shown in Fig. 5.1b.

Smallest Verifiable Latency Bound

The goal in this work is to verify tight latency bounds of single routers. The timestamp formulation of Sec. 4.4 is used to check latency bounds as simple safety properties on the router model $\mathscr{N}$. The latency property to check a bound of B cycles is written in this chapter as ϕ_B. As each model checking call verifies or disproves a single latency bound, a binary search (Algorithm 1) is implemented around the model checking calls to find the tightest provable latency bound. If a bound B is disproved, then the tightest possible bound can be no smaller than $B+1$; and if a bound B is verified, then there is no need to check any bounds larger than $B-1$. At termination of the binary search there are two checked bounds $B-1$ and B, such that $B-1$ is disproved and B is verified.

Algorithm 1: Find largest valid latency bound for router i under inferred traffic model $\mathscr{T}_i$.

1: $B_{min} \leftarrow 0$ ▷ Smallest bound that is not yet disproved
2: $B_{max} \leftarrow 30$ ▷ Largest bound that is not yet confirmed valid
3: $B_{valid} \leftarrow -1$ ▷ Tightest valid bound
4: **while** $B_{max} \geq B_{min}$ **do**
5: $B' \leftarrow B_{min} + \text{int}(B_{max} - B_{min})$
6: **if** $\mathscr{N} \| \mathscr{T}_i \models \phi_{B'}$ **then**
7: $B_{valid} \leftarrow B'$
8: $B_{max} \leftarrow B' - 1$
9: **else**
10: $B_{min} \leftarrow B' + 1$

5.3 Traffic Model Inference

As noted earlier, NoC designers might wish to optimize their designs for particular *kinds* of software benchmarks from a set of application domains. The PARSEC benchmarks [80] are one such example. However, it is desirable to optimize the design not just for the specific programs in the benchmark suite, but for all programs that generate "similar" traffic. This section describes the form of traffic model used in this chapter, and how such a model is inferred from simulation data. The model inference problem addressed in this section is, for each router, as follows:

> Given a finite set of traffic patterns for the 64 inputs to the mesh network, create for each router R_i a traffic model $\mathcal{T}_i$ that generates traffic for the router's five inputs that is a superset of the traffic induced on the same five channels when simulating the patterns on the mesh network.

The traffic models used in this chapter follow the methodology of Sec. 5.3. For each single router R_i, a customized traffic model $\mathcal{T}_i$ is generated using a three step process.

1. The first step (described in Sec. 5.3) is to define the form of constraints to use in the traffic model $\mathcal{T}_i$ for each router R_i, while leaving the parameters of the traffic model (e.g. for burstiness and rate constraints) unassigned.
2. The second step (described in Sec. 5.3) is to simulate benchmarks on the entire 64-router mesh network to propagate the network-level inputs onto all the channels of the network. Based on the induced traffic at the inputs of each router R_i, constraints are inferred.
3. The third step (described in Sec. 5.3) is to translate the inferred constraints for each router R_i into the parameters of the traffic model $\mathcal{T}_i$ that is used in verification.

Form of Traffic Model

As introduced in Sec. 4.5, traffic models are used for precise modeling of the traffic injected by sources. A traffic model $\mathcal{T}$ precludes injection of ill-formed flits and overly adversarial injection rates. The traffic models applied in this chapter constrain both the rate and the destinations of packets. As latency verification is performed on single routers in isolation (Fig. 5.2), a separate traffic model $\mathcal{T}_i$ is used for each router R_i. Because the router has five inputs (N,S,E,W,NI), its traffic model $\mathcal{T}_i$ generates traffic to inject on all five of the inputs. Including the traffic model, the model checking problem for a property ϕ on a single router R_i with formal model $\mathcal{N}$ is then $\mathcal{N} \| \mathcal{T}_i \models \phi$.

For each router R_i, the traffic model $\mathcal{T}_i$ imposes the following constraints:

- For each of the five input channels, Boolean constraints restrict the destinations of head flits to preclude routing paths that cannot occur in a mesh network with XY-routing.

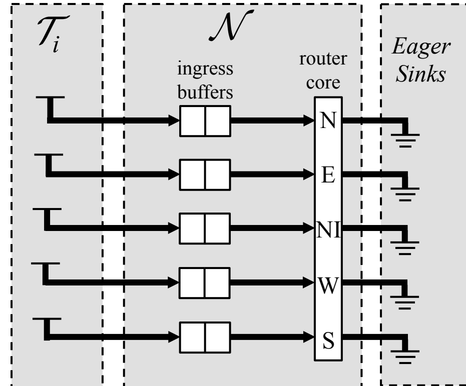


Figure 5.2: **Traffic Model and Router Model to Verify.** Router R_i with formal model $\mathcal{N}$ has neighbors abstracted away into traffic model $\mathcal{T}_i$. If R_i is R_9, this figure is a more detailed view of Fig. 5.1b. The ingress buffers each have a depth of 8 slots.

- For each of the five input channels, rate constraints for all traffic crossing the channel and entering into router R_i.

Boolean constraints restrict the destinations of injected traffic on each of the five input channels. The Boolean constraints in the traffic models of this chapter serve to impose network-level routing onto individual channels. The CMP router in this work uses deterministic XY-routing, meaning that a packet with destination x, y in the mesh is first routed horizontally until reaching column y, and then routed vertically to reach its destination at x, y. From the perspective of a single router, XY routing leads to restrictions on the destinations that packets of a channel can have. For example, packets entering a router on north or south inputs must not exit on east or west outputs, as this would imply that vertical routing occurred before horizontal. The allowed combinations of input and output ports in XY-routing are shown in Tab. 5.1. In the traffic model, the Boolean constraints are implemented using the non-deterministic choice methodology from in Sec. 4.5, so that a source can only inject packets with destinations that are consistent with XY-routing.

The motivation for rate constraints is to avoid unrealistic over-injection of traffic. If simulations never show a case where flits cross a channel in every cycle, then verification allowing this behavior will be unnecessarily pessimistic. To prove tight bounds, traffic rates must be precisely constrained. The form of rate constraints used are burstiness and average rate.

Now that the form of the traffic models is specified, the next step is to infer the values to customize the traffic model for each router.

Inferring Traffic Model Parameters from RTL Simulations

The concrete traces used for traffic model inference are generated from RTL simulation of the entire mesh network. The inputs patterns to the simulation come from the PARSEC benchmarks. The PARSEC benchmarks are too large to simulate exhaustively, so random sampling is used.

During RTL simulation, for each channel in the network, all the traffic to cross the channel is logged to a file. An example of a trace from a log file is shown in Tab. 5.2.

Rate constraints are inferred through finding the *maximum packet density* for different sized time windows in the simulation trace. For a single channel, the following example demonstrates how to maximum packet density constraints are inferred for a single input channel to a router. Assume that for this channel, the RTL simulation trace shows packets crossing the channel on cycles $2, 6, 23, 29, 42, 50$ (Tab. 5.2). For each non-dominated time window duration t, the maximum number of packets to cross the channel is $y(t)$. Therefore, $y(t)$ is the maximum packet density constraints for the trace. Since the constraint will be applied in (at most) a 30 cycle symbolic simulation in this case, any constraints with a time window exceeding 30 cycles can be neglected since they will not disallow any behaviors in a 30 cycle simulation that are not already disallowed by one or more other constraints.

cycle	2	3	4	...	6	7	8	...	23	24	25	...	29	30	31	...	42	43	44	...	50	51	52
p	H_{13}	B	T		H_2	B	T		H_{33}	B	T		H_{37}	B	T		H_{15}	B	T		H_{72}	B	T

Table 5.2: **Example of a Simulation Trace.** Example of a concrete traffic trace across a channel. This trace is an example of what would be induced by some patterns applied at the 64 inputs of the mesh network. The subscript of each head flit shows the packet's destination.

t	$y(t)$	Time of Occurrence in Trace
1	1	2-3, 6-7, 23-24, 29-30, 42-43, 50-51
5	2	2-7
20	3	23-43
28	4	23-51
45	5	6-51
49	6	2-51

Table 5.3: **Maximum Packet Density Constraints.** Packet density constraints inferred from simulation trace in Tab. 5.2. The rate constraints for this channel are represented by the vector of pairs $[(1,1),(5,2),(20,3),(28,4),(45,5),(49,6)]$.

Creating Traffic Model from Constraints

Once the maximum packet density constraints are inferred for each channel of each router, the next step is to translate these into the traffic models that are suitable for verification. The Boolean destination constraints are implemented using non-deterministic choice. The rate constraints are implemented using regulators. The steps given here are for implementing the rate constraints of a single input channel to a single router.

1. The starting point is the inferred rate constraints for the channel from RTL simulation. More precisely, the starting point is a vector of pairs $(t, y(t))$, where t is the shortest time during which $y(t)$ packets cross the channel. Consider the maximum packet density constraints in Tab. 5.3, that were derived from the trace in Tab. 5.2. In Fig. 5.3b, these points are denoted by the symbol '+', and $y(t)$ is considered to be the function that interpolates between these points with straight line segments.

2. Next, a set of (σ, ρ) regulators are inferred. Each is conservative with respect to $y(t)$. As described in Sec. 4.5, each regulator constrains and maximum burst size using parameter σ, and average rate using parameter ρ. Letting B be the largest number of packets observed within a time window size that is equal to the symbolic simulation depth (e.g. 30), the parameters for each regulator $i = 1 \ldots B$ are chosen by first assigning burst size (σ_i) to i, and then choosing the maximum ρ_i such that the regulator (σ_i, ρ_i) yields a function $\hat{y}(t)$ which bounds $y(t)$. The final regulator is always (B, ∞), since a burst size of B is conservative with respect to any 30 cycle symbolic simulation without ever replenishing the token bucket.

3. The rate constraint for the channel is then modeled by the conjunction of all learned token bucket constraints as shown in Fig. 5.3a.

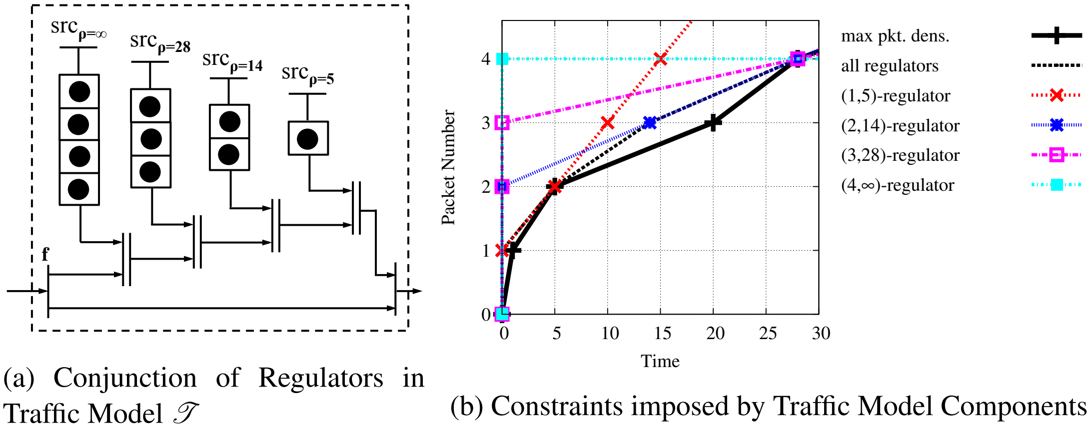
The regulators generated in the above steps are $(1,5)$, $(2,14)$,$(3,28)$, and $(4,\infty)$ for the constraints in Tab. 5.3. The mechanism to implement this channel's traffic model is shown in Fig. 5.3a. The constraint imposed by each token bucket regulator individually, and the constraint imposed by the conjunction of all regulators is shown in Fig. 5.3b. Each regulator in Fig. 5.3b is shown in the figure as the continuous function $\hat{y}(t) = \left(\sigma + \frac{t}{\rho}\right)$; note that the exact constraint imposed by the regulator is actually a step function through the same points, with each step corresponding to a token being added to the regulator's queue. All traffic patterns that fall below the line are modeled by our regulator, and those above it (e.g. sending 3 packets within 5 cycles) are not.

5.4 Experimental Results

TITAN is evaluated on selected routers from the 8×8 mesh, where each router has five input ports and five output ports [81]. Each input port has a single buffer with a depth of 8 slots. The traffic injected into the network during RTL simulation is generated from architectural simulation [96] of the PARSEC benchmark suite [80]. The architectural simulation traces are provided at the packet-level; for RTL simulation, each packet is assumed to be a single head, body, and tail flit injected on the same cycle. RTL simulation is performed using Icarus Verilog.

Convergence of Traffic Model Inference

Each trace of architectural simulation of the PARSEC benchmarks contains hundreds of thousands of packets injected over a span of millions of cycles, making exhaustive RTL simulation a costly



(a) Conjunction of Regulators in Traffic Model $\mathscr{T}$

(b) Constraints imposed by Traffic Model Components

Figure 5.3: **Rate Constraints Imposed by Regulator** The modeling element in (a) implements inferred rate constraints. The switching function f routes all head packets to the upper switch output, and body and tail packets to the lower switch output. This constrains the rate of head flits while allowing body and tail flits to follow without regulation. The constraints imposed by each of the four token buckets in (a) are plotted in (b). All constraints are conservative with respect to all parts of the simulation trace. The dotted black line is the overall constraint enforced by the mechanism in (a).

proposition. Instead of exhaustive simulation, TITAN infers formal models from RTL simulation of a small, random subset of the traces. For each router, 5 constraints are inferred. As explained in Sec. 5.3, the constraints are on the rate of each of the five sources.

The TITAN models are inferred from 500 RTL simulations, where each simulation is a randomly chosen sample of 5000 cycles from a randomly chosen PARSEC benchmark. It is then evaluated whether the traffic models inferred from the first $i-1$ traces remain valid for the ith trace; i.e., whether the ith trace could be generated by the models inferred from the first $i-1$ traces. Fig. 5.4 shows the fraction of all models that are valid for the first $i-1$ simulations and violated by the ith simulation. As the number of simulations performed approaches 500, the vast majority of inferred models remain valid with respect to further RTL simulations. This gives some confidence that the models are largely representative of those that might be inferred from RTL simulation of the exhaustive set of traces.

Latency Verification using Inferred Models

Latency verification experiments show that tighter bounds can be proved by using inferred traffic models instead of fully non-deterministic inputs. To verify a latency property on a router R_i with formal model $\mathscr{N}$, using an inferred traffic model $\mathscr{T}_i$, the overall model is as shown in Fig. 5.2; all five input channels of the router are replaced by a bounded non-deterministic traffic model, and all five output channels are replaced by eager sinks. A latency bound of B cycles is checking using property ϕ_B. The router model $\mathscr{N}$ is symbolically simulated for 20 or 30 cycles (depending on the solver used), and the validity of formula $\mathscr{N} \| \mathscr{T}_i \vDash \phi_B$ is checked for various values of B. A result

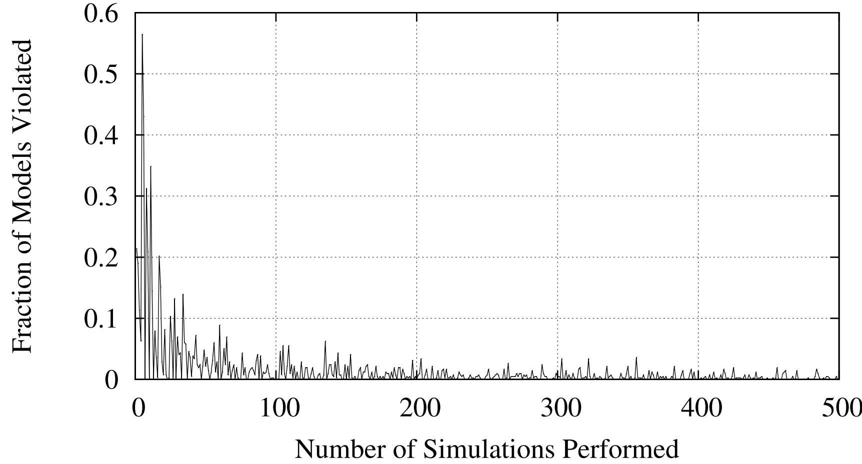


Figure 5.4: **Convergence of Inferred Traffic Models.** Traffic models inferred from hundreds of RTL simulations tend to be valid with respect to further simulations.

of valid means that it is not possible to exceed latency bound B while also conforming to the traffic model $\mathscr{T}_i$. The minimum verifiable bound is found using iterated SMT solving to check different values of B (Algorithm 1). Experiments are performed to find the tightest provable latency bounds for the routers along a diagonal of the 8×8 mesh. Two different solvers are used; the first is UCLID, and the second is Boolector [16].

Using UCLID for Symbolic Simulation and SMT-solving

Table 5.4 shows the tightest proved latency bound in a 20 cycle symbolic simulation for each router along a diagonal of the 8×8 mesh, as well as a single router with fully non-deterministic traffic inputs. The total runtimes for each router are given, and broken down into the runtimes for symbolic simulation, decision procedure encoding, and SAT-solving using MiniSat v2 [43]. The first router in the table is the one using fully non-deterministic inputs, and for this router a latency bound of 15 cycles is disproved with a counterexample, while a bound of 16 cycles is proved[2]. With traffic models, the proved latency bounds for some routers are smaller than the 16 cycle bound obtained with fully non-deterministic inputs. For example, on the routers at the corners of the network (i.e. R_0 at position 0,0 and R_{63} at position 7,7) a latency bound of 13 cycles is proved. The tighter bound exists on the corner routers because two the five input channels are unused and inject no traffic; this is reflected in the rate constraints of traffic models $\mathscr{T}_0$ and $\mathscr{T}_{63}$. For some other routers in the network (i.e. R_{18} and R_{36}), the inferred traffic models are too conservative to successfully verify any latency bounds that are tighter than what is achieved with fully non-deterministic traffic.

[2] In the case of latency verification with fully non-deterministic traffic, the model checking call to check a latency bound of B is $\mathscr{N} \models \phi_B$, instead of $\mathscr{N} \| \mathscr{T}_i \models \phi_B$ as would be used with a traffic model $\mathscr{T}_i$.

i	Pos	*Runtimes in Seconds* Symb. Sim.	Dec. Proc. Encod.	SAT Solv.	Total	Min. Latency Proved
non-det		19	174	1399	1592	16
0	0,0	16	134	806	957	13
9	1,1	23	243	3412	3678	15
18	2,2	27	289	1712	2028	16
27	3,3	22	215	3319	3555	15
36	4,4	29	289	2292	2610	16
45	5,5	23	241	3604	3868	15
54	6,6	22	217	4386	4625	15
63	7,7	18	134	672	824	13

Table 5.4: **Proving Latency Bounds using TITAN Models with UCLID.** Results are for a 20 cycle BMC, for a router with fully non-deterministic traffic, and for 8 routers using inferred TITAN traffic models from an 8×8 mesh (Fig. 5.1). The runtime is broken down into symbolic simulation, UCLID's decision procedure encoding, and SAT solving. The final column gives the tightest latency bound that can be proved. In some cases, TITAN models lead to tighter provable bounds than those achieved with non-deterministic inputs.

Verification using a Strengthened Traffic Model

In 20-cycle BMC, the verified bounds using inferred traffic models are only slightly smaller than the 16 cycle bound obtained using fully non-deterministic traffic (Tab. 5.4). Two changes are made in an attempt to obtain better results: the BMC depth is increased to 30; and additional constraints are added for a more precise traffic model. To facilitate the increase in BMC depth, the SMT solver used is changed from UCLID to Boolector version 1.4.1 [16]. This change circumvents the decision procedure encoding and SAT-solving that were the runtime bottlenecks with UCLID (Tab. 5.4). In this modified experiment methodology, UCLID performs the 30-cycle symbolic simulation, and then outputs the SMT problem in SMT-LIB format for solving with Boolector.

Strengthening Traffic Model with Aggregate Constraint: The traffic model $\mathscr{T}_i$ of each router R_i is augmented with rate constraints on the aggregate sum of all traffic entering R_i across the five input channels. This prevents all five inputs from sending their maximum bursts at the same time, by enforcing that the aggregate burstiness does not exceed what is observed in RTL simulation of the 8×8 mesh. Just as with single-channel regulators in Sec. 5.3, aggregate regulators are parameterized by σ and ρ terms inferred from simulation traces. However, aggregate regulators are implemented differently than the standard token bucket regulators. Instead of a token bucket that imposes the rate constraint on the transition relation of traffic model $\mathscr{T}_i$ (Fig. 4.8), the aggregate regulator constraints are implemented as a logical property $\mathscr{T}_i^{agg}$. Property $\mathscr{T}_i^{agg}$ is checked using counters that saturate at σ, are incremented once every ρ cycles, and decremented in each cycle by

i	Pos	Standard Model $\mathscr{T}_i$		Strengthened Model $\mathscr{T}_i$ and $\mathscr{T}_i^{agg}$		Strengthened vs. Standard	
		Runtime [seconds]	Latency [cycles]	Runtime [seconds]	Latency [cycles]	Runtime [% change]	Latency [% change]
non-det		1621	25	1621	25	-	-
0	0,0	2160	18	3083	15	+42.7%	-16.7%
9	1,1	7626	23	17454	20	+128.9%	-13.0%
18	2,2	5413	24	4444	24	-17.9%	0.0%
27	3,3	12060	24	16014	23	+32.8%	-4.2%
36	4,4	2851	25	4880	24	+71.2%	-4.0%
45	5,5	5848	24	18555	20	+217.3%	-16.7%
54	6,6	6486	24	9927	23	+53.1%	-4.2%
63	7,7	3372	17	4468	14	+32.5%	-17.6%

Table 5.5: **Proving latency bounds using TITAN models and 30-cycle BMC.** Standard model uses only the single channel regulators for each of a routers five input channels, while the strengthened model adds the aggregate rate constraint across all five channels. The row marked non-det is a fully non-deterministic traffic model (i.e. unconstrained), and is repeated in both columns only for comparison against the other routers. The final two columns compare solver runtime and proved bound across the two traffic models. The SMT solver used is Boolector 1.4.1 [16].

the number of head flits[3] sent from $\mathscr{T}_i$ to $\mathscr{N}$ across the five input channels of R_i. Property $\mathscr{T}_i^{agg}$ is **true** if the counter values are always non-negative, as a negative count is analogous to a packet being injected when a token bucket is empty. The model checking problem for a property ϕ on router R_i with aggregate rate constraints is then Eq. 5.1.

$$\mathscr{N} \| \mathscr{T}_i \vDash \mathscr{T}_i^{agg} \implies \phi \tag{5.1}$$

Evaluating Impact of Aggregate Constraint: Table 5.5 shows the tightest verified bounds for each router in a 30-cycle BMC using two different traffic models. The first, denoted “standard model”, is the same traffic model $\mathscr{T}_i$ used in Tab. 5.4, while the second, denoted “strengthened model”, is the standard traffic model augmented by the aggregate traffic constraint $\mathscr{T}_i^{agg}$. The strengthened traffic model increases the runtime and decreases the proved bound for all routers except R_{18} at position (2,2); in R_{18} the strengthened traffic model (i.e. $\mathscr{T}_{18}$ and $\mathscr{T}_{18}^{agg}$) proved the same bound in less runtime.

Relative Difficulty of Checking Different Bounds with SMT

Recall that the tightest verifiable latency bound is discovered by iterated model checking of different bounds within an overall binary search loop (Algorithm 1), and that property ϕ_B checks the

[3]The fact that the count is decremented by the number of head flits in each cycle is the reason why the aggregate regulator is implemented as a logical property and not as a token bucket. If implemented as a token bucket, it would not be possible to drain two tokens from the bucket during cycles when two head flits are injected into the router.

validity of a latency bound of B cycles. Using the router R_{18} at position (2,2) and the standard traffic model $\mathscr{T}_{18}$, an exhaustive check of all bounds is performed by individually checking properties $\phi_1, \phi_2, \ldots, \phi_{29}$, and the results are shown in Fig. 5.5. It is observed that the solver runtime is lower whenever verifying a bound that is much larger than the tightest verifiable bound, or much smaller than the largest disprovable bound. The runtimes grow largest as the checked bound approaches the true bound from either side, as the problem then becomes harder for the SMT solver to decide.

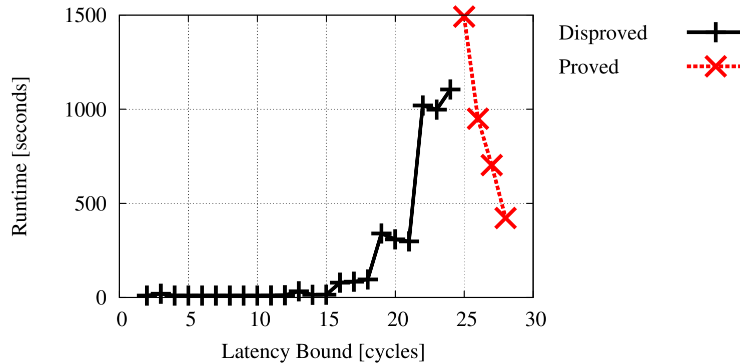


Figure 5.5: **SMT Solver Runtime versus Checked Latency Bound.** SMT solver runtime versus checked latency bound for router R_{18} at position (2,2) in the 8×8 mesh (Fig. 5.1). The SMT problem is more difficult for the solver to decide when the checked bound is close to the true worst-case bound. The SMT solver used is Boolector 1.4.1 [16], and the SMT problem is created from a 30-cycle symbolic simulation using UCLID.

5.5 Related Work

NoC designs can be modeled in various ways for latency verification. Synchronous dataflow (SDF) graphs [64] are suited to modeling networks with regular traffic such as multimedia applications. Network calculus is suited to reasoning about bursty traffic, as in the Internet. Statistical models are suited to average-case analysis. There is much crossover between the approaches.

SDF: While several efforts have used SDF to model NoCs (e.g. [85]), such modeling is limited to only a subset of NoCs. Assumptions of periodic sources and data-independent routing make SDFs well-suited to modeling multimedia NoCs, but not for general-purpose chip multiprocessor (CMP) NoCs. In a CMP, the injected traffic at each node can vary in burst size, have irregular periods, and choose destinations non-uniformly over time. Additionally, due to the lack of support for conditionals, SDFs are not expressive enough to model NoC designs with detailed routing and arbitrary logic.

Network Calculus: The analysis of general-purpose CMPs is typically based on simulation or probabilistic reasoning. Cruz [31, 32] presented network calculus as an efficient way to compute delay bounds in a FIFO network with sources satisfying burstiness constraints. Stochastic automata networks (SAN) [83] have also been used to model network traffic in SoCs [68]. While SANs allow for efficient reasoning about average case results, they are not suitable for worst-case analysis,

as is addressed in this chapter. Addressing limitations in the probabilistic analysis of stochastic models, adversarial queuing theory has been proposed [11]. Networks with arbitrary aggregate multiplexing may not satisfy the global FIFO property that is a precondition for tight bounds in network calculus [91].

Statistical modeling: Various statistical traffic models have been used to characterize network traffic. Varatkar and Marculescu [100] show that NoC traffic for a multimedia application is temporally self-similar, and characterize it using a Hurst parameter. Soteriou *et al.* [96] model traffic flow as a 3-tuple consisting of a Hurst parameter and parameters for hop distance and spatial injection distribution. However, these statistical models are not a formal and conservative representation of the underlying traffic, and it has been reported that designing according to such models can be misleading [55].

Formal Methods: The abstraction of a latency-insensitive system [20] has been explored as a verification aid. A latency-insensitive system assumes only that component delays are multiples of a shared clock, and can be functionally verified independent of the component delays. This abstraction is a useful way for propagating global invariants that can greatly speed up RTL model checking of functional properties of networks [22], and can make certain properties inductive [21]. However, such formal modeling has not previously been demonstrated to be useful for performance (QoS) properties.

5.6 Conclusion and Discussion

The chapter proposes a new formalism to model traffic in an NoC design. An automatic technique is given to infer the traffic model parameters using simulation traces derived from software benchmarks. Experimental results demonstrate that this abstraction-based approach can verify latency bounds for routers in an industrial-scale 8×8 mesh design. Apart from the application to formal verification, it would be interesting to explore other applications of the proposed model such as performance diagnosis.

BMC Depth and Completeness

In this chapter, all verification is performed using BMC. The deepest BMC performed is 30 cycles. The 30 cycle BMC depth appears to be too small to prove results, but there is reason for optimism regarding the results that might be obtained from a deeper BMC.

The Need for a Deeper BMC

Aside from special cases where a completeness threshold is reached [27], BMC is well-known to be an incomplete verification engine. This means that a latency bound that cannot be disproved within 30 cycles of the initial state may be disproved during later cycles. Due to the manner in which latency is verified in this work, it is not possible to disprove any bounds greater than 29 within a 30 cycle BMC; this would hold even if a network was deadlocked and the true latency bound

was infinite. If the proved latency bound is (suspiciously) close to the BMC depth, it is reasonable to guess that the BMC depth is the limiting factor in what latency bounds can be disproved, and that the verified latency bounds are not true bounds. Many of the latency bounds in Tab. 5.4 and Tab. 5.5 are close to the BMC depth and therefore might be disproved with a deeper BMC.

Optimism Regarding a Deeper BMC

While BMC depth is currently limited by tools and the size of the models being verified, the inferred traffic models will become less pessimistic as the BMC depth increases. Recall that the traffic models must be conservative with respect to all of the worst-case behaviors that can occur within the symbolic simulation depth. So for a 30 cycle BMC, the inferred traffic model of a channel must be conservative with respect to the maximum number of packets to cross the channel within any 30 cycle window of RTL simulation. If the BMC depth was increased to 60 cycles, then clearly the traffic model of the channel would need to allow more injected packets, but the number of injected packets in any 60 cycle window is always less than (or equal to) twice the maximum number injected in 30 cycles. In this way, the average injection rate allowed by the traffic model becomes less adversarial as BMC depth is increased.

Limitations of Proposed Approach

The TITAN method presented in this chapter is viewed as a first step toward formal traffic modeling. The key idea is to create a formal model of traffic to serve as an interface specification that is less pessimistic than fully non-deterministic traffic assumptions. While the given procedure for deriving traffic model parameters from simulation is a novel contribution, there are some limitations in how such traffic models could be applied.

Open Question about Applicability of Approach

Fig. 5.4 shows that the inferred traffic models in TITAN generalize well with respect to simulation traces, but there remains a valid concern about whether the inferred models are conservative with respect to future traffic patterns. Although network calculus may suffice in special cases, this dissertation does not give any general approach for determining whether inferred models are representative of the traffic generated by new software benchmarks without first simulating them. As simulating a benchmark reveals a latency bound for the exact pattern in the benchmark, this might be viewed as obviating the need for formal latency verification using generalized models. However, the latency bounds obtained by simulation alone are less general than the bounds verified using the inferred model.

The significant contribution of this chapter is the notion that traffic models can serve as interface specifications for decomposing network models when performing latency verification. This contribution is considered only a first step toward model decomposition. Further applications are left as open questions.

Relaxing the Eager Sink Assumption

As mentioned in Sec. 5.4, when performing analysis on a single router, the environment model that represents the rest of the network acts as an eager sink of traffic. This eager sink assumption is akin to assuming that neighboring routers never present backpressure to the router being verified. This assumption is optimistic and will likely not hold in practice, as significant backpressure occurs during times of high congestion. It is theoretically possible to use a sink model that is conservative with respect to the backpressure observed in simulation, but as the following example illustrates, this requires a very large BMC depth.

Theory: Consider two routers: R_9 and its north neighbor R_{17}. Fig. 5.1a shows that traffic exiting the north output of R_9 enters the ingress buffer of the south input channel of R_{17}. If the south channel ingress buffer of R_{17} is full, then it is unready to receive traffic and asserts backpressure. It is possible to measure during simulation the maximum number of consecutive cycles for which the ingress buffer is full; let this number be x. When router R_9 is then analyzed in isolation, the channel to abstracted router R_{17} can then be received in the abstract environment model by a sink satisfying bounded liveness; this sink can block traffic but must always accept waiting flits within x cycles. This notion of bounded-liveness on sinks is used extensively in Chapter 6.

Practice: The smallest modification to an eager sink would be a sink that accepts waiting traffic after at most 1 cycle. At a minimum, this causes a doubling of the worst-case latency through each router; but given that nearly all latency bounds proved in this work are already greater than 50% of the BMC depth, it is not possible to verify any meaningful bounds without increasing the BMC depth beyond the current capacity of the tools used.

Lessons From this Chapter that Shaped Chapter 6

A variety of significant challenges were uncovered while developing the work presented in this chapter. As explained, some of these challenges were not successfully addressed. The following list gives an overview of how the lessons learned in this chapter's work shaped the subsequent compositional approach presented in Chapter 6:

- Given that in most cases latency bounds verified using traffic models are only marginally smaller than those verified using fully-nondeterministic traffic, Chapter 6 eschews rate-constrained traffic models and instead allows injections with arbitrary rate.
- The bounded liveness model suggested here for traffic sinks is adopted in Chapter 6.
- The compositional reasoning approach in this chapter can be viewed as decomposing both the end-to-end latency bound property and the network model. The bound is decomposed into individual router bounds, and the model is decomposed into individual router models. The decomposition of the model proved challenging because neighboring routers are very much intertwined by the traffic and blocking between them. The approach of Chapter 6 avoids decomposing the model.

- As a corollary of not decomposing the model, the examples considered in Chapter 6 are significantly smaller than the 64-router mesh that motivated this chapter. The largest example there (an 8-agent ring interconnect) has a size, in number of state variables, that is roughly equal to a single five input router from this chapter.

- To achieve greater scalability, the next chapter uses a higher level of abstraction in representing communication, with xMAS data representing entire packets instead of the individual flits of a packet.

- The incompleteness of BMC proved problematic (Sec. 5.6), so Chapter 6 instead focuses on proving latency properties using complete verification methods such as induction and property directed reachability (PDR).

Chapter 6

Compositional Proofs using Induction

This chapter presents a compositional approach to formally verify latency properties of 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. The challenge is addressed in this chapter using a compositional model checking approach in which the overall latency bound problem is strengthened with a number of smaller sub-problems, termed *latency lemmas*. The sub-problems strengthen the overall latency bound property, but are easier to prove on account of being inductive. A method is presented for computing these lemmas based on the topology of the network and a subset of relevant state, and verification using latency lemmas is performed using both k-induction and the IC3/PDR model checking engines. The effectiveness of this compositional technique is demonstrated on illustrative examples and a ring interconnection network motivated by an industrial design. In the ring network, a latency bound that cannot be verified in 10,000 seconds without lemmas is proved inductively in just 75 seconds when the lemmas are used.

6.1 Introduction

High-level modeling of NoCs [22, 21, 24, 23] and automatic abstraction [50] 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 can be encoded as 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 [94], interpolation [71], and IC3/property-directed reachability (PDR) [12, 42] to verify this property. However, regardless of the strategy, it is generally 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, and unrolling of model transition relation to such depth is beyond the capacity of state-of-the-art model checking engines. Property-directed reachability [12, 42], while avoiding explicit unrolling of the transition relation, still does not scale past tens of clock cycles, as will be shown in Sec. 6.6 and Sec. 6.7.

This chapter addresses the scalability challenge using 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 much time does it spend along a particular subpath, etc. These proof sub-goals are termed *latency lemmas*. Methods to discover and apply them are the core contributions of this work in this chapter.

Specifically, this chapter shows 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 stage graph is defined through the use *age lemmas* that bound the total time between when a packet is injected into the network and when it exits each stage. The age lemmas are in turn created through the use of *progress lemmas*, which bound the number of cycles that a packet can spend in each stage. Age lemmas and progress lemmas are collectively referred to as latency lemmas, and by proving the latency lemmas one proves bounds corresponding to all paths through the network.

To summarize, the approach presented in this chapter makes the following novel contributions:

- A compositional approach to proving NoC latency bound properties by decomposition using latency lemmas.
- Methods of formulating latency lemmas for xMAS models using a *stage graph*.
- A method for encoding packet ages that is tailored for inductive latency verification, and experimental results demonstrating its efficacy.
- Experimental results on illustrative examples showing that the use of latency lemmas can reduce by 20-50x the runtime for inductively verifying latency bounds, and causes k-induction to verify latency bounds 4-10x faster than can be achieved with the state-of-the-art IC3/PDR technique using all of the same strengthenings. Furthermore, it is demonstrated that verification runtime can be traded off against tightness of proved latency bounds.
- Experimental results on an industrial-style ring interconnection network showing that latency lemmas give a speedup of greater than 50x, and in all cases allow latency bounds to be proved

inductively with a small induction depth. For an 8-agent ring, latency is verified using k-induction in 75 seconds with latency lemmas, and cannot be verified in 10,000 seconds without them.

The rest of this chapter is organized as follows. Sec. 6.2 introduces basic terminology and sketches the latency lemma approach using a simple example. Sec. 6.3 presents notation. Sec. 6.4 describes the approach in detail, including rules for creating the stage graph. Sec. 6.5 presents methodology used to evaluate the approach, and demonstrates efficient encoding of packet ages. Results for illustrative examples are presented in Sec. 6.6, and for the ring network in Sec. 6.7. Related work is given in Sec. 6.8, and conclusions are in Sec. 6.9.

6.2 Preliminaries

As introduced in Chapter 4, NoC designs are described using a high-level modeling formalism called executable micro-architectural specifications (xMAS models) [22]. Recall that xMAS models are compositions of simple primitives (Fig. 4.2), communicating data over channels (Fig. 4.1). In this chapter, the information that is communicated over a channel c represents either packets, where $c.data$ encodes various properties of the packet such as destination address and timing information (similar to Tab. 4.1), or else tokens where $c.data$ is null.

As described in Section 4.1, a transfer from initiator to target occurs whenever the channel signals $c.irdy$ 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 the temporal logic formula $\mathbf{G}\left(c.irdy \implies \mathbf{F}^{\leq x}\, c.trdy\right)$ holds, where $\mathbf{G}$ is the temporal operator "Globally" and $\mathbf{F}$ is "Eventually". In other words, x is the largest number of consecutive blocked cycles on channel c; a liveness bound of $x = 0$ means that a channel never blocks. Recall that transfer attempts are persistent (Eq. 4.1), meaning that if a transfer is blocked on a channel c, $c.irdy$ remains asserted until the block is resolved and the eventual transfer occurs [48].

An xMAS NoC model $\mathscr{N}$ is a composition of the kernel primitives described in Sec. 4.1, with each primitive implemented as combinational logic or a finite state machine. The network has an initial state in which token queues are full and packet queues are empty. Each packet queue comprises one or more queue slots, and every packet queue slot in the network model $\mathscr{N}$ is indexed by a unique identifier i.

Modeling Conventions in this Chapter

While the methodologies of Chapters 5 and 7 are almost identical but used to solve two different problems, the methodology in this chapter has some significant differences. For clarity, the differences are highlighted here.

- This chapter uses bit level model checking. Instead of describing xMAS network models in the UCLID modeling language, the models are described using Verilog, and then converted

to And Inverter Graphs (AIGs) for verification using the SAT-based model checking tool ABC. For models comprising bit-vector state variables, the translation is straightforward. The only significant difference is that UCLID variables can be assigned non-deterministic values, whereas non-determinism is created in Verilog by assigning non-deterministic state variables from primary inputs that can take any values.

- The data values carried on (non-token) channels represent entire packets. This differs from the convention used in Chapters 5 and 7, where data represents individual flits of a packet.
- Traffic injection rates from sources in this chapter are unconstrained.

Sketch of Latency Lemmas

The approach of this chapter, described in more detail in Sec 6.4, adds two kinds of latency lemmas, termed progress lemmas and age lemmas. An informal sketch is given here to show how each type of lemma is derived and applied, using a credit loop as an illustrative example.

Description of Credit Loop

The credit loop model in Fig. 6.1, adapted from work by Chatterjee *et al.* [21], implements credited flow of packets from a master agent to a target agent. Credited flow control is similar to the use of token bucket regulators, except that new tokens are not generated periodically, but are instead generated upon completion of processing some prior packet. The network has three queues: “available tokens” in the master, and “outstanding credits” and “ingress” in the target. The ingress queue stores packets, and the other two store tokens. The master’s source non-deterministically injects packets on channel a. Injected packets consume one available token when propagating through the join and onward to the ingress. Packets stored in the ingress can be consumed only if there are outstanding credits, and if so the data sink on channel d cannot block progress for more than 5 cycles. Whenever the data sink consumes a packet from the ingress, an outstanding credit is also consumed by the token sink. The token source simultaneously adds tokens to both the available tokens and outstanding credits queue in every cycle when both have free space.

Progress Lemmas – Bounds on Time to Make Progress

The first step toward proving an end-to-end latency bound for the credit loop is to compute a conservative upper bound on how long a packet might wait to advance once it is inside the ingress queue. When computing this bound it can be assumed that the ingress queue holds at least one data packet (i.e. $n_1 \neq 0$) and this is used a starting point to reason about different conditions for the rest of the network state.

(a) If a token is in the outstanding credits queue, then signal $d.irdy$ is asserted and the data sink may at any time consume a packet from the ingress while the token sink consumes an out-

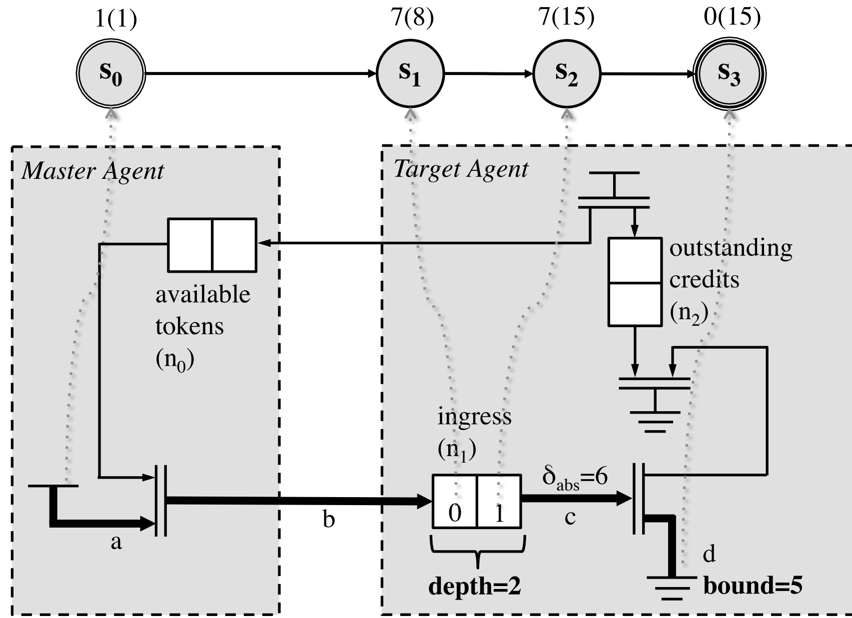


Figure 6.1: **Credit Loop Model $\mathscr{N}$ with Stage Graph $\mathscr{G}$.** The dashed arrows show the automatically-derived correspondence between queue slots in $\mathscr{N}$ and stages in $\mathscr{G}$. Channels in $\mathscr{N}$ that are drawn using bold lines carry packets and the remainder carry tokens. The n_i associated with each queue is the variable representing the number of items stored in it. The first number above each stage in $\mathscr{G}$ is an upper bound on the time spent in the stage, and the second (in parenthesis) is an upper bound on the age of a packet in the stage.

standing credit. The liveness bound of the data sink limits it to 5 cycles of blocking, so $c.trdy$ occurs within 5 cycles from states satisfying $n_2 \neq 0$.

(b) If the outstanding credits queue is empty, and available tokens queue is not full, then the token source will add an outstanding credit, causing $d.irdy$ to be asserted in the next cycle. The sink can only block for 5 cycles once $d.irdy$ is asserted, so in the worst case $c.trdy$ will occur within 6 cycles from states satisfying $n_2 = 0 \wedge n_0 \neq 2$.

(c) If the outstanding credits queue is empty, and the available tokens queue is full, then no token can ever be injected into the outstanding credits queue, and the packet in the ingress will never advance.

It is then hypothesized that every reachable network state with an ingress packet awaiting progress will satisfy either (a) or (b) described above. This hypothesis is formalized as the candidate inductive invariant $\theta_{c,STATE}$ (given by Eq. 6.1 for the credit loop). If $\theta_{c,STATE}$ is valid, and all reasoning is sound, a packet in the ingress should always make progress in, at-worst, the 6th future cycle (denoted by $\delta_{abs} = 6$ in Fig. 6.1), and property $\theta_{c,TIMING}$ (Eq. 6.2) explicitly checks this on channel c.

Properties $\theta_{c,STATE}$ and $\theta_{c,TIMING}$ are collectively called *progress lemmas*. The progress lemmas are conservative and generally over-approximate reachable state. As will be shown later (by Eq. 6.35 in Sec. 6.6), the condition $(n_2 = 0)$ is unsatisfiable when the ingress contains packets, so condition (b) from the above discussion is unachievable.

$$\theta_{c,STATE} := \quad c.irdy \implies \quad (n_2 \neq 0) \vee (n_2 = 0 \wedge n_0 \neq 2) \tag{6.1}$$

$$\theta_{c,TIMING} := \quad c.irdy \implies \quad \mathbf{F}^{\leq 6} c.trdy \tag{6.2}$$

Age Lemmas – Bounds on Time Since Injection

If packets visit queue slots in a known order, and the progress lemmas provide a way to bound the time spent at each slot, then it becomes possible to compute a bound on the total propagation delay through the network. The credit loop has an obvious ordering among queue slots, as a packet injected from the data source first occupies the tail of the ingress (or bypasses it), then the head of the ingress, then reaches the sink. The progress lemmas assert that channel c will transfer any waiting packet in the 6th future cycle in the worst case. This means that a packet will advance every 7 cycles. If a packet cannot spend more than 7 cycles in either ingress slot, an age bound of 8 cycles is implied for the ingress tail slot, and 15 cycles for the ingress head slot; we call these specialized bounds *age lemmas*, and formulate them using a *stage graph* as shown at the top of Fig. 6.1.

It is clear that the total latency is bounded by 15 if the age lemmas can be proved, yet including the latency lemmas makes the 15 cycle bound compositional and easier to verify using k-induction. Proving the 15 cycle latency bound requires an induction depth of 13 frames without latency lemmas, versus just 8 frames with them; the reduced induction depth translates to a 2x speedup in this illustrative example. In general, the induction depth required to prove a latency bound property without the lemmas is proportional to the total latency, while induction depth to prove the same bound using lemmas is proportional to the time for a packet to make progress. The speedup from using latency lemmas is therefore more pronounced when proving large latency bounds, as will be shown in Sec. 6.6 and 6.7.

6.3 Formalism

As sketched in the previous section, a set of conjectured latency lemmas makes it possible to efficiently verify a bound on the end-to-end latency from any packet source in the network to any packet sink. The model being verified is an xMAS model $\mathcal{N}$. Every queue slot in the network that stores packets (as opposed to tokens) is assigned a unique index i, and variable q_i refers to the content of the ith queue slot.

Checking Cumulative Latencies as Age Bounds

As introduced in Sec. 4.4, a latency bound is translated to a simple safety property by checking the *age* of a packet in each cycle. The age of the packet in slot i is denoted $age(q_i)$. Section 6.5 evaluates different ways of encoding $age(q_i)$ using specification variables[1], and shows that the choice of encoding significantly impacts verification runtime.

The mapping from queue slots in $\mathscr{N}$ to stages in $\mathscr{G}$ can depend on the state of $\mathscr{N}$. This allows the same queue slot to represent different stages of progress depending on certain aspects of network state. The possible mapping from the ith queue slot to the jth stage is defined by a formula $p_{i,j}$; the ith slot maps to the jth stage whenever $p_{i,j}$ is **true**.

$$p_{i,j} : Q_i \times W \mapsto \mathbb{B} \tag{6.3}$$

- Q_i is the set of states of queue slot i; q_i denotes a state of Q_i.
- W is the set of states for selected global variables including the number of items in any queue and the status of reservations in networks that use reservations; w denotes a state of W.

A few special cases are worth mentioning. If a slot i never maps to stage j, then $p_{i,j} =$ **false** regardless of q_i and w. If slot i always maps to stage j, then $p_{i,j} =$ **true** regardless of q_i and w.

Each stage j in the stage graph $\mathscr{G}$ has an associated t_j that is a claimed upper bound on the age of any packet that maps to the stage. An age lemma for the ith slot and j^{th} stage of progress is written as $\phi(i, p_{i,j}, t_j)$ (Eq. 6.4). 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. The property $\phi(i, p_{i,j}, t_j)$ checks that, whenever the network state satisfies $p_{i,j}$, any packet stored in slot i must have been injected into the network less than t_j cycles prior.

$$\phi(i, p_{i,j}, t_j) := \quad used_i \wedge p_{i,j} \implies age(q_i) < t_j \tag{6.4}$$

In this chapter, property Φ^L is **true** in a state of $\mathscr{N}$ if and only if the age lemmas hold for all progress stages in the stage graph $\mathscr{G}$.

$$\Phi^L := \bigwedge_{i \in [0, M-1], s_j \in \mathscr{G}} \phi(i, p_{i,j}, t_j) \tag{6.5}$$

Property Φ^G_t denotes a global latency bound of t. The global bound differs from stage bounds in that it is checked on packets stored in all queue slots regardless of the state of $\mathscr{N}$.

[1]Specification variables are defined here as variables in $\mathscr{N}$ that record expressions over system variables but do not influence them.

$$\Phi_t^G := \bigwedge_{i \in [0, M-1]} used_i \implies age(q_i) < t \tag{6.6}$$

Auxiliary Invariants (Ψ)

An advantage of modeling microarchitectures using the xMAS formalism is automated invariant strengthening. The automatically generated invariants are unrelated to QoS, but are essential for verifying any type of property in xMAS networks because they prevent the verifier from exploring unreachable states that may include deadlocked states. The set of auxiliary invariants is denoted Ψ and comprises local invariants on queues [22, 24], persistency invariants on channels [48], and design-specific numeric invariants ψ_{num} [21, 23]. Note that the design-specific numeric invariants are automatically derived in earlier works [21, 23], and added manually in this work.

Proving a Latency Bound

In this chapter, the model checking problem for proving a latency bound t is $\mathscr{N} \models \Phi_t^G$. With auxiliary invariants the problem becomes $\mathscr{N} \models \Phi_t^G \wedge \Psi$. The latency lemma approach, described in detail in the next section, further strengthens the checked property using progress lemmas Θ and age lemmas Φ^L, such that the overall problem becomes $\mathscr{N} \models \Phi_t^G \wedge \Psi \wedge \Phi^L \wedge \Theta$. This strengthened property is shown to be compositional, and it leads to inductive proofs with shorter induction depths and smaller runtimes.

6.4 Latency Lemmas

A distinguishing feature of this work is to strengthen overall latency properties using precise bounds termed age lemmas for different stages of progress arranged in a stage graph $\mathscr{G}$. Computing the amount of time that a packet can spend in each stage further requires computing transfer bounds for different channels in the network. This section presents algorithms for automatically deriving age lemmas for a subset of possible xMAS networks. The power of age lemmas is more general than just the subset of designs that can be handled automatically, as will be demonstrated in the ring interconnect example of Sec. 6.7. The rest of this section first presents an automated approach for creating a stage graph, and then presents an automated approach for computing the transfer bounds that are required for computing the ages of each stage in the stage graph.

Generating Age Lemmas (Φ^L) using Stage Graph $\mathscr{G}$

A stage graph is a tool for constructing age lemmas that lead to compositional proofs of overall latency bounds. In each state of the network, every queue slot that stores a packet maps to a stage in $\mathscr{G}$. The stage that a packet maps to determines the age bound to check on the packet.

Formally, a stage graph is an acyclic digraph $\mathscr{G} = (S,E)$ with vertices $s_0, s_1, \ldots, s_{L-1} \in S$ called stages. Each stage has an associated age lemma that checks a specialized bound on the age of any packets mapping to the stage. A stage s_j has an associated lemma $\phi(i, p_{i,j}, t_j)$ asserting that any packet in slot i of $\mathscr{N}$ that maps to stage s_j in $\mathscr{G}$ must have an age less than t_j. The conjunction of all age lemmas is denoted Φ^L (Eq. 6.5).

Stage graph construction is automated for acyclic networks. Acyclic networks are those without cycles in data paths, where "data path" is defined as any sequence $c_0, c_1, \ldots c_N$ of packet channels with each c_i and c_{i+1} being input and output channels of the same xMAS primitive. One can trivially check whether a network $\mathscr{N}$ is acyclic, for example by using depth-first search from each packet channel. An automated approach to construct stage graphs for acyclic networks is presented as a two step process: first creating stage graph topology, and second adding the age annotations to the stages.

Creating Stage Graph Topology

The queue slots of an acyclic network will always have a partial ordering with respect to when a packet can occupy them. The stage graph topology reflects this ordering. Each queue slot $(i; i \in [0, M-1])$ in network model $\mathscr{N}$ maps to a stage $(s_j; j \in [1, M])$ in stage graph $\mathscr{G}$. The mapping from queue slots to stages is accomplished by setting $p_{i,j}$ to **true** for combinations of i and j where $i = j-1$. A special source stage (s_0) is added for all packet sources, and a sink stage (s_{M+1}) is added for packet sinks. These source and sink stages are nonstandard in that no packets can ever map to them while in the network.

Edges in the stage graph reflect transitions that packets can make in $\mathscr{N}$. All data sources map to a stage s_0, and all data sinks map to stage s_{M+1}. Stages s_x and s_y in the stage graph $\mathscr{G}$ are connected by an edge if the components (source, sink, or queue slot) mapping to s_x and s_y are adjacent slots within a single queue, or if there exists a queue-free data path in $\mathscr{N}$ from the first component to the second.

Assigning Age Bounds to Stages

Once the stages and edges of the stage graph are created, age bounds must be assigned to each stage. The first step toward this is computing for each stage s_j, a value d_j that is the maximum residence time of the stage. Source stage (s_0) is assigned $d_0 = 1$, and sink stage (s_{M+1}) has $d_{M+1} = 0$; all other stages correspond to packet queue slots. For a stage (s_j) corresponding to a queue slot, the maximum residence time (d_j) cannot exceed 1 greater than the maximum blocking time of the channel that is the output of the queue containing this slot. While the maximum blocking time of each channel is not known a priori, the next subsection gives a way to compute an upper bound on it, and this upper bound on blocking time is denoted δ_{abs}. Therefore, d_j is assigned $1 + \delta_{abs}$.

Now that each stage in the network is assigned a residence time (d_j), age bounds for each stage are computed. The maximum age in any stage depends on the maximum residence time of that stage, and the maximum age of a packet when it enters the stage. For each stage s_j, this maximum age

then depends on the path from s_0 to s_j in $\mathscr{G}$ for which the sum of d_j is largest. This path sum is called t_j and it is the age bound of stage s_j.

Global Age Bound T_L from Age Lemmas

If every data packet in every reachable state of $\mathscr{N}$ maps to a stage in $\mathscr{G}$, then the largest t_j associated with any stage is a claimed global age bound for $\mathscr{N}$. The largest t_j among all stages is denoted T_L, and therefore $\Phi^L \implies \Phi^G_{T_L}$ provided that coverage of $\mathscr{G}$ is complete. T_L is therefore a global age bound, and property $\Phi^G_{T_L}$ is the global age bound property that is being strengthened with the latency lemmas. T_L is often conservative for several reasons:

- The channel blocking bounds that are computed are conservative, causing the residence times of each stage to be over-approximated. This occurs in the credit loop (Sec. 6.2) where the blocking time of packets in the ingress queue is overestimated.
- The stage graph conservatively over-approximates the connectivity of the network by ignoring logical propagation conditions.
- It may be impossible for any one packet to experience all of the the worst-case progress bounds, even if each is individually achievable.

Cyclic vs. Acyclic Networks

No automated procedure is given to construct a stage graph for cyclic networks. In such a network, a straightforward mapping from each slot to a single stage will induce cycles in $\mathscr{G}$, and this leads to infinite age bounds even if each stage has a known finite residence time. In such a case, the progress stages must be made more precise by refining the $p_{i,j}$ formulas to consider more than just the slot that a packet occupies. Using a ring interconnect as an example, Sec. 6.7 will demonstrate that a manual refinement can be used to obtain an acyclic stage graph from a cyclic network.

Deriving Channel Blocking Bounds and Progress Lemmas (Θ)

The preceding section shows that blocking bounds on output channels of packet queues determine the residence times of stages in the stage graph $\mathscr{G}$. A heuristic is now given to derive the blocking bounds by propagating timing guarantees across xMAS primitives. As the heuristic generates the blocking bounds, it also generates "progress lemmas" that formalize the assumptions made in deriving the bounds. These progress lemmas are added to the overall verification problem to check the assumptions.

Given a channel c, a blocking bound of x cycles is the claim of $c.irdy \implies \mathbf{F}^{\leq x} c.trdy$. The blocking bound of a channel c is derived by first computing for the channel a guarded bound set $r_{c.trdy}$ (Eq. 6.7). Each guarded bound $\langle g_i, \delta_i \rangle \in r_{c.trdy}$ is g_i a predicate on network state, and δ_i a bound on the number of cycles until $c.trdy$ becomes **true** after any network state satisfying g_i. In other words, the guarded bound set $r_{c.trdy}$ can be used to make the claim of Eq. 6.8.

$$r_{c.trdy} \equiv \{\langle g_0, \delta_0 \rangle, \langle g_1, \delta_1 \rangle, \ldots, \langle g_N, \delta_N \rangle\} \tag{6.7}$$

$$\begin{aligned} g_0 &\implies \mathbf{F}^{\leq \delta_0} c.trdy \\ g_1 &\implies \mathbf{F}^{\leq \delta_1} c.trdy \\ &\vdots \\ g_N &\implies \mathbf{F}^{\leq \delta_N} c.trdy \end{aligned} \tag{6.8}$$

From the guarded bound set, a generalized guarded bound $\langle g_{abs}, \delta_{abs} \rangle$ is created, where $g_{abs} := g_0 \vee g_1 \vee \cdots \vee g_N$ and $\delta_{abs} := \max(\delta_0, \delta_1, \ldots, \delta_N)$. The intuition behind the generalized guarded bound is that, whenever some guard holds, then some delay bound must also hold (Eq. 6.9).

$$g_{abs} \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy \tag{6.9}$$

$$\overbrace{c.irdy \implies g_{abs}}^{\text{guard coverage holds}} \quad \wedge \quad \overbrace{c.irdy \wedge g_{abs} \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy}^{\text{guard bound holds}} \tag{6.10}$$

$$\theta_{c,STATE} := \quad c.irdy \implies g_{abs} \tag{6.11}$$

$$\theta_{c,TIMING} := \quad c.irdy \implies \mathbf{F}^{\leq \delta_{abs}} c.trdy \tag{6.12}$$

$$\theta_c := \quad \theta_{c,STATE} \wedge \theta_{c,TIMING} \tag{6.13}$$

If the generalized guard condition holds in all states where a queue attempts to send a packet across the channel, then the generalized bound is an unconditional channel blocking bound that can be used for computing stage residence times. Eq. 6.10 is invariant if guard coverage is complete and the guard bounds are correct. Property $\theta_{c,STATE}$ (Eq. 6.11) checks that guard coverage holds when the initiator of channel c is attempting to send a packet. Assuming that $\theta_{c,STATE}$ is valid, then Eq. 6.10 simplifies to property $\theta_{c,TIMING}$ (Eq. 6.12). Property $\theta_{c,TIMING}$ checks that the bound (δ_{abs}) implied by the guards does in fact hold in the network. For each channel c, property θ_c (Eq. 6.13) is ultimately checked.

Given that the stage graph approach ultimately only makes use of δ_{abs} for computing stage residence times, the motivation for proving $\theta_{c,STATE}$ and $\theta_{c,TIMING}$ is only to formalize and validate the assumptions made in deriving δ_{abs}. If any modifications are needed in the approach for generating the guarded bound sets, counterexamples to $\theta_{c,STATE}$ and $\theta_{c,TIMING}$ will indicate where the modifications are required. If property $\theta_{c,STATE}$ fails, then a counterexample reaches a state where $c.irdy$ is **true** and g_{abs} is **false**; this can be remediated by adding to the guarded bound set a new

guarded bound $\langle g_i, \delta_i \rangle$ such that g_i is **true** for the bad state of the counterexample. If property $\theta_{c,STATE}$ passes and $\theta_{c,TIMING}$ fails, then it means that there exists some $\langle g_i, \delta_i \rangle$ in the guarded bound set such that the guard (g_i) does not imply the bound (δ_i); this would indicate unsoundness in the propagation rules for creating the guarded bound.

The preceding paragraphs demonstrate how to derive a conservative bound δ_{abs} for a single channel c. The residence times for each stage in the stage graph $\mathscr{G}$ are created by deriving such a bound for each channel c that is the output of a packet queue. The property checked on the network is then Θ (Eq. 6.14). The following sections present an approach for computing a guarded bound set for each such channel. This is done using operations for combining guarded bound sets, applied according to the xMAS network connections. A primitive has operations to determine the guarded bound set for each of its output signals using recurrence relations guarded bound sets of its input signals.

$$\Theta := \bigwedge_{\forall c \in \text{ pkt queue outputs}} \theta_{c,STATE} \wedge \theta_{c,TIMING} \tag{6.14}$$

Recurrence Relations for Future Readiness

Guarded bound sets represent future readiness of channel initiators and targets in a way that is analogous to how *irdy* and *trdy* signals represent their current readiness. For any signal *irdy* or *trdy*, $R(irdy)$ or $R(trdy)$ denotes a symbolic representation of the guarded bound set describing its future readiness. Just as the readiness signals *irdy* and *trdy* are defined using recurrence relations over other readiness signals and state variables, the guarded bound sets for future readiness are defined using recurrence relations over other guarded bound sets and state variables. The recurrence relations for $R(irdy)$ and $R(trdy)$ use the three operations MAX,PLUS, and ITE. MAX is used when readiness depends on the latest-arriving signal among two signals described by guarded bound sets. PLUS is used when readiness depends on the two events in sequence. ITE is used when readiness depends on one of two guarded bound sets, and the choice is determined by some Boolean state condition.

Guarded bound sets in symbolic form (e.g. $R(trdy)$) are ultimately expanded into their concrete form (e.g. $r_{trdy} \equiv \{\langle g_0, \delta_0 \rangle, \langle g_1, \delta_1 \rangle, \ldots, \langle g_N, \delta_N \rangle\}$). The first step toward concretizing the symbolic guarded bound sets is defining recurrence relations between inputs and outputs for each xMAS kernel primitive. The relations for each primitive are shown in graphical form in Fig. 6.2 and explained in the subsequent text; note that these specific relations are just one solution among many possible, and alternative relations can be defined that are more precise or more general.

Packet Source: Because a packet source *non-deterministically* injects data packets onto channel o, there is never an upper bound on when a packet transfer will be initiated on the channel. The guarded bound is therefore the empty set $\{\}$; there is no state condition of the network for which it is claimed that *o.irdy* will become **true** within a known time bound in the future. In other words, there is no assurance that the source will ever again attempt to inject a packet.

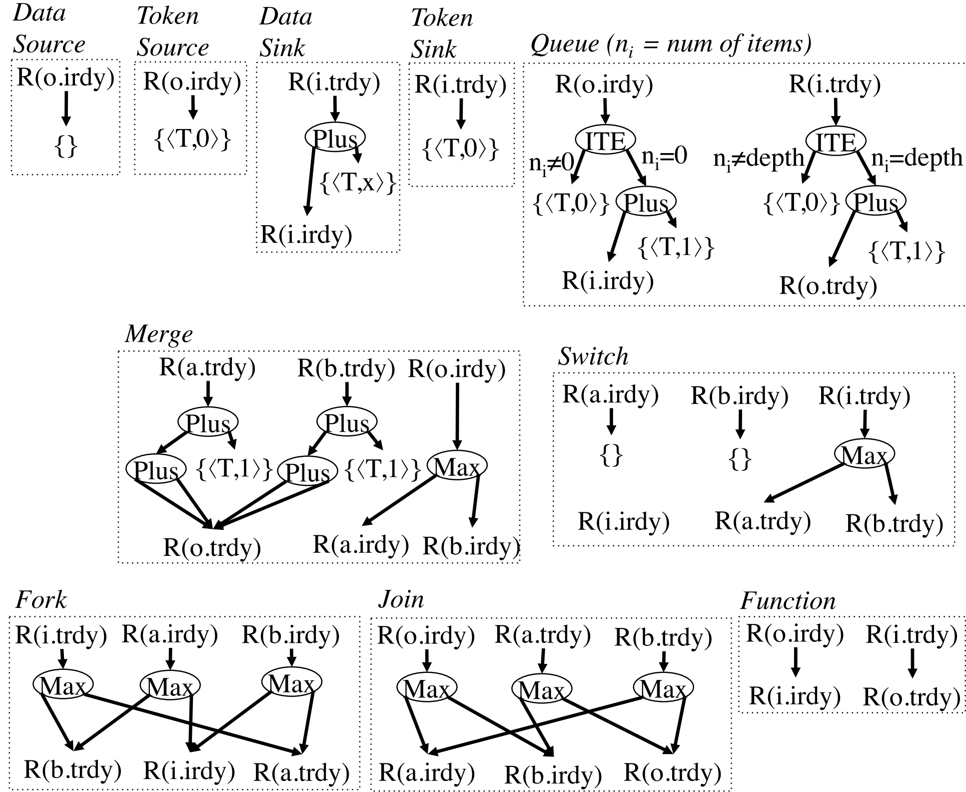


Figure 6.2: **Recurrence Relations for Future Readiness of xMAS Primitives.** Each box corresponds to a kernel primitive, with outputs at top and inputs at bottom. Guarded bound sets for outputs are defined over inputs and constants using operations MAX, PLUS, and ITE.

$$R(o.irdy) \quad := \quad \{\} \tag{6.15}$$

Token Source: A token source is the initiator of a single token channel o, and from any network state it tries to transfer a token across o in every cycle. In the guarded bound $\langle \top, 0 \rangle$, $\top$ is a guard that is always true, and 0 indicates that $o.irdy$ occurs in the current cycle.

$$R(o.irdy) \quad := \quad \{\langle \top, 0 \rangle\} \tag{6.16}$$

Packet Sink: A packet sink is the target of a single packet channel i. The sink provides to the network a service guarantee to always receive a waiting packet within x cycles. This means that $i.trdy$ is guaranteed to be **true** no more than x cycles after $i.irdy$ is **true**. Given that $R(i.irdy)$ is a (recurrence-defined) guarantee on when $i.irdy$ will occur, $R(i.trdy)$ is obtained by unconditionally adding x to it.

$$R(i.trdy) \quad := \quad \textsc{Plus}(R(i.irdy), \{\langle \top, x \rangle\}) \tag{6.17}$$

Token Sink: A token sink is the target of a token channel i. Token sinks are eager and always ready to accept a token regardless of network state.

$$R(i.trdy) \quad := \quad \{\langle \top, 0 \rangle\} \tag{6.18}$$

Queue: A queue is the target of channel i and the initiator of channel o. The queue's recurrence relations for future readiness use ITE to introduce a case-split depending on the number of items stored in the queue (denoted n_i). Consider for demonstration $R(o.irdy)$, which evaluates to $\{\langle \top, 0 \rangle\}$ if $n_i \neq 0$, and otherwise evaluates to $\textsc{Plus}(R(i.irdy), \{\langle \top, 1 \rangle\})$. In the first case the queue is non-empty, and the value $\{\langle \top, 0 \rangle\}$ reflects that the output is currently trying to initiate a transfer. In the second case the queue is empty, and the value $\textsc{Plus}(R(i.irdy), \{\langle \top, 1 \rangle\})$ reflects that the output is ready to initiate no more than 1 cycle after the input is ready to initiate a transfer. This relation holds because any attempted transfer from the input is immediately received into the empty queue to cause an attempted output transfer in the next cycle. Target readiness $R(i.trdy)$ is handled similarly, with a case split depending on whether or not the queue is currently full.

$$R(o.irdy) := \textsc{Ite}(n_i \neq 0, \quad \{\langle \top, 0 \rangle\}, \textsc{Plus}(R(i.irdy), \{\langle \top, 1 \rangle\})) \tag{6.19}$$

$$R(i.trdy) := \textsc{Ite}(n_i \neq depth, \{\langle \top, 0 \rangle\}, \textsc{Plus}(R(o.trdy), \{\langle \top, 1 \rangle\})) \tag{6.20}$$

Merge: A merge primitive arbitrates between two input channels a and b, and is the initiator of a single output channel o. The merge primitive uses a Boolean state variable u to store the current priority among the two input channels. To create a round robin arbitration policy, the state of u is inverted whenever the high priority input transfers a packet through the merge. The recurrence relations abstract away the arbitration priority u to give a conservative bound for each input that does not depend on the current value of u. The high priority input channel can only be blocked for $R(o.trdy)$ cycles before transferring a packet and causing a priority inversion; because of this, the low priority input always attains high priority in at most $\textsc{Plus}(R(o.trdy), \{\langle \top, 1 \rangle\})$ cycles. Therefore, the overall progress bound for an input channel of undetermined priority is the time to attain high priority added to the progress time once high priority is attained.

$$R(o.irdy) \quad := \quad \textsc{Max}(R(a.irdy), R(b.irdy)) \tag{6.21}$$

$$R(a.trdy) \quad := \quad \textsc{Plus}(\textsc{Plus}(R(o.trdy), \{\langle \top, 1 \rangle\}), R(o.trdy)) \tag{6.22}$$

$$R(b.trdy) \quad := \quad \textsc{Plus}(\textsc{Plus}(R(o.trdy), \{\langle \top, 1 \rangle\}), R(o.trdy)) \tag{6.23}$$

Switch: A switch primitive is parameterized by a switching function f. Because no assumptions are made on the data value of a packet arriving on the input channel i, it cannot be determined whether the function f would route the packet to output channel a or output b. Therefore no upper bound is asserted on the future readiness of *irdy* for either output channel. This can optionally be refined using ITE to take into consideration the data value of packets on input channel i.

$$R(a.irdy) := \{\} \tag{6.24}$$
$$R(b.irdy) := \{\} \tag{6.25}$$
$$R(i.trdy) := \textsc{Max}(R(a.trdy), R(b.trdy)) \tag{6.26}$$

Join: A join primitive consumes a packet from input channel a and a token from channel b to produce a single output packet on channel o. The join is only ready to produce a packet on o if the upstream primitives of both input channels are ready to initiate. It is only ready to consume an input packet from input a or b when the other input is ready to initiate and the output channel is ready to receive. The future readiness of each of the three signals depends on the latest future readiness of two other signals, and therefore the MAX operator is used on the guarded bound sets.

$$R(o.irdy) := \textsc{Max}(R(a.irdy), R(b.irdy)) \tag{6.27}$$
$$R(a.trdy) := \textsc{Max}(R(o.trdy), R(b.irdy)) \tag{6.28}$$
$$R(b.trdy) := \textsc{Max}(R(o.trdy), R(a.irdy)) \tag{6.29}$$

Fork: A fork primitive consumes a single input packet from channel i and produces one packet on output channel a and a token on b. As in the join, the fork uses the MAX operator on the guarded bound sets because each output signal is triggered by the latest to occur among two input signals.

$$R(i.trdy) := \textsc{Max}(R(a.trdy), R(b.trdy)) \tag{6.30}$$
$$R(a.irdy) := \textsc{Max}(R(i.irdy), R(b.trdy)) \tag{6.31}$$
$$R(b.irdy) := \textsc{Max}(R(i.irdy), R(a.trdy)) \tag{6.32}$$

Function: A function primitive transforms data, but passes *irdy* directly from input to output, and passes *trdy* directly from output to input. Whenever the initiator of input channel i is ready, then the function primitive is itself ready to initiate. Whenever the target of output channel o is ready to receive, then the function is itself ready to receive.

$$R(i.trdy) := R(o.trdy) \tag{6.33}$$
$$R(o.irdy) := R(i.irdy) \tag{6.34}$$

Concretizing Guarded Bound Sets

The guarded bound sets that describe bounds on future readiness (e.g. $r_{c.trdy}$) are created by expanding the symbolic representations of the same (e.g. $R(c.trdy)$) using the dependency graph of the recurrence relations. The dependency graph is created by composing the recurrence relations (Fig. 6.2) for each primitive in the network according to common signals. Each *irdy* or *trdy* signal in $\mathcal{N}$ is an input of one primitive and an output of another; analogously, each $R(irdy)$ or $R(trdy)$ depends upon the recurrence relations of one primitive, and is also depended upon by those of another primitive. Note that circular dependencies can exist.

For each packet queue in the network, with output channel denoted c, CREATEGUARDEDBOUNDSET (Algorithm 2) is called to extract the concretized guarded bound set $r_{c.trdy}$ from the dependency graph. Starting from node $R(c.trdy)$ in the dependency graph, function EXPANDSUBTREE recursively computes guarded bound sets for each supporting node's future readiness, and includes a check to detect cyclic dependencies and break them by returning an empty set. The guarded bound sets for each node are computed in the tail of the recursion, and thus computed over two already-concretized guarded bound sets. The steps of the algorithm for combining the guarded bound sets of a node's left and right children are chosen according to whether the node is implementing a MAX (line 18), PLUS (line 21), or ITE (line 24) operation. For MAX and PLUS, the guarded bound sets are combined as Cartesian products augmented by the appropriate numeric operation. The ITE operation does not use a Cartesian product because the guards of the two children are made disjoint by including the ITE predicate in opposing polarities.

CREATEGUARDEDBOUNDSET returns to line 2 with a guarded bound set ($r_{c.trdy}$) for target readiness of channel c. Because channel c has a queue as its initiator, and only cases where channel c is blocked are relevant, all guards are strengthened with the condition that the initiating queue is non-empty (lines 3-5). A pruning step is then applied (line 6) to remove from the set any guarded bounds ($\langle g_i, \delta_i \rangle$) that are trivially unsatisfiable. One example of this a guard that asserts that a single queue is both full and empty[2]. Finally, CREATEGUARDEDBOUNDSET returns the final concretized guarded bound set $r_{c.trdy}$, which is then used as explained at the start of this section (Eq. 6.7) to derive residence times of stages in $\mathcal{G}$.

[2]Some guards that are not pruned away on account of trivial unsatisfiability may still in fact be unsatisfiable in all reachable network states.

Algorithm 2: For channel c that is initiated by a packet queue storing number of items n_x, expand recurrence relations to obtain a concretized guarded bound set $r_{c.trdy}$. The members of set $r_{c.trdy}$ are guarded bounds $\langle g_i, \delta_i \rangle$, where guard g_i is a condition on network state, and δ_i is a claimed bound on when $c.trdy$ will be **true** starting from any state satisfying g_i.

```
 1: procedure CreateGuardedBoundSet(channel c)
 2:     r_c.trdy ← ExpandSubtree(R(c.trdy), {})
 3:     nonEmpty := (n_x ≠ 0)          ▷ a queue initiating transfer on c can be assumed non-empty
 4:     for ⟨g_i, δ_i⟩ ∈ r_c.trdy do
 5:         ⟨g_i, δ_i⟩ ← ⟨g_i ∧ nonEmpty, δ_i⟩
 6:     r_c.trdy ← Prune(r_c.trdy)     ▷ prune away guarded bounds where guard g_i is trivially false
 7:     return r_c.trdy                ▷ r_c.trdy ≡ {⟨g_0, δ_0⟩, ... ⟨g_N, δ_N⟩}
 8: function ExpandSubtree(node, visited)
 9:     if node ∈ visited then         ▷ cycle detected when node is visited for second time
10:         return {}
11:     else if node is childless then ▷ node is a constant (e.g.{⟨⊤, 1⟩}) and requires no expansion
12:         return node
13:     else                           ▷ need to expand children before creating guarded bound set
14:         visited ← visited ∪ node
15:         r_n ← {}                   ▷ initialize guarded bound set for node
16:         r_left ← ExpandSubtree(left, visited)
17:         r_right ← ExpandSubtree(right, visited)
18:         if node = Max(left, right) then
19:             for ⟨g_i, δ_i⟩ ∈ r_left, ⟨g_j, δ_j⟩ ∈ r_right do
20:                 r_n ← r_n ∪ ⟨g_i ∧ g_j, max(δ_i, δ_j)⟩   ▷ add to guarded bound set for node
21:         else if node = Plus(left, right) then
22:             for ⟨g_i, δ_i⟩ ∈ r_left, ⟨g_j, δ_j⟩ ∈ r_right do
23:                 r_n ← r_n ∪ ⟨g_i ∧ g_j, δ_i + δ_j⟩       ▷ add to guarded bound set for node
24:         else if node = Ite(predicate, left, right) then
25:             for ⟨g_i, δ_i⟩ ∈ r_left do
26:                 r_n ← r_n ∪ ⟨g_i ∧ predicate, δ_i⟩       ▷ add to guarded bound set for node
27:             for ⟨g_j, δ_j⟩ ∈ r_right do
28:                 r_n ← r_n ∪ ⟨g_j ∧ ¬predicate, δ_j⟩      ▷ add to guarded bound set for node
29:         r_n ← Prune(r_n)
30:         return r_n
```

Limitations in Deriving Blocking Bounds

This work presents recurrence relations that are found to be useful for deriving blocking bounds in the motivating example networks, but these specific recurrence relations are not precise enough to handle all possible networks. Property Θ (Eq. 6.14) is formulated such that its failure will indicate when these recurrence relations are insufficient for a given network. Furthermore, a counterexample to Θ can guide the development of rules that lead to a more precise set of recurrence relations. Some situations that would require modified recurrence relations are highlighted here.

One abstraction used in the recurrence relations is to consider only whether a queue is full or empty, and not the number of items in the queue. A similar abstraction is made in proving deadlock free-

dom by Verbeek *et al.* [102]. The significant change in this chapter's approach relative to Verbeek's is that it goes beyond deadlock freedom to include numeric progress bounds in the reasoning. Yet, like Verbeek's work, this formulation is sound but incomplete in light of the abstraction.

A second conservative abstraction in the recurrence relations presented is that no assumptions are made about data values of packets. This prevents the recurrence relations from giving any bounds on readiness outputs of the switch primitive. Refinement would be needed in a network where progress of a data packet depends on switch output[3]. It is suggested to handle this situation with manual refinement, such as replacing the recurrence relations of the switch with ones that take consider the data value of the input packet. Ongoing work by Viktorov and Gotmanov [104] aims to overcome this limitation by propagating rules that can be automatically refined to handle cases such as this. Counterexample-guided abstraction-refinement techniques [29] could also be used.

6.5 Experimental Methodology

The methodology used across all experiments in this chapter is described here. The xMAS models are created within a C++ framework[4], with primitives as objects. Progress lemmas and age lemmas are added automatically, and flattened word-level Verilog is generated with all properties added as assertions. The Verilog is bit-blasted to an and-inverter-graph[5] (AIG) using the VeriABC flow [67]. Properties are verified on the AIG using the bit-level model checker ABC [15][6] on a 2.4GHz Intel Core i5 processor with 4GB of RAM. The bounded model checking[7] (BMC), property directed reachability[8] (PDR) [42], and k-induction[9] calls are performed by ABC.

Evaluating Looseness of T_L with Bounded Model Checking

The global age bound (T_L) that is implied by the age lemmas can be loose. The tight latency bound for a given network is not generally known, but the looseness of T_L is bounded by comparison to T_{FEAS}, where T_{FEAS} is the smallest number such that $\mathcal{N} \models \Phi^G_{T_{FEAS}}$ cannot be disproved by BMC within allotted resource bounds. T_{FEAS} is found by iteratively increasing T and using BMC to disprove each Φ^G_T until reaching the first value of T that cannot be disproved; that value serves as T_{FEAS}. There cannot exist a tighter bound than T_{FEAS} because $\Phi^G_{T_{FEAS}-1}$ is disproved with a counterexample. However, due to the incompleteness of BMC $\Phi^G_{T_{FEAS}}$ may not be proved, and could instead be an artifact of the BMC resource limits.

[3] For example, consider a network in which a data packet is one input to a join primitive and the second input comes from a switch output. If data abstraction precludes determining whether the switch output ever becomes ready to initiate, then no finite bound on progress can be inferred for the packet waiting at the join input.

[4] `https://github.com/danholcomb/xmas-front-end`

[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; write_counter -n foo.cex;"

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

[9] ABC commands "read_aiger foo.aig; orpos; ind -avw; bmc3 -F *k*;" where *k* is the depth used by the ind command

Efficient Encoding of Packet Ages

The formulation of an age bound property requires that the age of a packet in slot i (denoted $age(q_i)$) can be checked as a simple safety property; in other words, the age of a packet can be evaluated in a single state of $\mathcal{N}$. Two different simple safety encodings for age are given in Section 4.4, termed “timestamp encoding” and “stopwatch encoding”. While semantically equivalent, experimental results in this section show several orders of magnitude speedup in inductive verification when using stopwatch encoding instead of timestamp encoding. The primary difference between the encodings, and the apparent cause of the speedup, is that stopwatch encoding represents ages canonically whereas timestamp encoding does not.

The runtimes for timestamp and stopwatch encodings are compared using a network with a non-deterministic source injecting packets into a single queue of depth 6 that is drained by a sink with a blocking bound of 5 (i.e. Fig. 6.4 with parameters set differently). Using BMC, T_{FEAS} is found to be 36, and for each encoding the property $\Phi^G_{36} \wedge \Psi$ is verified across different values of the largest representable time t_{max}. The plot in Fig. 6.3 compares the two encodings. When t_{max} is increased from 2^6 to 2^{12}, the runtime to prove the property with timestamp encoding increases by 13x, while the runtime to prove the same property with stopwatch encoding increases by only 2x. Similar relative performance for the two encodings is observed on a variety of problems. All experiments in the remainder of this chapter implicitly use stopwatch encoding.

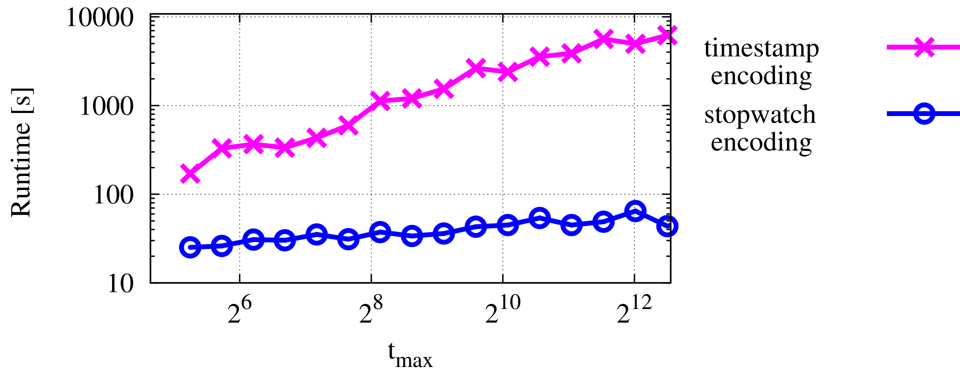


Figure 6.3: **Comparison of Stopwatch Age Encoding and Timestamp Age Encoding.** Runtime to prove $\mathcal{N} \models \Phi^G_{36} \wedge \Psi$ with ABC’s induction engine, for different values of t_{max}. The verification runtime scales better with t_{max} when using stopwatch encoding instead of timestamp encoding.

6.6 Illustrative Examples

Several examples highlight strengths and weaknesses of using latency lemmas. The primary strength is a dramatic improvement in verification runtime, and the weakness is that the bounds proved using lemmas are in some cases loose. All latency lemma reasoning including the stage

graph construction is automated for the examples in this section. The subsequent example of a ring interconnect in Sec. 6.7 is an xMAS extension where some manual reasoning is needed to create an acyclic stage graph and apply latency lemmas.

Single Queue

An example of a single packet queue (Fig. 6.4) demonstrates scalability of latency lemmas in a simple network without arbitration, routing, or flow control. The liveness bound of the sink is fixed to 3, and the depth of the queue is varied. The progress lemmas give a bound of $\delta_{abs} = 3$ for the queue's output channel, and therefore each slot in the queue maps to a stage s_j in $\mathcal{G}$ with a residence time of $d_j = 4$. The global bound implied by the lemmas is $T_L = 1 + 4 * depth$. For each queue depth it is found that $T_{FEAS} = T_L - 1$ (Fig. 6.5), indicating that T_L is one cycle larger than the tightest feasible bound. The bound T_L at each queue depth is proved inductively, using two different strengthenings of the latency property $\Phi^G_{T_L}$. The first property ($\Phi^G_{T_L} \wedge \Psi$) proves latency bound T_L without lemmas, strengthened only by auxiliary invariants Ψ. The second property ($\Phi^G_{T_L} \wedge \Psi \wedge \Phi^L \wedge \Theta$) proves the same bound strengthened by the age lemmas (Φ^L) and progress lemmas (Θ). Without latency lemmas, the induction depth required for the proof is never less than T_{FEAS}. With latency lemmas, the induction depth is 4 independent of the queue depth, demonstrating the compositionality of the approach. In a queue with depth 10, a latency bound of 41 cycles is proved in less than 2 seconds of verification runtime with latency lemmas versus 267 seconds without.

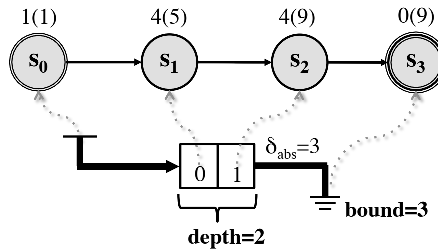


Figure 6.4: **Queue network with Stage Graph $\mathcal{G}$ Shown Above.** For the particular *depth* of 2 that is shown, the global age bound implied by the lemmas (T_L) is 9.

Credit Loop

The credit loop (Fig. 6.1) introduced in Sec. 6.2 is now revisited in more detail. Experiments are performed to explore the scalability of the latency lemma approach, different verification engines, and the tradeoff of verification runtime versus tightness of proved bounds. A credit loop has a numeric invariant ψ_{num} [21] (Eq. 6.35) asserting that each outstanding credit corresponds to exactly one available token or ingress queue packet; this numeric invariant is included as part of auxiliary invariant Ψ.

$$\psi_{num} := n_0 + n_1 = n_2 \tag{6.35}$$

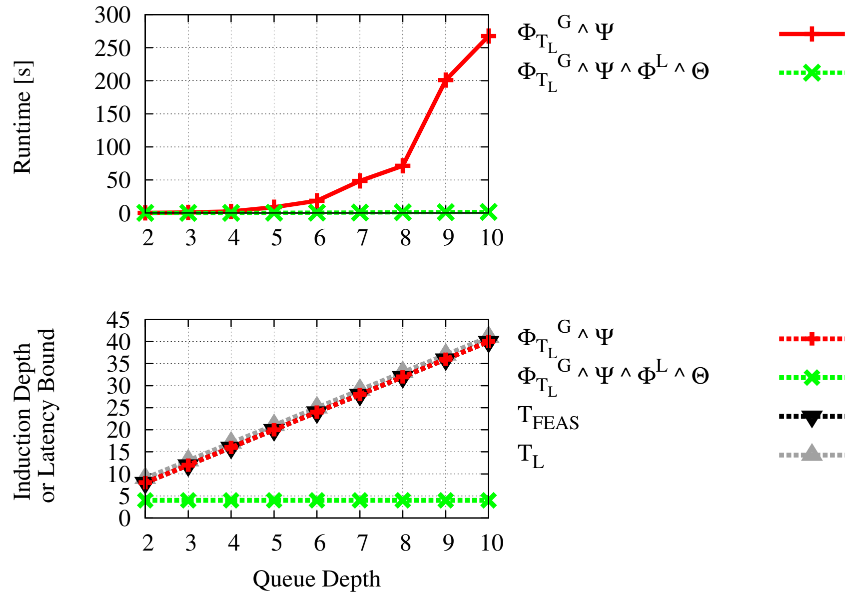


Figure 6.5: **Runtime versus Problem Size in Queue.** Sweeping the queue depth in a network comprising a single queue, and a sink with liveness bound of 3. The Y-axis in the lower plot has a slightly different meaning for each of the plotted datasets: For the two properties, the Y-axis is the induction depth required to prove them; for T_{FEAS} and T_L, the Y-axis is a latency bound (either the tightest feasible bound, or the bound proved by the lemmas).

Sweeping the Depth of Queues

As the depth of the credit loop queues are swept from 2 to 10 (Fig. 6.6), the bounds implied by the lemmas (T_L) at each depth exceed the tightest feasible bound (T_{FEAS}) on account of the conservativeness of the progress lemmas. As in Sec. 6.2, the bound of the sink is 5. The inclusion of latency lemmas yields inductive latency proofs in 8 frames of unrolling and less than 11 seconds of runtime for all depths. For a queue depth of 10, including the lemmas gives a speedup of 120x.

Comparing Proof Engines

Induction is evaluated against the PDR verification engine when the latency property is formulated with and without the strengthening of the latency lemmas. In this experiment, the queue depths are fixed to 6 and the sink bound is again 5. Attempts are made to prove 2 different bounds; the first is the tightest feasible bound (T_{FEAS}), and the second is the looser bound (T_L) implied by the latency lemmas. The results are shown in Tab 6.1. When proving tight bound T_{FEAS}, the PDR engine gives a 3x speedup over induction, and adding latency lemmas does not significantly impact runtime. When proving the looser bound T_L, adding latency lemmas causes a dramatic speedup in inductive verification. The speedup is 34x compared to induction without the lemmas, and over 7x compared to PDR with or without lemmas. The speedup is caused by the latency lemmas making the proof compositional and hence provable in only 8 frames of unrolling.

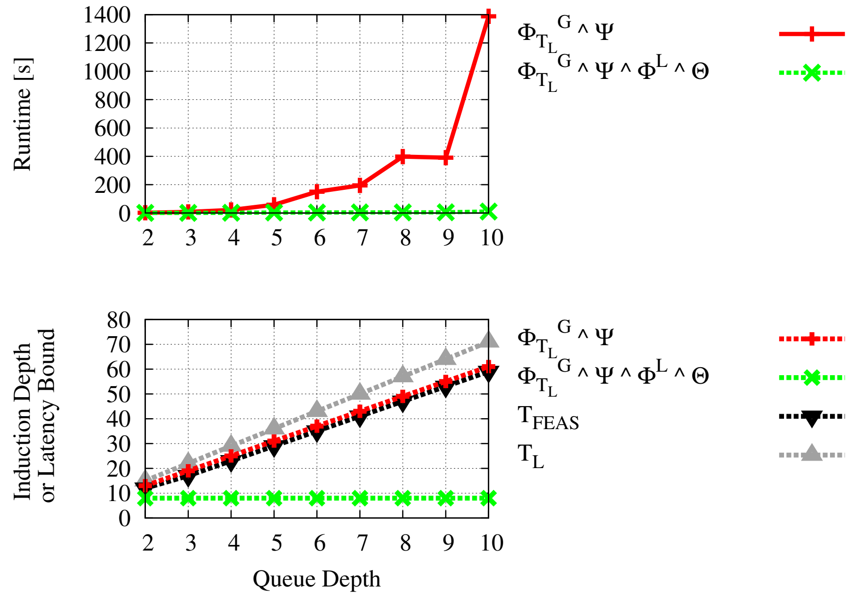


Figure 6.6: **Runtime versus Problem Size in Credit Loop.** Comparing runtime and induction depth for proving latency bounds with and without latency lemmas while varying the depth of the queues in a credit loop (Fig 6.1).

Precision vs Scalability

When using latency lemmas, there exists a tradeoff of inductive verification runtime against looseness of proved bounds. This generalizes the speedup observed in Tab 6.1 when proving the looser bound T_L instead of tight bound T_{FEAS}. The tradeoff is shown by proving individually each bound from T_{FEAS} to $T_L + 5$ (Fig. 6.7). The black vertical line indicates the tight bound $T = T_{FEAS} = 35$ and the gray indicates $T = T_L = 43$. As the latency bound increases from T_{FEAS} to T_L, the property strengthened by the lemmas gets progressively easier to prove, as evidenced by the reduction in both verification runtime and the number of frames needed for the proof. The points where the plotted data cross the black and grey vertical lines correspond to the four rows in Tab. 6.1 that use k-induction as the verification engine.

Virtual Channel

The virtual channel model in Fig. 6.8 comprises two credit loops that are independent except for a shared channel e. Non-deterministic sources inject packets at a_0 and a_1; packets injected on channel a_0 are routed to d_0, and a_1 to d_1. To appropriately route packets from each source, function primitives "tag" the packets by appending a single 0 or 1 bit before they cross the shared channel, and the target's switch primitive routes incoming packets according to this tag. The tags are removed by additional function primitives before entering their respective ingress queues. Similar to the credit loop example, the numeric invariant ψ_{num} (Eq. 6.36) is included in the auxiliary

	Runtime (s)	**Frames**	**Cex**	**Engine**	**Property**
disprove bound 34	52.79	42	Y	bmc	Φ^G_{34}
	1045.62	200	-	bmc	Φ^G_{35}
	Runtime (s)	**Frames**	**Proved**	**Engine**	**Property**
verify bound $T_{FEAS} \equiv 35$	118.79	37	Y	kind	$\Phi^G_{35} \wedge \Psi$
	95.89	37	Y	kind	$\Phi^G_{35} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	29.57	41	Y	pdr	$\Phi^G_{35} \wedge \Psi$
	27.97	41	Y	pdr	$\Phi^G_{35} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	Runtime (s)	**Frames**	**Proved**	**Engine**	**Property**
verify bound $T_L \equiv 43$	150.97	37	Y	kind	$\Phi^G_{43} \wedge \Psi$
	3.41	**8**	**Y**	**kind**	$\Phi^G_{43} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	30.61	40	Y	pdr	$\Phi^G_{43} \wedge \Psi$
	25.73	47	Y	pdr	$\Phi^G_{43} \wedge \Psi \wedge \Phi^L \wedge \Theta$

Table 6.1: **Comparing Verification Engines on Credit Loop.** Proving latency bounds for credit loop with queue depths of 6 and a sink blocking bound of 5. The tightest feasible bound (T_{FEAS}) is 35 cycles and the the bound implied by the lemmas is 43 cycles. The latency lemmas do not significantly effect runtime when proving the tighter bound, but allow the looser bound of 43 cycles to be proved in only 3.41 seconds and 8 frames.

invariant Ψ.

$$\psi_{num} := (n_0 + n_1 = n_2) \wedge (n_3 + n_4 = n_5) \tag{6.36}$$

When all queues in the virtual channel are assigned a depth of 5, and the sink bounds are 3 and 6 respectively, the stage graph $\mathcal{G}$ that is generated is shown in Fig. 6.8a. The global latency bound (T_L) implied by the latency lemmas is therefore 41 on account of stage s_{10}. Latency verification runtimes for the virtual channel network are shown in Tab. 6.2. The tightest feasible bound (T_{FEAS}) is discovered using BMC to be 35 cycles, and the PDR engine is faster than k-induction when proving T_{FEAS}. When proving the looser bound (T_L), strengthening the property using latency lemmas allows the induction engine to prove the property 4x faster than any other approach at verifying the same bound. Fig. 6.9 generalizes the tradeoff between verification runtime and tightness of proved bound that is made possible by the latency lemmas.

Token Bucket Regulator

Traffic metering is an alternative to the credit-based flow control used in the credit loop and virtual channel examples. One example of a metering circuit is a token bucket regulator, in which

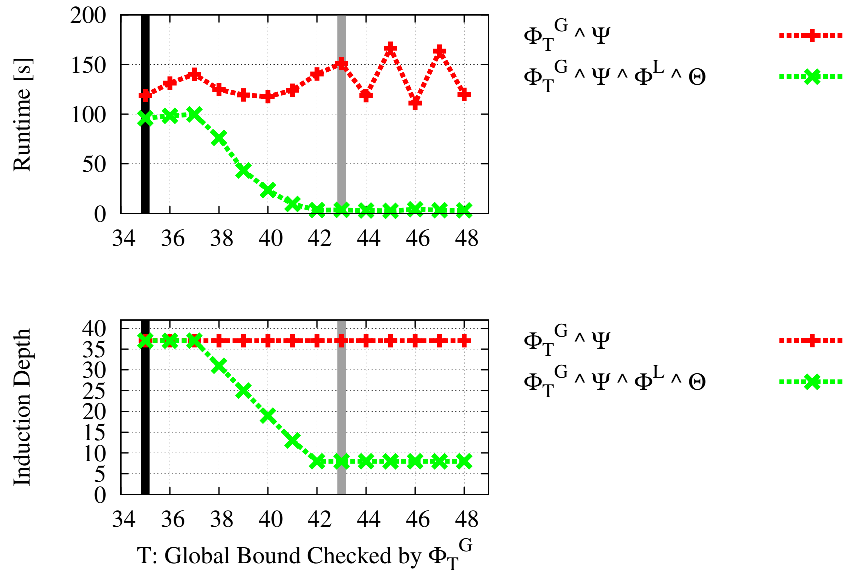
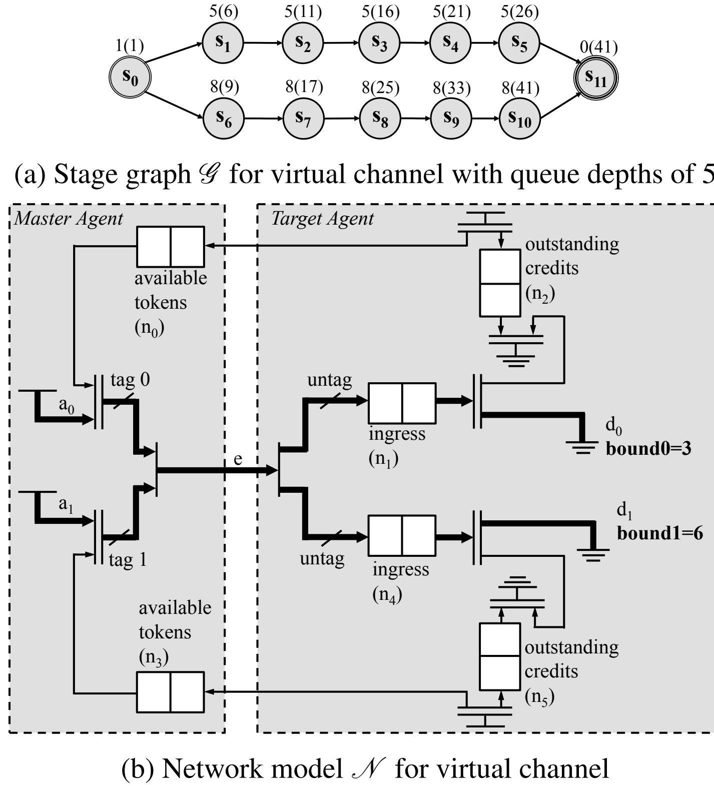


Figure 6.7: **Verification Runtime versus Proved Latency Bound in Credit Loop.** When proving a latency bound property strengthened by latency lemmas, a larger latency bound (T) leads to a reduced runtime and reduced induction depth.

tokens are periodically added to a fixed sized bucket, and consumed whenever a packet is sent. As discussed in Sec. 4.5 in the context of traffic modeling, the size of the token bucket limits the burstiness of the traffic because packets must wait for tokens once the bucket is emptied. The period at which new tokens are added to the bucket serves to constrain the average injection rate.

In this chapter, a token bucket is implemented differently than in Chapter 4 so as not to require periodic token sources. Here a token bucket is implemented in a master agent using an eager token source and two parallel branches of token queues (Fig. 6.10); injected packets from the packet source consume a token from each branch of the token bucket and therefore can only be injected when both branches have tokens in their bottommost positions. The numeric invariant ψ_{num} (Eq. 6.37) arises due to the parallel branches of the token bucket. The left branch is a single queue that sets the capacity of the token bucket to σ; this is the largest number of packets that can be injected consecutively before the bucket becomes empty. The sequence of ρ queues with depth 1 controls how quickly the bucket can be replenished. If a token is consumed from each branch the source will add a new token to each branch in the next cycle, but that token will not be usable during the next ρ cycles while it propagates through the right branch. Therefore the $i+\sigma^{th}$ data packet cannot be sent within ρ cycles of the ith data packet; this limits the long-term average injection rate to σ/ρ.

$$\psi_{num} := n_0 = n_1 \tag{6.37}$$



(a) Stage graph $\mathscr{G}$ for virtual channel with queue depths of 5

(b) Network model $\mathscr{N}$ for virtual channel

Figure 6.8: **Virtual Channel Network $\mathscr{N}$ and Stage Graph $\mathscr{G}$.** (a) All packets injected on a_0 are routed toward sink d_0, and packets injected on a_1 are routed toward d_1. (b) Stage graph for virtual channel network with all queues assigned depth 5. The 5 slots of the upper ingress queue map to stages s_1 through s_5, and the 5 slots of the lower ingress queue map to stages s_6 through s_{10}.

Experiments are performed using a token bucket configured with $\sigma = 2$ and $\rho = 10$, for a maximum burst size of 2 and a long-term average injection rate of 1 packet per 5 cycles. The target's ingress has depth of 5, and its sink has bound of 4. The sink will drain an ingress packet at least once per 5 cycles, matching exactly the maximum sustainable injection rate of the token bucket. Because the injection rate matches the sink rate, the ingress can only fill due to bursty traffic; the token bucket limits the burst size to 2, thus preventing the ingress from ever filling with packets. Therefore only the 2 slots at the head of the ingress can ever be used, and the other 3 slots are unused in all reachable states of $\mathscr{N}$.

Our reasoning using stage graph $\mathscr{G}$ infers a bound of $T_L = 26$ because it is unaware of the maximum realizable ingress capacity of 2; yet the tight bound is much smaller ($T_{FEAS} = 9$) on account of it. Since a full ingress will take 25 cycles to drain given the sink rate, proving any bound much smaller than T_L using induction will need to block off unreachable initial states where the ingress is full. When checking a bound of 10 cycles inductively using property Φ^G_{10}, initial states with a full ingress lead to latency violations in 10 cycles and are then blocked off. Yet, when checking a larger bound of (e.g.) 15 cycles, the blocking off does not occur until the 15th cycle, and this

	Runtime (s)	Frames	Cex	Engine	Property
disprove bound 33	90.95	40	Y	bmc	Φ^G_{33}
	3348.12	200	-	bmc	Φ^G_{34}
	Runtime (s)	**Frames**	**Proved**	**Engine**	**Property**
verify bound T_{FEAS} $\equiv 34$	217.36	35	Y	kind	$\Phi^G_{34} \wedge \Psi$
	166.72	35	Y	kind	$\Phi^G_{34} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	51.53	35	Y	pdr	$\Phi^G_{34} \wedge \Psi$
	49.07	38	Y	pdr	$\Phi^G_{34} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	Runtime (s)	**Frames**	**Proved**	**Engine**	**Property**
verify bound T_L $\equiv 41$	276.78	35	Y	kind	$\Phi^G_{41} \wedge \Psi$
	12.23	**9**	**Y**	**kind**	$\Phi^G_{41} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	66.27	35	Y	pdr	$\Phi^G_{41} \wedge \Psi$
	47.36	44	Y	pdr	$\Phi^G_{41} \wedge \Psi \wedge \Phi^L \wedge \Theta$

Table 6.2: **Comparing Verification Engines on Virtual Channel.** Results correspond to virtual channel model shown in Fig. 6.8b, with all queues having depth 5, and sink bounds of 3 and 6. The tightest feasible bound (T_{FEAS}) is 34, and the bound implied by the latency lemmas (T_L) is 41.

leads to a higher verification runtime. This trend is observed in Fig. 6.11 where the runtime and induction depth to prove a bound of T increase as T is swept from T_{FEAS} to $T_L + 5$. However, once the latency bound approaches T_L, the proof succeeds without needing to block off initial states where the ingress is full. The difficulty of the proof is then equivalent to a queue of depth 5 with a non-deterministic source, and the runtime and induction depth drop accordingly.

6.7 Non-Stallable Ring Interconnect

A ring network [38] is a topology for routing traffic amongst a number of agents. Each agent in the ring comprises arbitration logic, a ring queue slot, and an ingress queue (Fig. 6.12). Packets reach their destinations by circling around the ring until being admitted into their destination agent's ingress. The ring network is parameterized by the number of agents, depth of the ingress queues, and the liveness bound of each agent's sink.

A packet that is injected at agent i and destined for agent j will first occupy the ring slot of agent i. The packet circles the ring thereafter (i.e. occupying slots $6, 7, 8, 6, \ldots$ in Fig. 6.12) and requests admission to the ingress whenever arriving at agent j. If the request is denied, the packet *bounces* back onto the ring to repeat the request after making a trip around the ring. Unfair arbitration logic prioritizes traffic in the ring over traffic attempting to enter the ring from a source. This

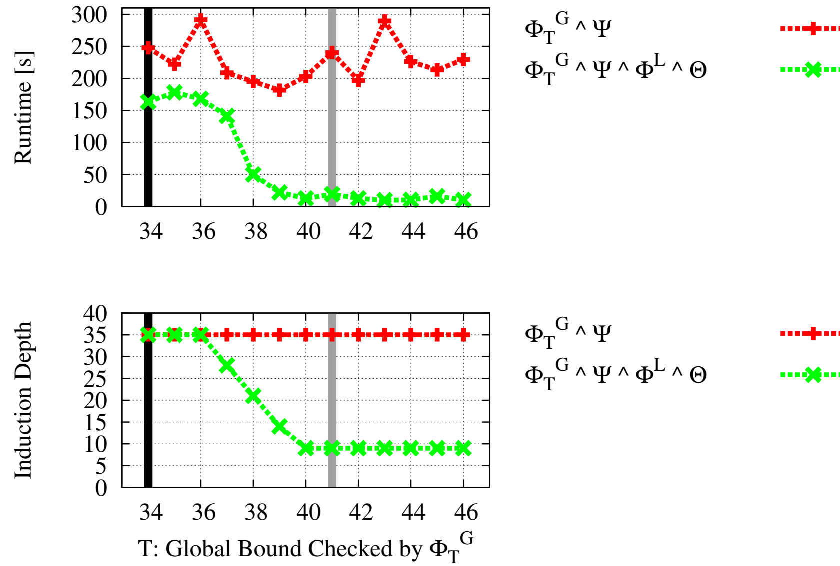


Figure 6.9: **Runtime versus Proved Latency Bound in Virtual Channel.** When sweeping the checked latency bound, as the bound approaches T_L, the induction runtime and induction depth are both reduced on account of the latency lemmas.

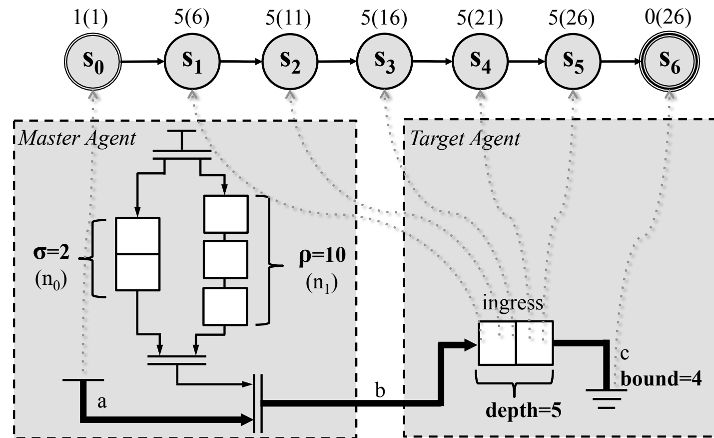


Figure 6.10: **Token Bucket Traffic Metering Circuit.** Parallel queues collectively constrain the burstiness (σ) and average rate (ρ/σ) of traffic injected by the master.

unfair arbitration ensures that packets in the ring are never blocked, but it does permit sources to be blocked indefinitely.

Each packet that a source injects into the ring is non-deterministically assigned a destination address between 0 and $n-1$ that indicates the agent to which it should be routed. This non-deterministic choice is implemented using the methodology of Sec 4.5. The destination is encoded

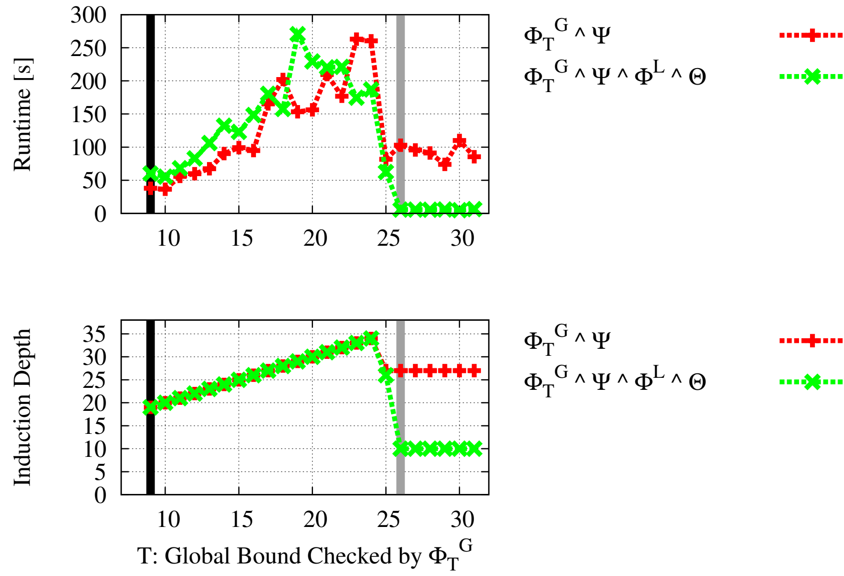


Figure 6.11: **Verification Runtime versus Proved Latency Bound in Token Bucket.** Comparing runtime and induction depth required to prove different latency bounds in token bucket network (Fig. 6.10). The difficulty of the problem increases as the global bound T is increased, but the latency lemmas simplify the problem once the bound approaches T_L

in a designated field of the packet data, analogous to how destinations are encoded for head flits in Tab. 4.1. For a packet stored in queue slot i, let $dst(q_i)$ denote its destination address. The auxiliary inductive invariant Ψ includes ψ_{dst} (Eq. 6.38) to block off unreachable states where packets in the network have invalid destinations.

$$\psi_{dst} := \bigwedge_{i \in [0, M-1]} used_i \implies dst(q_i) \in [0, n-1] \tag{6.38}$$

Implementation of a Ring Agent

Each agent in the ring is created in xMAS design style using modified versions of the basic xMAS primitives (Fig. 4.2) to implement reservations and unfair arbitration. When a packet is transferred from one ring agent to the next, the first primitive encountered is a switch that routes the packet downward toward the admission logic if this agent is the packet's destination, and upward to the bypass channel otherwise. Packets that are routed to the admission logic encounter a demultiplexer that is controlled by sequential reservation logic. The state of the sequential reservation logic and the number of free slots in the ingress determine whether the packet is admitted or bounced. A merge primitive then propagates onward a bounced packet or bypass packet, or no packet at all.

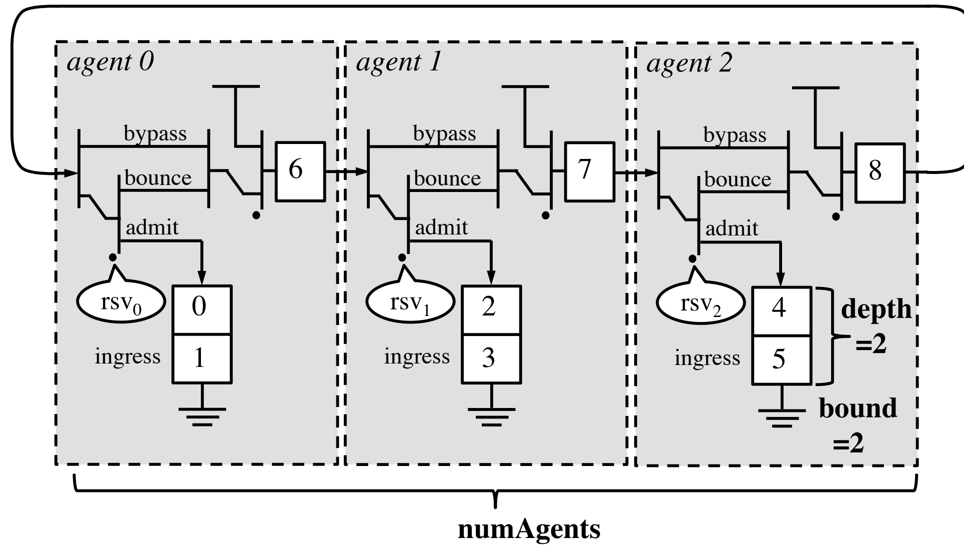


Figure 6.12: **Parameterized Ring Network.** Ring network shown is instantiated with 3 agents and an ingress queue depth of 2. All channels in ring network carry packets, and control is implemented using sequential reservation logic instead of tokens.

Finally, a merge primitive unfairly gives priority to any packet propagating through the ring, and allows the source to inject packets only if there is no competing packet on the ring.

Receive Reservation Logic

A naïve ring implementation can have infinite latency even though 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 [70] are a mechanism to enforce fairness; together with bounded liveness of sinks, receive reservations ensure that packets on the ring have finite latency bounds. The receive reservation scheme used by the ring agents is described here:

1. Each agent can issue a single receive reservation. If an agent's reservation is available, then the agent issues it to any packet that is bounced (due to a full ingress queue). The next ingress slot to become free is reserved for this packet.

2. If the agent has an outstanding receive reservation, packets without the reservation are denied entry to the ingress queue unless more than 1 slot is free.

3. When a packet with a receive reservation returns to its destination agent after circling the ring, it is granted entry to the ingress queue if any slots are free, and the reservation then becomes available for other packets. If no slots are free, the reservation is renewed by the packet and remains unavailable.

The receive reservation logic for each agent in a 3-agent ring implements the state machine shown in Fig. 6.13. The reservation is tracked by a state variable, denoted rsv_i for agent i, that operates as a sort of counter. When the receive logic state (rsv_i) is n, it indicates that the packet with the reservation will return in n cycles. When the state reaches 0, the next arriving packet on the ring is the same one for which the reservation was made. The state of rsv_i is $\perp$ when the reservation is available. Receive reservations are fair with respect to packets in the ring. Whenever one packet returns the reservation (see edge *return reservation* in Fig 6.13), the packet trailing it on the ring has a chance to make the reservation in the next cycle. Each packet in the ring gets a turn at making a receive reservation in order.

The dashed edge labelled *renew reservation* in Fig. 6.13 is taken when a packet holding the reservation returns to its destination agent and bounces on account of there not yet being any free slots in the agent's ingress queue. This can only occur if the sink's blocking bound is larger than the delay around the ring (i.e. the number of agents). It is assumed that the sink bound is smaller than the ring delay, and therefore the dashed edge is ignored. This means that packets arriving when the reservation state (rsv_i) is 0 will always have a free ingress slot and be admitted to the ingress.

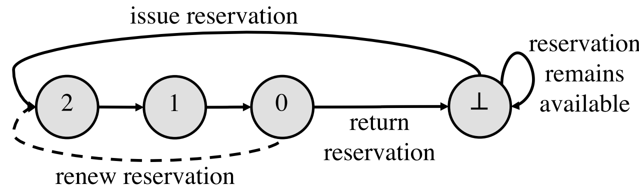


Figure 6.13: **Receive Reservation State Machine.** The state machine shown is implemented within every agent in a 3-agent ring. The reservation state (rsv_i) is $\perp$ when the reservation is available, and is 0,1, or 2 when the reservation is outstanding. When rsv_i is 0, the packet holding the reservation is the next to arrive, and when rsv_i is 2, the reservation has just been made for bounced packet that now occupies the ring slot of agent i.

Creating Age Lemmas using Stage Graph $\mathscr{G}$

The location of packets in the ring network is not a precise enough indicator of progress to create an acyclic stage graph because a packet can occupy the same slot many times as it circles the ring. Progress of a packet circling the ring is instead marked both by changes in the location of the packet, and changes in the reservation state of its destination agent. In an n-agent ring, every combination of the n ring slots, the n packet destinations, and the $n+1$ reservation states, must correspond to a stage in $\mathscr{G}$, so the total number of stages for the ring slots is $n \times n \times (n+1)$. For clarity, the explanation here deals only with the $n \times (n+1)$ stages for packets that have agent 2 as their destination; in implementation all destinations are considered.

Using the 3-agent ring as an example, composing the reservation state machine of Fig. 6.13 (without dashed edge) with the packet's behavior of advancing to the next ring slot in every cycle produces the state machine of Fig. 6.14. This product machine is the key to creating a stage graph for packets in the ring. Each state in Fig. 6.14 has two labels, the first is the state of the reservation state

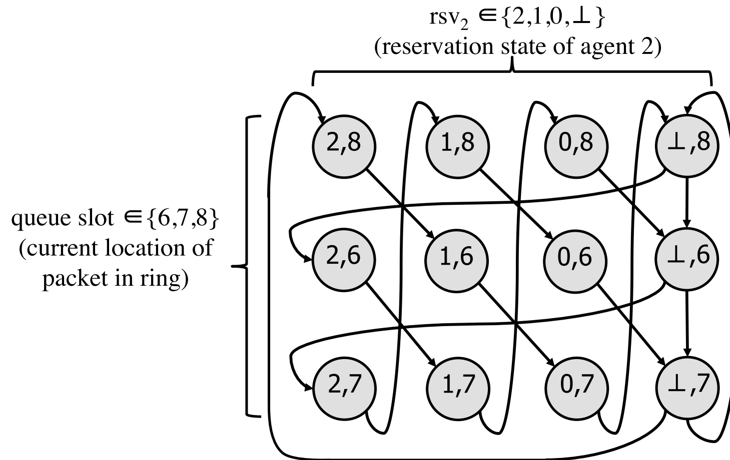


Figure 6.14: **Product Automaton of Receive Reservation and Occupied Ring Slot.** Each state corresponds to one reservation state and the index of a currently occupied ring slot from the 3-agent ring in Fig. 6.12.

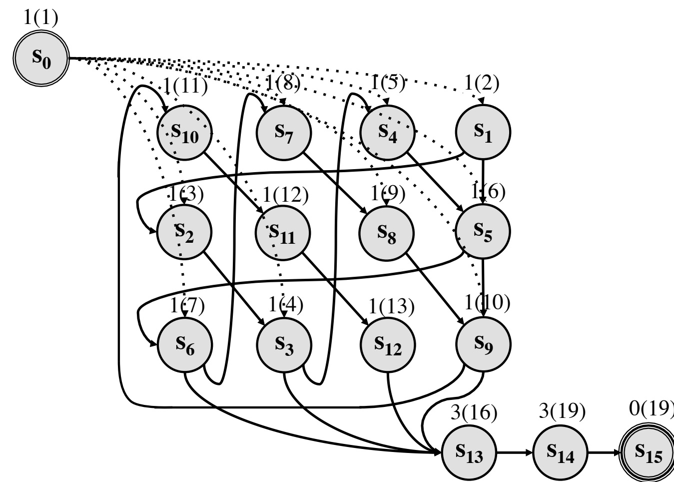


Figure 6.15: **Stage Graph $\mathscr{G}$ for the 3-agent Ring Network.** Above each stage s_j is stage residence time d_j and age bound t_j in parentheses

machine (Fig. 6.13) and the second is the index of a ring slot (Fig. 6.12). As a packet destined for agent 2 moves around the ring, in every cycle it maps to some state of this product automaton. For example, state $(0,7)$ in Fig. 6.14 is the state that a packet in slot 7 maps to when the receive reservation of agent 2 has state $rsv_2 = 0$. The mapping from packets to states in Fig. 6.14 could serve as an indicator of progress if only the product automaton were acyclic. An acyclic stage graph is obtained from the product automaton by showing that there are pairs of edge-connected states in Fig. 6.14 that no single packet can map to. When these transitions are removed, an ordering among progress stages is revealed.

- $(0,7) \rightarrow (\perp,8)$ can never be made by a packet destined for agent 2 because it corresponds to a packet bouncing at agent 2 (from slot 7 to slot 8) while agent 2 has its reservation returned (i.e. it transitions to state $rsv_2 = \perp$ that indicates that a reservation is available). The transition is impossible because bouncing and returning a reservation are exclusive; only packets admitted to the ingress cause the reservation to return to the available state.

- $(\perp,7) \rightarrow (\perp,8)$ can never be made by a packet destined for agent 2 because it corresponds to a packet that bounces (from slot 7 to slot 8) while an available reservation remains available. The transition is impossible because any packet bouncing while the reservation is available would result in the reservation being issued to it.

Without the two transitions described above, Fig. 6.14 becomes acyclic and can be used to order the progress stages of a packet in the ring. The product automaton of Fig. 6.14 becomes the stage graph of Fig. 6.15 by simply removing the two unrealizable transitions, and adding stages for sources, ingress slots, and the sink. The mapping from queue slots in the ring network to stages in Fig. 6.15 is given by the age lemmas in Tab. 6.3.

stage s_j Fig. 6.15	d_j	age lemma $\phi(i, p_{i,j}, t_j)$		
		i	$p_{i,j}$	t_j
s_0	1	-	-	1
s_1	1	8	$dst(q_8) = 2 \wedge rsv_2 = \perp$	2
s_2	1	6	$dst(q_6) = 2 \wedge rsv_2 = 2$	3
s_3	1	7	$dst(q_7) = 2 \wedge rsv_2 = 1$	4
s_4	1	8	$dst(q_8) = 2 \wedge rsv_2 = 0$	5
s_5	1	6	$dst(q_6) = 2 \wedge rsv_2 = \perp$	6
s_6	1	7	$dst(q_7) = 2 \wedge rsv_2 = 2$	7
s_7	1	8	$dst(q_8) = 2 \wedge rsv_2 = 1$	8
s_8	1	6	$dst(q_6) = 2 \wedge rsv_2 = 0$	9
s_9	1	7	$dst(q_7) = 2 \wedge rsv_2 = \perp$	10
s_{10}	1	8	$dst(q_8) = 2 \wedge rsv_2 = 2$	11
s_{11}	1	6	$dst(q_6) = 2 \wedge rsv_2 = 1$	12
s_{12}	1	7	$dst(q_7) = 2 \wedge rsv_2 = 0$	13
s_{13}	3	4	**true**	16
s_{14}	3	5	**true**	19
s_{15}	0	-	-	19

Table 6.3: **Age Lemmas for 3-agent Ring.** The age lemmas are shown only for packets with agent 2 as their destination. The first column is the stage in $\mathscr{G}$ (from Fig. 6.15) that corresponds to the age lemma described on the row.

Latency Verification Results for Ring Interconnect

The latency lemma approach is evaluated on a 3-agent ring and an 8-agent ring, each with ingress depth of 2 and sink bound of 2. For the 3-agent ring, the tightest feasible bound (T_{FEAS}) is 18, and the bound implied by the latency lemmas (T_L) is 19; for the 8-agent ring, T_{FEAS} is 78 and T_L is 79. The runtimes and number of frames for proving a bound of T_L on each ring, with and without latency lemmas, are shown in Tab. 6.4 and 6.5. Property $\Phi^G_{T_L} \wedge \Psi$ proves the latency bound T_L without latency lemmas, and property $\Phi^G_{T_L} \wedge \Psi \wedge \Phi^L \wedge \Theta$ proves it with the lemmas added. Strengthening the global latency bound property with latency lemmas reduces the verification runtime for both k-induction and PDR in both sizes of the ring networks.

The latency bound for the 8-agent ring (Tab. 6.5) is proved within 10,000 seconds by each engine only when the lemmas are used. The induction engine is able to verify the property with the lemmas 9x faster than PDR verifies the same, and at least 130x faster than either engine does without lemmas.

	Runtime (s)	**Frames**	**Cex**	**Engine**	**Property**
disprove bound 17	24.12	20	Y	bmc	Φ^G_{17}
	868.28	200	-	bmc	Φ^G_{18}

	Runtime (s)	**Frames**	**Proved**	**Engine**	**Property**
verify bound T_L $\equiv 19$	62.34	18	Y	kind	$\Phi^G_{19} \wedge \Psi$
	1.31	**4**	**Y**	**kind**	$\Phi^G_{19} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	88.39	12	Y	pdr	$\Phi^G_{19} \wedge \Psi$
	6.57	14	Y	pdr	$\Phi^G_{19} \wedge \Psi \wedge \Phi^L \wedge \Theta$

Table 6.4: **Latency Verification Runtimes for 3-agent Ring.** Proving latency bounds for 3-agent ring with ingress depth 2 and sink bound 2. The 19 cycle bound that is implied by the lemmas exceeds T_{FEAS} by only 1 cycle.

6.8 Related Work

One way of addressing QoS guarantees at the architectural level is to use resource reservation and contention-free routing [47]. Analysis can be performed manually, but formal verification is still useful for providing guarantees.

Network calculus [31] has been demonstrated as a useful tool for NoC performance analysis [107]. 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 [50], but they only address scalability problems

	Runtime (s)	Frames	Cex	Engine	Property
disprove bound 77	3901.95	80	Y	bmc	Φ^G_{77}
	10,000.00	111	-	bmc	Φ^G_{78}

	Runtime (s)	Frames	Proved	Engine	Property
verify bound $\equiv T_L$ $\equiv 79$	10,000.00	-	-	kind	$\Phi^G_{79} \wedge \Psi$
	75.28	**10**	**Y**	**kind**	$\Phi^G_{79} \wedge \Psi \wedge \Phi^L \wedge \Theta$
	10,000.00	-	-	pdr	$\Phi^G_{79} \wedge \Psi$
	662.15	73	Y	pdr	$\Phi^G_{79} \wedge \Psi \wedge \Phi^L \wedge \Theta$

Table 6.5: **Latency Verification Runtimes for 8-agent Ring.** Proving latency bounds for 8-agent ring with ingress depth 2 and sink bound 2. The 79 cycle bound that is implied by the lemmas exceeds T_{FEAS} by only 1 cycle. This latency bound can be proved within 10,000 seconds only when latency lemmas are used.

arising from the size of the network, rather than from proving a large latency bound, while the present work addresses the latter issue.

The notion of using LTL properties where all eventually properties have time bounds is also referred to as a prompt system [59]. Prior works compare liveness and safety methods for verifying grant latencies on a particular style of weighted round robin arbiter [57].

Several works have explored (unbounded) liveness verification of communication fabrics. The standard approach of verifying liveness using a liveness-to-safety transformation [6] does not scale to large networks in practice [48]. Alternative approaches include reducing deadlock conditions to a set of equations [48, 102], and proving liveness using the help of intermediate safety assertions [90].

The use of latency lemmas is conceptually similar to ranking functions [99], 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, a 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. Viktorov and Gotmanov [104] propose a theorem-proving approach to latency verification in xMAS networks that is based on ranking functions. Their inference rules are analogous to the rule-based propagation used in this work.

6.9 Conclusion

The work presented in this chapter gives 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. These latency lemmas are then used to strengthen the global latency bound property. The approach is fully automated for classes of acyclic networks constructed from basic xMAS primitives, and some manual input is

required for cyclic xMAS networks or xMAS-like networks that use an extended set of primitives. Further automating the stage graph construction for cyclic networks is a topic for future work. The latency lemma approach is applied to several examples including an industry-inspired ring design, and is shown to decrease runtime for proving latency bounds, while also decreasing the induction depth needed to prove latency bounds.

Part III

NoC Parameter Synthesis

Chapter 7

Buffer Sizing

The quality of network-on-chip (NoC) designs depends crucially on the size of buffers. Buffers impose a significant area and power overhead, but are essential for ensuring high throughput and low latency. This chapter presents a new approach for minimizing the cumulative buffer size in on-chip networks, so as to meet throughput and latency requirements, given high-level specifications on traffic behavior. The overall approach is to use model checking based on satisfiability modulo theories (SMT) solvers, within an overall counterexample-guided synthesis loop.

The work in this chapter was originally published in the 2011 Design Automation and Test in Europe (DATE) conference [14]. The approach as presented in this chapter is identical to that used in the published version, but the experimental results are expanded. This work makes use of the traffic models introduced in Chapter 4.5. Contrary to Chapter 5 where the traffic models are inferred from simulation, the traffic models in this chapter are assumed to be given a priori as a specification. The traffic model is inconsequential to the buffer sizing technique that is the contribution of this chapter, and the same technique can be applied with any traffic model or even fully non-deterministic packet sources. The regulated traffic model is used only to create a more interesting sizing problem, as fully-nondeterministic traffic leads to solutions with buffers that are drastically over-provisioned for common use cases.

7.1 Introduction

Recall that an NoC architecture consists of a network of interconnected routers, where each router communicates with neighboring routers and a processor core or a specialized IP block. Inter-node communication is performed by the transmission of data packets through routers, where they are typically stored in buffers. This chapter addresses a key problem in the design and implementation of NoCs — the minimization of total buffer size while still guaranteeing a particular quality of service.

Buffers play a critical role in NoC design: increasing the sizes of the buffers can significantly reduce the average latency of packets and increase the throughput. However, even with scalable

mesh architectures where routers only exchange data with their neighbors (e.g., [58]), the size of the ingress buffer for each input channel has a serious impact on the area and power of an NoC router design. For example, Hu and Marculescu [53] indicate that changing the buffer size at each input channel from 2 words to 3 words will increase the overall network area by 30%. The buffer sizing problem is further complicated by the heterogeneity of traffic patterns in NoCs. For example, more buffers must be allocated in heavily loaded channels. Hence, a technique is needed to judiciously allocate buffer capacity for each channel in accordance with a given traffic model specification.

This chapter proposes a formal technique for *minimizing the cumulative buffer size* while meeting design and performance constraints with respect to specified traffic models. Given a formal NoC model, SMT-based model checking is used to find the minimal buffer sizes that guarantee some throughput and latency for the specified traffic model. The approach is based on counterexample-guided synthesis, where an SMT-based model checker is repeatedly invoked both for synthesizing buffer sizes and for checking whether the synthesized buffer sizes guarantee the performance property for specified traffic patterns. Buffer sizings found with this approach guarantee performance for all traffic patterns that can be generated by a traffic model, without having to explicitly enumerate each one.

In summary, this chapter makes the following novel contributions:

1. Proposing a new SMT-based model checking technique, based on counterexample-guided synthesis, for minimizing the cumulative buffer size in an NoC design.
2. Showing how a formal term-level approach can effectively model NoC designs and traffic patterns.
3. Demonstrating the effectiveness of the buffer sizing technique on NoC designs involving arbitration and credit logic.
4. Experimental evaluation comparing two different representations of symbolically-sized buffers.

This chapter is organized as follows. Section 7.2 describes the models used for NoC designs and defines the buffer minimization problem. The SMT-based approach is given in Section 7.3. Experimental results are presented in Section 7.4, related literature is reviewed in Section 7.5, and Section 7.6 concludes.

7.2 Formal Model and Problem Definition

The approach in this chapter uses a network model $\mathscr{N}$ and a traffic model $\mathscr{T}$. The network model uses the router core from Chapter 4.3 and stateful and stateless xMAS primitives communicating over channels. Let $\mathscr{B}$ represent the set of xMAS queue components in the model $\mathscr{N}$. Recalling that each xMAS buffer is parameterized by its depth, each buffer $b_i \in \mathscr{B}$ has associated size,

which is the number of entries that can be stored in it. However, as the task of this chapter is to determine buffer sizes, the sizes are now symbolic constants, and the problem at hand is to determine appropriate values for these constants.

Modeling Symbolic-Sized FIFOs

New to this chapter is the use of symbolically sized queues. Symbolically sized queues require only a minor modification to the UCLID queue implementations presented in Chapter 4.2. Recall that queues can be implemented using as the underlying storage mechanism either fixed-size circular buffers (Sec. 4.2) or arbitrary-sized records (Sec. 4.2). Both options are explored in this chapter. In each queue implementation, a bit-vector state variable keeps track of the number of items stored in the queue. When this variable is equal to the size of the queue, the queue is said to be full, and *trdy* on its input channel is de-asserted to block incoming flits until slots become free. Representing the size of the queue with a symbolic constant instead of a specific constant causes the queue to be symbolically sized.

Some care must be taken when using symbolically sized queues that are implemented as circular buffers. The symbolic constant is restricted to taking values that are less than or equal to the physical size of the queue. Because the physical size of the queue is fixed in the model, an upper bound on the symbolic queue sizes must be known in advance to ensure that the circular buffer is sufficiently large. When bounded model checking is used, one conservative upper bound is the BMC depth since no more than one packet can enter the queue each cycle, but often a more practical smaller bound can be used. No such limitation exists in the record-base queue implementation, which can store an arbitrary finite number of elements; using a 24-bit symbolic constant for size limits the number of items stored to 2^{24}, but in practice the sizes used never approach this.

Traffic Model Specification

Traffic patterns for NoC designs vary widely depending on the environment in which the NoC is being used. NoC designs used in multimedia applications usually experience more regular traffic patterns. On the other hand, traffic patterns for NoC fabrics within CMP designs are less regular. It is not unreasonable to be concerned with NoC performance for specific classes of traffic patterns [38].

The traffic model $\mathscr{T}$ used in the chapter for injecting traffic into the network follows the formulation Sec. 4.5. Specifically, the constraints enforced by the traffic model in this chapter are:

- Every injected head flit is immediately followed by a tail flit in the next cycle. This is enforced using a state machine similar to the one shown in Fig. 4.7, except lacking a state to generate body flits.
- Token bucket rate constraints are enforced. The specific rate constraints vary according to the experiment performed.

A new twist on traffic models in this chapter is that counterexample traffic patterns are collected and replayed later for use in synthesis. A counterexample traffic patterns is one generated by the traffic model $\mathscr{T}$ that causes a property in network model $\mathscr{N}$ to fail. Each counterexample pattern p_i is therefore a sequence of packets transferred across the channels from $\mathscr{T}$ to $\mathscr{N}$. Traffic model $\mathscr{T}$ is removed when replaying the pattern p_i, and instead the inputs to $\mathscr{N}$ in each cycle are assigned the values from the pattern p_i.

QoS Performance Properties

While the approach of this chapter can size buffers to meet any performance property that can be formulated as a safety property, the properties considered here are *latency* and *non-blocking*.

A latency property, $\phi_{latency}$, is parameterized by a maximum number of cycles x allowed for packet transit. Following the timestamp encoding of Section 4.4, property $\phi_{latency}$ is **true** in the ith cycle if and only if no buffer slot stores a flit that was injected into the network before time $n_i - x$.

A non-blocking property, $\phi_{non-block}$, enforces that packets are never forced to wait at the interface between the traffic model $\mathscr{T}$ and the network $\mathscr{N}$. Property $\phi_{non-block}$ is **true** if, for each channel c from $\mathscr{T}$ to $\mathscr{N}$, $c.irdy \implies c.trdy$. This property is defined by Chatterjee *et al.* [21].

A correct NoC must satisfy property ϕ (Eq. 7.1) in all reachable states. The overall property ϕ can be considered a throughput specification, because $\phi_{non-block}$ specifies that traffic is always admitted into the network $\mathscr{N}$, and $\phi_{latency}$ specifies that all traffic to enter the network will also exit the network (and does so within a given time bound, although this is not required for throughput).

$$\phi := \phi_{latency} \wedge \phi_{non-block} \tag{7.1}$$

SMT-based Buffer Sizing

Given a formal NoC model $\mathscr{N}$ with symbolically-sized queues as described in Sec. 7.2, a traffic model $\mathscr{T}$ as described in Sec. 7.2, and a performance property ϕ as described in Sec. 7.2, the *buffer size synthesis problem* is to compute a buffer sizing S such that the composition of the sized NoC model and $\mathscr{T}$ satisfies ϕ.

A *buffer sizing* S is defined as a mapping from buffers to sizes $S : \mathscr{B} \rightarrow \mathbb{N}$, where $S(b_i)$ denotes the size of buffer b_i, and its cumulative size is denoted $|S| = \sum_{b_i \in \mathscr{B}} S(b_i)$. An NoC $\mathscr{N}$ with buffer sizing S is denoted $\mathscr{N}[S]$, and its composition with traffic model $\mathscr{T}$ is denoted $\mathscr{N}[S] \| \mathscr{T}$. Thus, a *correct* sizing of an NoC is one that satisfies $\mathscr{N}[S] \| \mathscr{T} \vDash \phi$.

For a sized NoC $\mathscr{N}[S]$, $\mathscr{N}[S] \| \mathscr{T} \vDash \phi$ is decided by SMT-based bounded model checking (BMC). The need for SMT arises from the presence of symbolic variables in the composite model $\mathscr{N}[S] \| \mathscr{T}$, including non-deterministic choice variables used to model traffic patterns and abstract terms or uninterpreted functions used to model packet content. Buffer synthesis is built upon the above SMT-based verification method, and the next section presents the overall SMT-based technique for optimal buffer size synthesis.

7.3 The CEBUS Approach

The counterexample-guided buffer size synthesis (CEBUS) approach finds the *minimum cumulative buffer sizing* S such that $\mathscr{N}[S]\|\mathscr{T} \vDash \phi$, and $|S|$ is minimized among all possible solutions.

This is accomplished by iteratively solving two problems:

1. *Buffer size verification (BSV):* This step tries to find a traffic pattern p that is generated by $\mathscr{T}$ and disproves $\mathscr{N}[S]\|\mathscr{T} \vDash \phi$, where S is computed in the previous BSS step, or is the initial sizing S_0. If such a traffic pattern p is found, it is added to a set P of patterns to be used for synthesis by BSS.

2. *Buffer size synthesis (BSS):* For an NoC model $\mathscr{N}$ and a set of traffic patterns P generated by $\mathscr{T}$, this step computes a buffer sizing S that has minimal cumulative size $|S|$ among all solutions to $\mathscr{N}[S]\|P \vDash \phi$. This sizing is only correct with respect to set of traffic patterns P; it may or may not be correct with respect to the more general $\mathscr{T}$, as $\mathscr{T}$ can generate many more patterns than just those in P.

As shown in Fig. 7.1, the iterations between BSS and BSV continue until the process terminates successfully upon finding a minimal buffer sizing that ensures correctness for all traffic patterns that can be generated by $\mathscr{T}$, or else terminates unsuccessfully by finding a set P of patterns for which no sizing can ensure correctness.

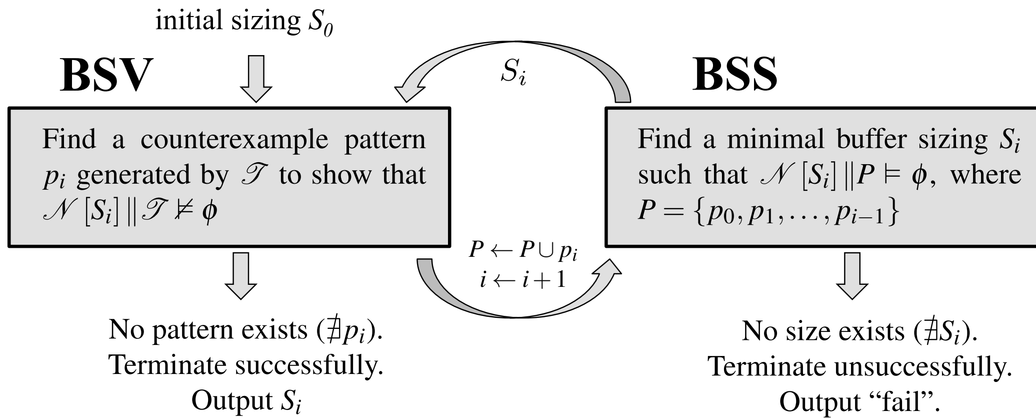


Figure 7.1: **CEBUS Procedure for Optimal Buffer Size Synthesis.** Starting with a candidate sizing S_i (or initial sizing S_0), find a traffic pattern p_i that is generated by $\mathscr{T}$ and causes $\mathscr{N}[S_i]\|\mathscr{T} \nvDash \phi$. Add this traffic pattern p_i to the set P of traffic patterns and move to the next iteration by incrementing i. Then find a buffer size assignment S_i such that $\mathscr{N}[S_i]\|P \vDash \phi$ holds. Repeat BSV using using this new sizing S_i, and continue iterating around the loop until reaching a termination condition.

Buffer Size Synthesis

The goal of buffer size synthesis (BSS) is to find a buffer sizing S such that $|S|$ is minimum and $\mathcal{N}[S] \| P \vDash \phi$ for a set of traffic patterns P that have been generated by $\mathcal{T}$. BSS is broken down into two steps, satisfiability and minimization. The satisfiability problem is referred to as BSS-SIZE and the minimization problem is referred to as BSS.

BSS-SIZE ($\mathcal{N}$,P,$size$) is the problem of determining whether there exists a sizing S smaller than $size$ such that $\mathcal{N}[S] \| P \vDash \phi$ holds. More precisely, BSS-SIZE($\mathcal{N}, P, size$) returns an S such that $\mathcal{N}[S] \| P \vDash \phi \wedge |S| \leq size$, if one exists; otherwise, it returns $\perp$.

Solving BSS-SIZE only requires a satisfiability check (instead of checking a quantified Boolean formula) because P is a finite set of traffic patterns. The SMT formula for this is constructed as the conjunction of BMC unrollings with the patterns in P applied, where a common set of symbolic constants serves as the buffer sizes across all the unrollings.

$$\exists S. \bigwedge_{p_i \in P} \mathcal{N}[S] \| p_i \vDash \phi \tag{7.2}$$

A solution to BSS-SIZE, if one exists, is a buffer sizing S such that $\mathcal{N}[S] \| P \vDash \phi$, but this S is not necessarily the minimum solution. In order to find the minimum sizing S such that $\mathcal{N}[S] \| P \vDash \phi$, BSS performs a binary search over $size$ using a sequence of calls to BSS-SIZE ($\mathcal{N}$,P,$size$). Algorithm 3 shows pseudo code for BSS. Intermediate variables $MinBufSize$ and $BufSizeLB$ keep track of the minimum buffer size seen thus far where ϕ holds, and the maximum buffer size such that $\neg\phi$, respectively; these two variables bound the range to search for a minimal solution. The BSS procedure terminates with $MinBufSize$ as the minimum cumulative buffer size. At this point, $MinBufSize = BufSizeLB + 1$, and there does not exist an S smaller than $MinBufSize$ such that $\mathcal{N}[S] \| P \vDash \phi$.

Algorithm 3: Procedure BSS($\mathcal{N}, P, LB$): Compute minimum cumulative buffer size for NoC model $\mathcal{N}$ and set of traffic patterns P.

```
1:  MinBufSize ← UB                         ▷ UB is a known upper bound on optimal size
2:  BufSizeLB ← LB       ▷ LB is a lower bound on optimal size that is already ruled out
3:  while BufSizeLB + 1 ≠ MinBufSize do
4:      size ← (MinBufSize − BufSizeLB)/2 + BufSizeLB
5:      S_RES ← BSS-SIZE(N, P, size)          ▷ check existence of sizing with size total slots
6:      if S_RES ≠ ⊥ then                      ▷ found correct sizing, sizing is best result yet
7:          S ← |S_RES|
8:          MinBufSize ← |S|
9:      else                                   ▷ no sizing exists using size total buffer slots
10:         BufSizeLB ← size
11: return S                                   ▷ optimal buffer sizing, all smaller sizes ruled out
```

Buffer Size Verification

The solution to BSS provides a minimal buffer sizing S such that $\mathscr{N}[S]\|p \models \phi$ for all traffic patterns $p \in P$, but this sizing may not ensure correctness for all p generated by $\mathscr{T}$. Buffer size verification (BSV) addresses this by checking whether $\mathscr{T}$ can generate a pattern p that disproves $\mathscr{N}[S]\|\mathscr{T} \models \phi$

BSV $(\mathscr{N}[S],\mathscr{T})$ is the problem of determining whether $\mathscr{N}[S]\|\mathscr{T} \not\models \phi$. As noted earlier, this can be solved as a BMC problem. A solution (satisfying assignment) to this problem is a traffic pattern p generated by $\mathscr{T}$ that causes $\neg\phi$. If BSV has no solution (returns $\perp$) then traffic model $\mathscr{T}$ cannot generate a pattern for sized NoC $\mathscr{N}[S]$ that will cause ϕ to fail. If BSV has a solution, it produces a traffic pattern as proof that the sized NoC is insufficient for meeting the performance property; in this case, a new sizing will need to be synthesized.

Optimal Buffer Sizing

A procedure denoted CEBUS uses a combination of BSS and BSV to find the minimum cumulative buffer size for NoC model $\mathscr{N}$ and traffic model $\mathscr{T}$. Let S_0 be the initial sizing with all buffers minimum sized, except for network interface buffers which have size of 2 to enable storing both flits of a packet. CEBUS begins by calling BSV $(\mathscr{N}[S_0],\mathscr{T})$ to generate from $\mathscr{T}$ a traffic pattern p_0 such that $\mathscr{N}[S_0]\|\mathscr{T} \not\models \phi$. Create a set of patterns P with p_0 as its only member. Then CEBUS calls BSS $(\mathscr{N},P,|S_0|)$ in order to obtain the minimum buffer sizing S_1 such that $\mathscr{N}[S_1]\|P \models \phi$ holds. Then call BSV again to find a traffic pattern p_1 to disprove the sizing S_1. If one is found, add p_1 to P and repeat BSS to find S_2. When performing synthesis for the second time, the buffer sizes from the prior synthesis round (S_1) are discarded, and only the total size ($|S_1|$) is retained as a lower size bound in the current synthesis round. This lower bound is retained across iterations because any size that did not permit a solution in a previous iteration can not permit a solution in the current iteration, on account of the set of patterns P in the current iteration being a superset of those in the previous iteration. The CEBUS loop continues to alternate between BSS and BSV until terminating due to one of the following reasons: 1. BSS $(\mathscr{N},P,|S_{i-1}|)$ returns $\perp$, which means no buffer size exists for the current set of traffic patterns P; or 2. BSV $(\mathscr{N}[S],\mathscr{T})$ returns $\perp$, meaning that no traffic pattern exists that causes $\mathscr{N}[S]\|\mathscr{T} \not\models \phi$. Algorithm 4 gives the pseudo code for this procedure.

Algorithm 4: Procedure CEBUS $(\mathscr{N}, \mathscr{T})$: Compute the optimal buffer sizing for NoC model $\mathscr{N}$ and traffic model $\mathscr{T}$.

1: $i \leftarrow 0$
2: $S_0 \leftarrow$ initial buffer sizes
3: $P \leftarrow \emptyset$ ▷ set of patterns begins empty
4: **while true do**
5: $p_i \leftarrow \mathrm{BSV}(\mathscr{N}[S_i], \mathscr{T})$ ▷ check for traffic pattern that disproves candidate sizing S_i
6: **if** $p_i \neq \bot$ **then**
7: $P \leftarrow P \cup p_i$ ▷ new pattern to consider in subsequent rounds of synthesis
8: $i \leftarrow i+1$
9: $S_i \leftarrow \mathrm{BSS}(\mathscr{N}, P, |S_{i-1}|)$ ▷ find optimal sizing for current set of patterns
10: **if** $S_i = \bot$ **then**
11: **return** "failure, no buffer sizing exists" ▷ $\nexists S_i$ such that ϕ for all P
12: **else**
13: **return** "success, optimal sizing S_i found" ▷ $\nexists p_i$ generated by $\mathscr{T}$ such that $\mathscr{N}[S_i] \wedge \neg\phi$

7.4 Experimental Buffer Sizing Results

Two case studies are performed to evaluate the CEBUS approach. The first is a small example of credit-based flow control. The larger second example is credited flow control through a chip-multiprocessor (CMP) router. Each example verifies the performance properties described in Section 7.2. Network and traffic models are created in UCLID [1] using the ATLAS [13] system to manage and flatten hierarchy. The UCLID verifier performs bounded-model checking within the CEBUS loop, using its internal SMT solver with the MiniSat SAT solver [43] as a back end. Experiments are run on a Linux workstation with 64-bit 3.0 GHz Xeon processors and 2 GB of RAM.

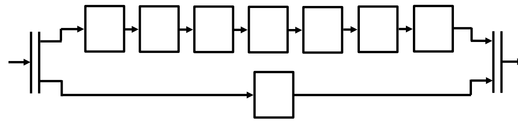


Figure 7.2: **Modeling Element for Non-pipelined Delay.** This modeling element is used to implement a delay on the token return path in buffer sizing experiments.

Credit Logic

A first experimental evaluation of CEBUS is performed on credit logic. Credit logic is a common NoC design pattern used to control channel usage within a router, as was introduced in Chapter 6. The credit logic used in buffer sizing experiments is shown in Fig. 7.3. A master router uses credited flow control to send flits to a target router. The master can only send flits when one or

more tokens are stored in the available tokens queue denoted by b_2, and one or more slots are free in the ingress queue denoted by b_3. A non-pipelined delay of 7 cycles (implemented as in Fig. 7.2) is added to the channel on which tokens are returned from target to master.

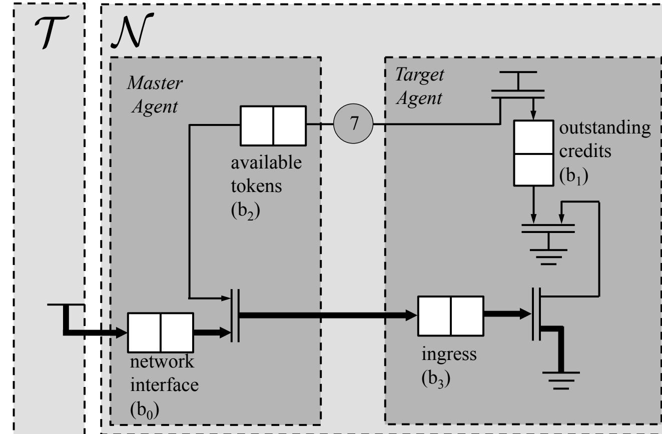


Figure 7.3: **Credit Logic Model for Buffer Sizing Experiment.** This example models credit-logic commonly used in NoCs. The formal network model $\mathscr{N}$ includes a master and target agent, with the the four buffers being sized labelled. A 7 cycle delay element, depicted here by a circle but implemented as in Fig. 7.2, is added to the token return path. Traffic model $\mathscr{T}$ injects data into the network. The channel from $\mathscr{T}$ to $\mathscr{N}$ is specified as a non-blocking channel.

The setup for the credit logic experiment is as follows. The traffic model $\mathscr{T}$ implements token bucket regulation as in Fig. 4.8a, with parameters $(\sigma,\rho) = (2,9)$, and the latency bound checked in property ϕ is 8 cycles. In the initial state of BMC, the network interface and ingress queues are empty, and the available tokens and outstanding credits queues are filled with tokens. The traffic model allows a burst of up to 2 packets (4 flits) to be sent. Because it takes 7 cycles for a token to be returned and the latency bound is 8 cycles, the latency property $\phi_{latency}$ will be violated if each flit must wait for the returned token from the previous flit. The non-blocking property $\phi_{non-block}$ disallows any solutions that would achieve low latency by blocking injections from $\mathscr{T}$ when tokens are not available. Upsizing the buffers ensures that flits do not need to wait too long for tokens, and furthermore ensures that flits will never be blocked at the input.

Table 7.1 and Table 7.2 present the SAT problem sizes and UCLID runtimes for each iteration of the CEBUS loop on the credit loop model; Table 7.1 uses the circular buffer implementation of queues (Sec. 4.2) and Table 7.2 uses the record-based implementation (Sec. 4.2). The overall trends are the same across the two queue implementation styles. The size of the SAT problem for solving BSS increases with CEBUS iterations as the set P of traffic patterns grows, while the size of the SAT problem for solving BSV is the same for all iterations. The runtime for BSV increases at later iterations of the CEBUS loop, because there are fewer traffic patterns that can violate the buffer sizes, as the sizes are already checked in synthesis against numerous challenging patterns. It is likely insignificant that CEBUS required one extra iteration to converge in the record-based queue implementation. While it is not possible to find the same sizing on different iterations of the

CEBUS loop, it is possible to find different sizings with the same total number of buffer slots. This occurs in iterations 2 and 3 in Tab. 7.1.

i	$\|S_i\|$	*Buffer-Size Synthesis (BSS)*						*Buffer-Size Verification (BSV)*					
		CNF Size		Runtime (sec)				CNF Size		Runtime (sec)			
		Vars	Clauses	SAT	Enc	SSim	Total	Vars	Clauses	SAT	Enc	SSim	Total
0	-	-	-	-	-	-	-	73k	220k	0.6	17.4	0.8	18.8
1	5	43k	130k	0.7	9.7	1.7	12.0	73k	220k	3.4	17.3	0.8	21.5
2	11	74k	223k	0.9	19.1	1.8	21.9	73k	220k	2.2	17.2	0.8	20.2
3	11	98k	294k	1.3	24.2	4.7	30.3	73k	220k	1.9	17.3	0.8	19.9
4	12	133k	399k	1.9	34.1	7.0	42.9	73k	220k	3.7	17.4	0.8	21.9
5	13	179k	538k	3.7	44.6	10.2	58.5	73k	220k	3.5	17.4	0.8	21.6
6	14	210k	630k	4.4	54.0	13.9	72.3	73k	220k	8.3	17.3	0.8	26.4

Table 7.1: **Credit Loop Buffer Sizing Results using Circular Buffer Queue Style.** On each iteration i of the CEBUS loop, BSS uses a sequence of calls to BSS-SIZE in order to find the minimal sizing S that is correct for all patterns in set P, and BSV finds a counterexample traffic pattern that demonstrates $\mathscr{N}[S] \| \mathscr{T} \nvDash \phi$. The column headings are as follows: i is the iteration number for the CEBUS loop; $|S_i|$ is the minimum cumulative buffer size found by BSS in iteration i; Vars and Clauses represent the number of variables and CNF clauses in the corresponding SAT problem(s); SAT is the time spent in satisfiability solving; Enc is the time taken encoding the word-level input to CNF; SSim is the time taken by symbolic simulation; Total is the overall time spent in the respective iteration. Note that the problem sizes and runtimes for BSS are the average problem sizes over all the sizes tried in the binary search for finding the minimal size.

i	$\|S_i\|$	*Buffer-Size Synthesis (BSS)*						*Buffer-Size Verification (BSV)*					
		CNF Size		Runtime (sec)				CNF Size		Runtime (sec)			
		Vars	Clauses	SAT	Enc	SSim	Total	Vars	Clauses	SAT	Enc	SSim	Total
0	-	-	-	-	-	-	-	69k	206k	0.6	11.4	7.5	19.5
1	5	37k	113k	0.2	5.9	2.2	8.3	69k	206k	0.4	11.5	7.5	19.4
2	6	54k	161k	0.4	8.0	6.3	14.7	69k	206k	1.4	11.3	7.5	20.2
3	7	88k	265k	1.3	13.0	13.6	27.9	69k	206k	3.0	11.5	7.5	22.1
4	11	109k	328k	1.6	16.9	25.1	43.7	69k	206k	0.7	11.4	7.5	19.7
5	13	134k	404k	2.7	22.4	41.1	66.2	69k	206k	2.9	11.4	7.5	21.9
6	13	180k	540k	3.3	27.9	63.7	94.9	69k	206k	4.1	11.5	7.5	23.1
7	14	212k	636k	4.1	33.2	90.0	127.3	69k	206k	9.0	11.5	7.5	28.0

Table 7.2: **Credit Loop Buffer Sizing Results using Record-based Queue Style.** See caption of Table 7.1 for explanation of column headings.

Chip Multiprocessor Router

This second buffer sizing case study is a model of a chip-multiprocessor (CMP) router based on a design by Peh [81]. This CMP design is designed to be part of an on-chip communication fabric for connecting processors, memories, and other IP blocks to one another. Credited flow control is used for communication between each router and its neighbors. This experiment considers interactions between the three routers R_1, R_9 and R_{17}, as shown in Fig. 7.4. The parts of the three routers that are modeled are described here and shown in Fig. 7.4b. For each router, the 5-input, 5-output router core is modeled as described in detail in Sec. 4.3, but the unused ports are removed from the router core. The credit logic connecting the north output of R_1 to the south input of R_9 is modeled, as is the credit logic connecting the south output of R_{17} to the north input of R_9. The token return path in each credit loop has a 5-cycle non-pipelined delay added.

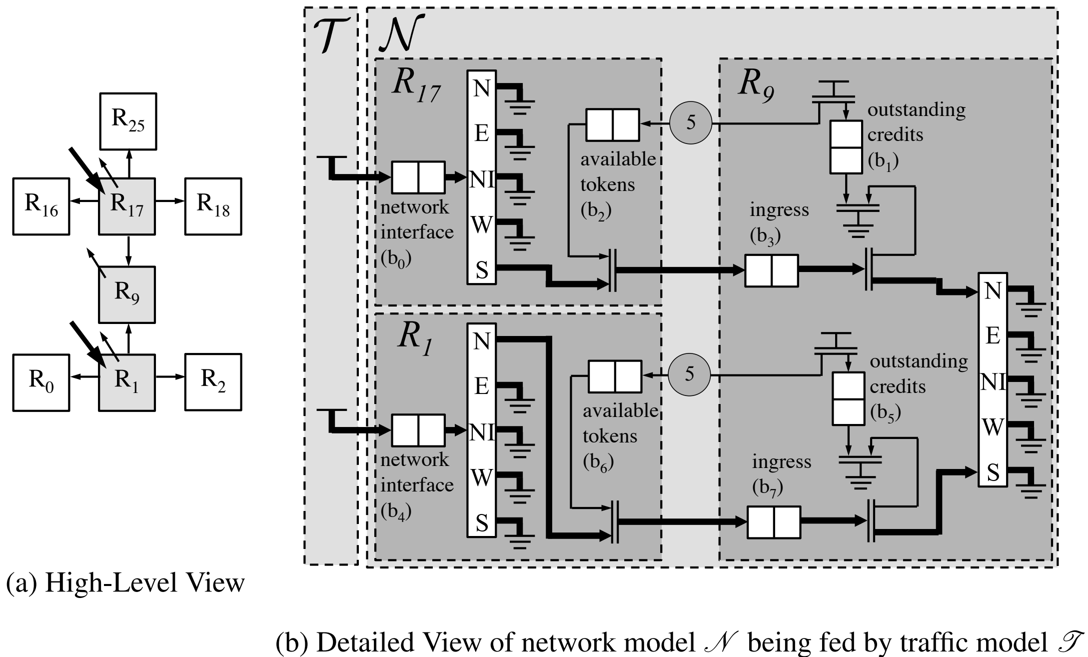
Experiments are performed using BMC with a depth of 22. As in the credit logic example, all flit queues are empty in the initial state of BMC, and token queues are filled. Property ϕ checks that the two channels from the traffic model are non-blocking, and checks a latency bound of 8 cycles.

Traffic model $\mathscr{T}$ injects flits into the network interface buffers of R_1 and R_{17}. The traffic model imposes that head flits injected into the network interface of R_1 obeys token bucket regulation with $(\sigma,\rho) = (1,7)$, and that head flits injected into network interface of R_{17} obey $(\sigma,\rho) = (1,9)$ regulation. An additional constraint enforces that no more than 3 packets (6 flits) are injected during any execution. Since the two sources can cause contention by routing packets to the same destination, the choice of destination addresses in the traffic model now matters. The destination addresses of head flits are generated from uninterpreted functions.

Discussion of BSV and BSS Runtimes

Table 7.3 and Table 7.4 show the problem sizes and runtimes for the CMP router buffer sizing experiments; Table 7.3 uses the circular-buffer queue implementation style, and Table 7.4 uses the record-based implementation style. Interesting contrasts are drawn between the BSS and BSV problems based on these results:

- The size of the BSS problem grows linearly with the number of patterns that are used in synthesis; this is observed in the roughly linear increase in the number of CNF clauses and variables, and the decision procedure encoding and symbolic simulation time. This is expected, as each new pattern requires an additional unrolling of the model transition relation for 22 cycles (the BMC depth) from the initial state. However, the large SAT problems that arise from BSS at later CEBUS iterations are not especially hard for the solver, and the SAT solving runtime is a minor contributor to the overall runtime.
- The size of the BSV problem is unchanged at each iteration of the CEBUS loop as the BSV problem is always just to disprove a single candidate buffer size. However, finding a counterexample is increasingly difficult as CEBUS progress. Very carefully crafted patterns



(a) High-Level View

(b) Detailed View of network model $\mathscr{N}$ being fed by traffic model $\mathscr{T}$

Figure 7.4: **CMP Network Model for Buffer Sizing Experiment.** Fig. 7.4a shows high level view of network model, which is basically a subset of larger mesh (e.g. as shown in Fig. 5.1). Traffic is injected on the network interface ports of R_{17} and R_1 as indicated by the bold arrows. Only routers R_1,R_9, and R_{17} are modeled in detail in $\mathscr{N}$. Fig. 7.4b shows detailed view of the three routers modeled in $\mathscr{N}$, and shows the eight buffers that are being sized in experiments. Traffic model $\mathscr{T}$ injects traffic into $\mathscr{N}$, and the two channels from $\mathscr{T}$ to $\mathscr{N}$ are specified as non-blocking.

are necessary to disprove a sizing at later iterations. This is seen in Tab. 7.5, where the final[1] BSV counterexample traffic pattern p_{10} has all traffic being sent to the same destination, and all traffic injected in a single burst. By contrast, the first pattern in Tab. 7.5 appears quite unexceptional, and was discovered quickly by the sat solver.

The CMP buffer sizing experiment also provides a good point of comparison for circular-buffer (Table 7.3) and record-based (Table 7.4) implementations of symbolically sized queues. The problem sizes using each implementation are similar in terms of the number of SAT variables and CNF clauses. However, the symbolic simulation runtimes are 1 to 2 orders of magnitude slower when using the record implementation, while the decision procedure encoding is marginally faster. Overall, the circular buffer implementation appears to be more scalable in this experiment.

[1] Note that BSV in the 11th and final CEBUS iteration does not produce a counterexample because the candidate sizing (S_{11}) is verified as correct, meaning that no pattern can be found to disprove it.

i	*Buffer-Size Synthesis (BSS)*							*Buffer-Size Verification (BSV)*					
	$\|S_i\|$	CNF Size		Runtime (sec)				CNF Size		Runtime (sec)			
		Vars	Clauses	SAT	Enc	SSim	Total	Vars	Clauses	SAT	Enc	SSim	Total
0	-	-	-	-	-	-	-	180k	540k	2.4	41.7	1.8	46.0
1	10	78k	235k	1.3	18.7	3.0	23.1	180k	540k	6.7	41.6	1.8	50.2
2	12	118k	354k	1.6	29.5	7.2	38.4	180k	540k	1.4	41.6	1.8	44.9
3	14	209k	626k	3.3	56.1	11.3	70.7	180k	540k	5.0	41.4	1.8	48.3
4	15	253k	760k	3.5	76.6	13.1	93.3	180k	540k	5.9	41.3	1.8	49.0
5	15	288k	865k	5.2	83.8	23.1	112.2	180k	540k	8.6	41.6	1.8	52.1
6	16	372k	1116k	7.8	108.6	30.8	147.3	180k	540k	9.1	41.6	1.8	52.6
7	18	434k	1303k	9.5	133.3	41.8	184.7	180k	540k	73.8	41.7	1.8	117.3
8	19	486k	1459k	14.4	152.3	50.8	217.6	180k	540k	43.8	41.8	1.8	87.4
9	20	525k	1577k	16.3	172.5	61.6	250.6	180k	540k	40.4	41.5	1.8	83.8
10	21	578k	1735k	13.5	209.5	65.7	288.8	180k	540k	16.9	41.6	1.8	60.3
11	21	646k	1937k	20.1	203.6	88.6	312.4	180k	540k	113.4	41.6	1.8	156.8

Table 7.3: **CMP Buffer Sizing Results using Circular Buffer Queue Style.** See caption of Table 7.1 for explanation of column headings.

Discussion of BSS and BSV Counterexamples

In the CEBUS experiments using circular-buffer implementations of queues (Tab. 7.3) applied to the CMP router, each call to BSS or BSV returns a counterexample until a termination condition is reached. The counterexamples in BSV are traffic patterns that cause a property to fail, and the counterexamples in BSS are buffer sizes that prevent properties from failing.

Each failed verification attempt in BSV returns as a counterexample a traffic pattern for which a candidate buffer size is inadequate. For the ith iteration of the CEBUS loop, this counterexample pattern is denoted p_i. The counterexamples for the first 10 candidate sizes are shown in Table 7.5; the 11th candidate size (S_{11}) is successfully verified, as no pattern can be generated by traffic model $\mathscr{T}$ to disprove it. As more iterations of the CEBUS loop are completed, the traffic patterns become increasingly adversarial. The final counterexample traffic pattern, p_{10} in the 10th iteration, overwhelms the network by sending the global maximum of 3 packets in a single burst and all having the same destination. The packets are sent starting in the 7th cycle and not at the start of the pattern, because the traffic model for the source of R_1 has a token bucket that is replenished by a periodic token source in the 8th cycle (See Chapter 4.5). By starting the burst in the 7th cycle, the token bucket is reloaded while the tail flit of the first packet is injected, allowing the head flit of a second packet to follow immediately.

The minimal buffer sizing (S_i) at each iteration i of the CEBUS loop is given in Tab. 7.6. A significant fact to note is that individual buffer sizes do not monotonically increase over the iterations. This is allowed because buffer size assignments are not retained from one iteration to the next. Instead, only the total buffer sizing ($|S_i|$) is retained, and is used as a lower bound for BSS in future

i	Buffer-Size Synthesis (BSS)							Buffer-Size Verification (BSV)					
	$\|S_i\|$	CNF Size		Runtime (sec)				CNF Size		Runtime (sec)			
		Vars	Clauses	SAT	Enc	SSim	Total	Vars	Clauses	SAT	Enc	SSim	Total
0	-	-	-	-	-	-	-	173k	520k	2.4	29.1	65.2	96.9
1	10	54k	164k	1.1	8.4	10.8	20.4	173k	520k	2.0	29.3	65.2	96.6
2	12	79k	237k	2.2	12.8	43.6	58.7	173k	520k	2.4	29.1	65.3	96.9
3	14	149k	449k	5.3	24.4	90.5	120.3	173k	520k	7.1	29.3	65.4	101.9
4	15	183k	549k	6.5	28.7	159.4	194.7	173k	520k	7.4	28.9	65.0	101.3
5	15	235k	706k	9.8	39.4	264.5	313.7	173k	520k	22.7	29.6	65.3	117.7
6	16	278k	834k	10.2	49.0	398.2	457.4	173k	520k	78.4	29.5	65.3	173.2
7	17	334k	1002k	15.1	60.3	541.9	617.4	173k	520k	52.8	29.3	65.4	147.6
8	19	371k	1114k	17.7	65.3	805.3	888.4	173k	520k	135.1	29.1	65.2	229.5
9	19	427k	1282k	20.4	75.1	1067.1	1162.8	173k	520k	161.3	29.4	65.4	256.2
10	21	468k	1424k	23.1	84.5	1301.9	1409.5	173k	520k	230.7	29.8	65.2	325.8
11	21	503k	1593k	27.8	89.3	1567.4	1684.6	173k	520k	393.4	29.6	65.2	488.2

Table 7.4: **CMP Buffer Sizing Results using Record-based Queue Style.** See caption of Table 7.1 for explanation of column headings.

iterations.

7.5 Related Work

The problem of buffer minimization has been widely studied in the digital signal processing (DSP) community. Synchronous dataflow (SDF) models [64] in particular are used to reason about the minimum buffer size required for a feasible schedule to exist in order for the model to be deadlock-free and to conform to timing constraints [98]. In general, this buffer minimization problem is NP-complete [5].

Various techniques have been applied to address the NP-completeness. Poplavko *et al.* [85] use an SDF model of an NoC to perform timing analysis and for rate-optimal buffer sizing. Geilen *et al.* [46] use model checking to determine whether there exists a buffer sizing smaller than some bound that admits a deadlock-free schedule; the minimal sizing is found using iterated calls to the model checker. Stuijk *et al.* [98] present a dynamic programming algorithm that generates a set of candidate buffer sizings that is guaranteed to contain all Pareto-optimal points in the buffer-size versus throughput space; the Pareto-optimal points among the set are then found by self-timed simulation. Wiggers *et al.* [106] approximate minimal buffer sizing for a given throughput using a network-flow formulation, but the closeness of approximation is not bounded.

SDF can model only a limited class of NoCs. Assumptions of periodic sources and data-independent routing make SDFs well-suited to modeling multimedia NoCs, but not for general-purpose chip multiprocessor (CMP) NoCs. In a CMP, the injected traffic at each node can vary in burst size, have irregular periods, and choose destinations non-uniformly over time. Additionally, due to

traffic pattern	node	Cycle																					
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p_0	1									H_1	T							H_1	T				
	17								H_9	T													
p_1	1								H_8	T													
	17																			H_0	T	H_0	
p_2	1														H_1	T			H_0	T			
	17																						
p_3	1			H_9	T																		
	17			H_9	T							H_9	T										
p_4	1																						
	17							H_9	T			H_9	T										
p_5	1															H_{16}	T	H_{16}					
	17					H_0	T																
p_6	1					H_9	T			H_9	T												
	17															H_0	T						
p_7	1					H_9	T			H_9	T												
	17										H_9	T											
p_8	1												H_9	T									
	17								H_{17}	T		H_9	T										
p_9	1							H_1	T		H_9	T											
	17										H_9	T											
p_{10}	1							H_9	T	H_9	T												
	17							H_9	T														

Table 7.5: **Counterexample Traffic Patterns from BSV in CEBUS on CMP Design.** On each iteration i, BSV disproves a candidate size by generating from the traffic model $\mathscr{T}$ a pattern p_i that causes a property to fail. Some of the traffic patterns are shorter than the 22 cycle BMC depth; in these patterns a property fails before the 22nd cycle of BMC. The subscript of each head flit is its destination address. These patterns are produced in experiments using the circular buffer queue implementation, corresponding to the results shown in Tab. 7.3

the lack of support for conditionals, SDFs are not expressive enough to model NoC designs with detailed routing and arbitrary logic.

Thus, analysis of general-purpose CMPs is typically based on simulation or probabilistic reasoning. Using a trace-driven approach, Kahng *et al.* [55] show that sizing buffers without considering burstiness can cause significant error in maximum latency prediction. Using network analysis, injected traffic with bounded burstiness leads to bounds on required buffer sizes [31][32]. Stochastic automata networks (SAN) [83] have also been used to model network traffic in SoCs [68]. While SANs allow for efficient reasoning about average case results, they are not suitable for worst-case analysis. Addressing limitations in the probabilistic analysis of stochastic models, adversarial queuing theory has been proposed [11]. If traffic injection is modeled as a Poisson distribution,

i	Buffer size assignment S								$\|S\|$
	$S(b_0)$	$S(b_1)$	$S(b_2)$	$S(b_3)$	$S(b_4)$	$S(b_5)$	$S(b_6)$	$S(b_7)$	
0	2	1	1	1	2	1	1	1	10
1	1	1	1	1	2	2	3	1	12
2	2	1	2	1	2	2	3	1	14
3	2	1	2	1	3	2	3	1	15
4	2	1	2	1	2	2	3	2	15
5	2	1	2	1	2	2	4	2	16
6	3	1	2	1	2	2	4	2	17
7	2	2	3	2	3	2	4	1	19
8	3	2	3	1	2	2	4	2	19
9	2	3	4	2	3	2	4	1	21
10	2	3	4	2	2	2	4	2	21

Table 7.6: **Buffer Sizes Produced by BSS in CEBUS for CMP Design.** On each iteration i, BSS produces a size that causes property to be satisfied. Each buffer b_i corresponds to the ones shown in Fig. 7.4b. These results are from the CMP experiment using the circular buffer queue implementation, corresponding to the results shown in Tab. 7.3.

queuing theory provides a closed form solution to find the buffers most likely to be full [53]. For the same total buffer budget, increasing the size of these oft-used buffers has a better impact on latency than uniformly upsizing [53].

The approach of this chapter attempts to find some middle ground between limitations of SDF and the lack of guarantees from simulation-based approach. In particular, the following features differentiate this chapter's approach from the rest:

- Reasoning about NoC models occurs at the micro-architectural level. This means that instead of characterizing the network with routing probability at each node, the routing logic of each router is modeled exactly, with sufficient details of its control flow. This level of abstraction is similar to the Boolean Data Flow (BDF) model, while buffer minimization is more commonly investigated for the less expressive SDF model.

- An SMT-based approach for determining the minimum buffer capacity for a NoC that satisfies the performance properties.

The counterexample-guided approach has been used before, for computing abstract models [28] and for program synthesis [95]. The work described in this chapter is the first to adapt this methodology for synthesizing buffer sizes in NoC designs.

7.6 Conclusion

This chapter presents a novel approach for minimizing the cumulative buffer size in on-chip networks, so as to meet throughput and latency requirements, given high-level specifications on traffic behavior. The approach uses model checking based on satisfiability modulo theories (SMT) solvers, within an overall counterexample-guided synthesis loop. Experimental results on models of NoC components show the promise of the proposed technique. While this chapter has focused on using quantifier instantiation to resolve quantifier alternation, an alternative approach to consider would be using the same modeling technique with a QBF solver to address the alternating quantifiers in a single monolithic solver call.

The ability to a give a formal guarantee of QoS for a buffer is a unique contribution compared to prior works, yet the approach may not scale up to problems that contain more than a few tens of queues, or problems with large latency bounds that can only be checked with a deep BMC. In such a problem, not only does the size of the problem grow, but the number of iterations of the CEBUS loop is also likely to grow.

Chapter 8

Conclusions and Future Work

This dissertation presents work toward formal verification and synthesis of on-chip networks for satisfying performance properties. An overarching theme that ties the chapters together is verifying latency bound properties with model checking. The two applications presented for model checking latency properties are latency verification, and synthesis of optimal buffer sizes while ensuring that latency bounds are satisfied.

The main challenges that are addressed throughout this dissertation are the creation of appropriate models, and techniques for scalable model checking. Both challenges arise from the scale of the problems of interest, where a model may contain many thousands of state variables, and the latency properties being checked can be hundreds or even thousands of cycles. Techniques are given for using simple sequential models that are expressive enough for model checking of latency properties, while significantly less complex than the arbitrary RTL that typically describes NoCs. Domain-specific techniques for scalable model checking are also given, based on both SAT solving and SMT solving.

Chapter 4 presents the modeling approach, based on the xMAS formalism developed by Chatterjee *et al.* [22]. Network models in xMAS resemble a simplified RTL with unnecessary details hidden, and are created as compositions of a set of simple xMAS primitives. The xMAS models in this dissertation are modified to be amenable to latency verification. Furthermore, it is shown that xMAS can be used to describe traffic models in addition to network models. The models in this dissertation are created by hand, and a useful direction for future work is to explore automatic generation of xMAS models from arbitrary RTL. If this is not possible, then it may be useful to formally check latency equivalence of the original RTL and its hand-crafted xMAS representation.

Chapter 5 presents the first of two approaches for compositional latency verification. This approach reduces both the number of variables, and the number of unrollings of the model transition relation, by decomposing the model and the latency property along router boundaries. To allow the overall network model to be decomposed into individual routers, inferred traffic models serve as interface specifications between the routers. A promising direction for future work is using the inferred traffic models for diagnosing the root causes of large latencies on new benchmarks. Whenever

the latency induced by a new benchmark exceeds the latency proved using models inferred from previous benchmarks, then the new benchmark must cause one or more of the traffic models to be violated, and the model that is violated indicates the root cause of the high latency.

Chapter 6 presents the second of two approaches for compositional latency verification. This approach keeps the model and property whole, but strengthens the property using latency lemmas. The latency lemmas allow the property to be proved inductively, reducing the number of unrollings required, and achieving scalability. This approach produces very promising results, and is able to verify a 79 cycle bound on an 8-agent ring interconnection network with more than two orders of magnitude speedup over alternative approaches. As the generation of latency lemmas is not automated for all possible xMAS networks, future work building upon this chapter could consider further automating the generation of latency lemmas.

Chapter 7 presents an approach for optimal sizing of buffers while ensuring that performance properties are met. A counterexample-guided approach is used to iteratively generate and then verify or disprove candidate buffer sizes.

The challenging model checking problems generated in this dissertation are being used as solver benchmark problems. The SMT problems from Chapter 5 were submitted in SMT-LIB format for inclusion in SMT-COMP [1]. The aiger files from Chapter 6 were submitted for inclusion in the hardware model checking competition [2]. The UCLID problems from Chapter 7 are being used internally within Sanjit Seshia's research group at UC Berkeley.

[1] http://www.smtcomp.org

[2] fmv.jku.at/hwmcc/

Bibliography

[1] UCLID Verification System. Available at `http://uclid.eecs.berkeley.edu`.

[2] N Adiga, M Blumrich, D Chen, P Coteus, A Gara, ME Giampapa, P Heidelberger, S Singh, BD Steinmacher-Burow, T Takken, M Tsao, and P Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49, March 2005.

[3] V S Adve and Mary Vernon. Performance analysis of mesh interconnection networks with deterministic routing. *Parallel and Distributed Systems*, January 1994.

[4] C Barrett, R Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of Satisfiability*, 2009.

[5] SS Bhattacharyya, PK Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*. 1996.

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

[7] A Biere, A Cimatti, and E Clarke. Symbolic model checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[8] Per Bjesse and Koen Claessen. SAT-based verification without state space traversal. *Formal Methods in Computer-Aided Design*, 2000.

[9] Paul Bogdan and Radu Marculescu. Statistical physics approaches for network-on-chip traffic characterization. In *Proceedings*, pages 461–470. New York, NY, USA, 2009.

[10] E Bolotin, I Cidon, and R Ginosar. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 2004.

[11] Allan Borodin, Jon Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P Williamson. Adversarial queuing theory. *J. ACM*, 48(1):13–38, 2001.

[12] Aaron R Bradley. SAT-Based Model Checking Without Unrolling. *Verification, Model Checking, and Abstract Interpretation*, 2011.

[13] Bryan A Brady, R Bryant, Sanjit A Seshia, and John W O'Leary. ATLAS: Automatic Term-level abstraction of RTL designs. *ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 2010.

[14] Bryan A Brady, Daniel E Holcomb, and Sanjit A Seshia. Counterexample-Guided SMT-Driven Optimal Buffer Sizing. *Design Automation and Test in Europe*, 2011.

[15] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. *Computer Aided Verification*, 2010.

[16] R Brummayer and A Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. *Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[17] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, (8):677–691, 1986.

[18] Randal E Bryant, Shuvendu K Lahiri, and Sanjit A Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. *Computer Aided Verification*, pages 78–92, July 2002.

[19] J R Burch, E M Clarke, K L McMillan, D L Dill, and L J Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of Fifth Annual IEEE Symposium on Logic in Computer Science, 1990 (LICS '90)*, pages 428–439, 1990.

[20] Luca P Carloni, Kenneth L McMillan, and Alberto L Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, September 2001.

[21] Satrajit Chatterjee and M Kishinevsky. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics. In *Computer Aided Verification*, 2010.

[22] Satrajit Chatterjee, M Kishinevsky, and Umit Y Ogras. Quick Formal Modeling of Communication Fabrics to Enable Verification. In *High Level Design Validation and Test Workshop (HLDVT)*. HLDVT, 2010.

[23] Satrajit Chatterjee and Michael Kishinevsky. Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics. *Formal Methods in System Design*, 40(2):147–169, 2012.

[24] Satrajit Chatterjee, Michael Kishinevsky, and Umit Y Ogras. xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification. *IEEE Design & Test of Computers*, 29(3):80–88, 2012.

[25] 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*, 51(5):559–572, 2007.

[26] M Christodorescu, S Jha, S A Seshia, Dawn Song, and Randal E Bryant. Semantics-Aware Malware Detection. *IEEE Symposium on Security and Privacy*, 2005.

[27] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 85–96, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[28] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *12th International Conference on Computer Aided Verification (CAV*.

[29] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[30] Stephen A Cook. The complexity of theorem-proving procedures. In *the third annual ACM symposium*, pages 151–158, New York, New York, USA, 1971. ACM Press.

[31] Rene L Cruz. A calculus for network delay, part I. Network elements in isolation. *IEEE Transactions on Information theory*, 37(1):114–131, 1991.

[32] Rene L Cruz. A calculus for network delay, part II: Network analysis. *IEEE Transactions on Information theory*, 37(1):132–141, 1991.

[33] Matteo Dall'Osso, Gianluca Biccari, Luca Giovannini, Davide Bertozzi, and Luca Benini. Xpipes: a latency insensitive parameterized network-on-chip architecture for multiprocessor SoCs. In *Proceedings of the 21st International Conference on Computer Design (ICCD'03)*, pages 536–539, 2003.

[34] W J Dally. Virtual-channel flow control. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):194–205, 1992.

[35] W J Dally and C L Seitz. The Torus Routing Chip. *CALIFORNIA INST OF TECH PASADENA DEPT OF COMPUTER SCIENCE*, 1986.

[36] W J Dally and C L Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *Computers, IEEE Transactions on*, (5):547–553, 1987.

[37] William J Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. *Design Automation Conference*, 2001.

[38] William J Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[39] M Davis, G Logemann, and D Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.

[40] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[41] J Dielissen, Andrei Radulescu, Kees Goossens, and E Rijpkema. Concepts and implementation of the Philips network-on-chip. In *Proceedings*, pages 1–6. Citeseer, 2003.

[42] Niklas Een and Alan Mishchenko. Efficient implementation of property directed reachability. In *Proceedings of IWLS*, 2011.

[43] Niklas Een and Niklas Sörensson. An extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, 2004.

[44] C Fallin, X Yu, G Nazario, and O Mutlu. A high-performance hierarchical ring on-chip interconnect with low-cost routers. *SAFARI Technical Report No. 2011-007, Computer Architecture Lab, Carnegie Melon University*, 2011.

[45] Vinod Ganapathy, Sanjit A Seshia, Somesh Jha, Thomas W Reps, and Randal E Bryant. Automatic discovery of API-level exploits. In *the 27th international conference*, page 312, New York, New York, USA, 2005. ACM Press.

[46] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd annual Design Automation Conference*, pages 819–824. New York, NY, USA, 2005.

[47] Kees Goossens, J Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414–421, September 2005.

[48] Alexander Gotmanov, Satrajit Chatterjee, and Michael Kishinevsky. Verifying Deadlock-Freedom of Communication Fabrics. *Verification, Model Checking, and Abstract Interpretation*, 2011.

[49] Pierre Guerrier and Alain Grenier. A generic architecture for on-chip packet-switched interconnections. *Proceedings of the conference on Design Automation and Test in Europe*, 2000.

[50] Daniel E Holcomb, Bryan A Brady, and Sanjit A Seshia. Abstraction-Based Performance Analysis of NoCs. *Design Automation Conference*, June 2011.

[51] Daniel E Holcomb, Alexander Gotmanov, Michael Kishinevsky, and Sanjit A Seshia. Compositional Performance Verification of NoC Designs. In *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, July 2012.

[52] Y Hoskote, S Vangal, A Singh, N Borkar, and S Borkar. A 5-GHz Mesh Interconnect for a Teraflops Processor. *Micro, IEEE*, 27(5):51–61, 2007.

[53] Jingcao Hu and Radu Marculescu. Application-specific buffer space allocation for networks-on-chip router design. In *International Conference on Computer Aided Design*, pages 354–361. Washington, DC, USA, 2004.

[54] Susmit Jha, Bryan A Brady, and Sanjit A Seshia. Symbolic Reachability Analysis of Lazy Linear Hybrid Automata. In *Formal Modeling and Analysis of Timed Systems*, pages 241–256. Berlin, Heidelberg, 2007.

[55] Andrew B Kahng, Bill Lin, K Samadi, and Rohit Sunkam Ramanujam. Trace-driven optimization of networks-on-chip configurations. *Design Automation Conference*, 2010.

[56] Richard M Karp. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, pages 85–103. 1972.

[57] B A Krishna, J Michelson, V Singhal, and A Jain. Liveness vs Safety–A Practical Viewpoint. *7th international Haifa Verification conference on Hardware and Software*, 2011.

[58] S Kumar, A Jantsch, JP Soininen, M Forsell, M Millberg, J Oberg, K Tiensyrja, and A Hemani. A network on chip architecture and design methodology. In *IEEE Symposium on VLSI*, pages 117–124, 2002.

[59] Orna Kupferman, Nir Piterman, and Moshe Y Vardi. From liveness to promptness. *Formal Methods in System Design 2009*, January 2009.

[60] K Lahiri, A Raghunathan, and S Dey. Evaluation of the traffic-performance characteristics of system-on-chip communication architectures. In *14th International Conference on VLSI Design*, pages 29–35. IEEE Comput. Soc, 2001.

[61] Shuvendu K Lahiri and Randal E Bryant. Deductive Verification of Advanced Out-of-Order Microprocessors. In *Computer Aided Verification*, pages 341–354. 2003.

[62] Shuvendu K Lahiri and Sanjit A Seshia. The UCLID Decision Procedure. *Computer Aided Verification*, 2004.

[63] Shuvendu K Lahiri, Sanjit A Seshia, and Randal E Bryant. Modeling and Verification of Out-of-Order Microprocessors in UCLID. *Formal Methods in Computer-Aided Design*, 2002.

[64] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[65] Hyung Gyu Lee, Naehyuck Chang, Umit Y Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. *Transactions on Design Automation of Electronic Systems (TODAES*, 12(3), August 2007.

[66] Jian Liang, Sriram Swaminathan, and Russell Tessier. ASOC: a scalable, single-chip communications architecture. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 37–46, 2000.

[67] Jiang Long, Sayak Ray, Baruch Sterin, Alan Mishchenko, and Robert K Brayton. Enhancing ABC for LTL Stabilization Verification of SystemVerilog/VHDL Models. *International Workshop on Design and Implementation of Formal Tools and Systems*, January 2011.

[68] Radu Marculescu, Umit Y Ogras, and Nicholas H Zamora. Computation and communication refinement for multiprocessor SoC design: A system-level perspective. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, page 592. ACM Request Permissions, July 2006.

[69] Joao Marques-Silva and Karem A Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[70] Matthew Mattina, George Z Chrysos, and Stephen Felix. Method and apparatus for synchronous unbuffered flow control of packets on a ring interconnect. *US Patent 7,539,141*, May 2009.

[71] K L McMillan. Interpolation and SAT-based model checking. *Computer Aided Verification*, 2003.

[72] M Millberg, E Nilsson, R Thid, and A Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Design Automation and Test in Europe*. IEEE Computer Society Washington, DC, USA, 2004.

[73] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, 2001.

[74] L Ni and P McKinley. A survey of wormhole routing techniques in direct networks. *COMPUTER,*, January 1993.

[75] R Nieuwenhuis and A Oliveras. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM*, 2006.

[76] Umit Y Ogras, Paul Bogdan, and Radu Marculescu. An analytical approach for network on chip performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2010.

[77] Umit Y. Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in NoC design: a holistic perspective. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74, New York, NY, USA, 2005. ACM.

[78] Umit Y Ogras and Radu Marculescu. *Modeling, Analysis and Optimization of Network-on-Chip Communication Architectures*. Springer, March 2013.

[79] P Pande, C Grecu, and M Jones. Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Transactions on Computers*, 2005.

[80] PARSEC Benchmark Suite. Available at http://parsec.cs.princeton.edu/. Technical report.

[81] Li-Shiuan Peh. Flow control and micro-architectural mechanisms for extending the performance of interconnection networks. *portal.acm.org*, 2001.

[82] Alessandro Pinto, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. Constraint-driven communication synthesis. In *Design Automation Conference*, page 788, 2002.

[83] Brigitte Plateau and Karim Atif. Stochastic Automata Network of Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[84] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[85] P Poplavko, Twan Basten, Marco J G Bekooij, J van Meerbergen, and B Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2003*, pages 63–72. New York, NY, USA, 2003.

[86] M Prasad, A Biere, and A Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 2005.

[87] Yue Qian, Zhonghai Lu, and Wenhua Dou. Applying network calculus for performance analysis of self-similar traffic in on-chip networks. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '09*, pages 453–460. New York, NY, USA, 2009.

[88] G Ravindran and M Stumm. A performance comparison of hierarchical ring- and mesh-connected multiprocessor networks. *Third Annual Symposium on High-Performance Computer Architecture*, pages 58–69, 1997.

[89] S Ray and R K Brayton. Scalable progress verification in credit-based flow-control systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 905–910, 2012.

[90] Sayak Ray and Robert K Brayton. Well-foundedness in Credit-Based Flow-Control Systems. *International Workshop on Logic Synthesis*, pages 1–8, March 2012.

[91] G Rizzo and J LeBoudec. “Pay bursts only once” does not hold for non-FIFO Guaranteed Rate nodes. *Performance Evaluation*, 2005.

[92] 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. *Micro, IEEE*, 29(1):10–21, 2009.

[93] Sanjit A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, Carnegie Mellon University, 2005.

[94] M Sheeran, S Singh, and Gunnar Stalmark. Checking safety properties using induction and a SAT-solver. *Formal Methods in Computer-Aided Design*, 2000.

[95] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM Press, 2006.

[96] V Soteriou, Hangsheng Wang, and L Peh. A statistical traffic model for on-chip interconnection networks. In *Conference*, pages 104–116, 2006.

[97] Dimitrios Stiliadis and Anujan Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking (ToN)*, 1998.

[98] S Stuijk, Marc Geilen, and Twan Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, October 2008.

[99] Alan M Turing. Checking a large routine. *Conference on High Speed Automatic Calculating Machines*, 1949.

[100] Girish Varatkar and Radu Marculescu. Traffic analysis for on-chip networks design of multimedia applications. In *Proceedings*, pages 795–800. New York, NY, USA, 2002.

[101] F Verbeek and J Schmaltz. Formal specification of networks-on-chips: deadlock and evacuation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 1701–1706, 2010.

[102] Freek Verbeek and Julien Schmaltz. Hunting deadlocks efficiently in microarchitectural models of communication fabrics. *Formal Methods in Computer-Aided Design*, 2011.

[103] Freek Verbeek and Julien Schmaltz. Easy Formal Specification and Validation of Unbounded Networks-on-Chips Architectures. *Transactions on Design Automation of Electronic Systems*, 17(1):1–28, January 2012.

[104] Yuriy Viktorov and Alexander Gotmanov. Latency Analysis in Microarchitectural Models of Communication Fabrics. *Problems of Advanced Micro- and Nanoelectronic Systems Development (MES)*, pages 67–72, 2012.

[105] David Wentzlaff, Patrick Griffin, Henry Hoffmann, L Bao, B Edwards, C Ramey, M Mattina, C Miao, John Brown, and Anant Agrawal. On-chip interconnection architecture of the tile processor. *Micro*, 2007.

[106] Maarten H Wiggers, Marco J G Bekooij, and Gerard J M Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 658–663. New York, NY, USA, 2007.

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