Network on a Chip:
Modeling Wireless Networks with Asynchronous VLSI

Rajit Manohar and Clinton Kelly, IV
Computer Systems Laboratory
School of Electrical and Computer Engineering
Cornell University, Ithaca NY 14853

Abstract

We introduce the notion of a network-on-a-chip: a programmable, asynchronous VLSI architecture for fast and efficient simulation of wireless networks. The approach is inspired by the remarkable similarity between networks and asynchronous VLSI. Our approach results in simulators that can evaluate network scenarios much faster than real-time, enabling a new class of network protocols that can dynamically change their behavior based on feedback from in-situ simulation. We describe our simulation architecture, and present results that validate our approach.

1 Introduction

The complexities of modeling the behavior of networks has resulted in the widespread use of network simulation as a technique to evaluate network protocols and architectures. Network simulators like OPNET [4] and ns-2 [10] aid designers in determining whether a particular protocol change would improve the overall behavior of the network. Research in the area of network simulation has been focused on developing fast software simulation techniques for Internet-scale networks (for instance, see [2, 3, 5]), as well as the design of flexible and compositional component-based simulation architectures (for instance, see [3, 14]).

Wireless ad hoc networks present a somewhat different point in the design space for network simulators. A wireless channel is more complex than a traditional wired transmission channel. For instance, messages sent by different entities to distinct destinations can interfere with one another, something that is not a typical occurrence in a switched wired network. Environmental factors such as variation in terrain can affect the available bandwidth in a wireless network. Wireless network simulators like OPNET include modules that model these additional properties of wireless networks to improve the accuracy of network simulation.

An ad hoc network by its very nature is a highly dynamic environment. The mobility of endpoints causes variations in the quality of communication links. In addition, different applications using a wireless network might have different quality of service requirements. Mobility also results in a changing network topology, which could, for instance, affect the performance of multi-hop routing protocols. To combat these two issues, a wireless endpoint can be designed so that information about quality of service parameters as well as network topology changes can be incorporated into the protocol used to transmit data in the network. The net result would be a protocol that adapts to changing network conditions.

Ideally, we would like to evaluate the effect of modifying the transmission protocol before using the modified protocol. Knowing the effect of the change could, for instance, prevent us from wasting power at a resource-constrained endpoint. To design a protocol that can evaluate the effect of a change in network conditions and use this information for adaptation, we need the capability to simulate quickly the result of deploying a new protocol.

In this paper we introduce the notion of a *network on a chip* (NoC): a programmable, asynchronous VLSI architecture for efficient and fast simulation of wireless networks. The approach is inspired by the remarkable similarity between networks and asynchronous VLSI. Our approach is focused on the design of VLSI simulators that can evaluate certain classes of wireless network scenarios much faster than real-time, enabling a new class of network monitoring and configuration algorithms that can dynamically change network behavior based on feedback from in-situ simulation. Section 2 discusses existing approaches to fast network simulation, and describes asynchronous VLSI design and its relation to networks. Section 3 discusses some of the issues in adopting our approach, including design challenges and potential limitations. Section 4 contains the results of our evaluation that validates our approach, and Section 5 presents concluding remarks.

2 Background

2.1 Network Simulation

A network consists of a number of nodes that can communicate with one another by exchanging messages. In a wireline network, a node is connected to a collection of neighboring nodes by physical communication links. In a wireless network, a node simply broadcasts a message and some subset of nodes that are physically proximate to the transmitting node will receive the message. In general, the connectivity graph that describes the nodes and communication links in a wireless network is highly dynamic. Given a description of the nodes in the network and the topology, a network simulator attempts to simulate the behavior of the network.

The core of a network simulator is a discrete-event simulation engine. Each entity in the network being simulated can generate *events*, and the events themselves trigger other events. Each event occurs at some point in time, known as its *timestamp*, and the events are stored in an event queue that is ordered in time. The simulation proceeds by selecting the first event, and executing it. This event will in turn add additional events to the event queue. If the simulator has accurate time estimates for each event, and every physical event is modeled correctly, the simulator will correctly model the behavior of the complete network.

The drawback to using such a discrete-event simulation approach is that each event in the network is executed sequentially, whereas a real network is a highly concurrent system. Therefore, simulating an hour of activity (i.e., an hour of real time) in a complex network can take many hours, if not days, of simulation time using a direct discrete-event based approach. This problem is exacerbated when the number of active nodes in the network increases, because this increases the number of events that the network simulator must execute in a given interval of real time.

There are several approaches that are commonly adopted by researchers to tackle the problem of simulation speed:

- The use of parallelism in the simulator itself;
- The use of abstraction and analytic techniques to ignore the details of the simulation that are deemed unimportant for the modeling problem at hand.

Parallel Discrete-Event Simulation. This approach attempts to run the discrete-event simulation engine on a multiprocessor system. However, there is one constraint that any discrete-event simulator must maintain—namely, that the simulator must produce a result that is the same as the result obtained when each simulation event is executed in the order specified by its timestamp. This constraint is easy to achieve when the simulator itself runs on one processor because all the events are stored in one global event queue that is ordered in time. If we attempt to distribute this event queue among multiple, independently operating processors, then a processor not only must handle events in its own event queue, but it might receive events from other processors. The key problem here is that an event received from a remote processor may arrive too late, thereby violating our simulation requirement. This is sometimes referred to as the *event ordering problem*.

Current parallel simulation environments are based on either conservative or optimistic simulation techniques [1, 2, 6]. Conservative techniques stall local execution until the processor is certain that any event received from a remote processor will not arrive late. Optimistic techniques continue local execution, but provide a roll-back capability in case an event from a remote processor arrives late. Simulation environments such as Parsec [2] provide a way for the user to select the underlying algorithm used by the discrete-event simulator. However, the additional synchronization required among processors introduces overhead in the parallel simulation that limits speedup [2].

Abstraction and Analytic Techniques. Abstraction is a technique whereby certain details of the underlying network are ignored in the interests of improving the performance of the simulation [5]. Since abstracting away details might change the results of the simulation, care must be taken to identify the parts of the simulation that can be safely ignored. This approach can also be combined with analytic techniques [12], where multiple steps of the simulation are mathematically analyzed and the results of the analysis are used to reduce the number of events required by the discrete-event simulator. Another recently proposed analytic technique treats network flows as fluids, and models them with stochastic differential equations [9]. The analysis presented is specific to transmission control protocol (TCP) flows in Internet routers, with the goal being to understand certain observed properties of congestion control algorithms.

As an alternative to adopting a discrete-event simulation approach, researchers have begun exploring the notion of using a physical network testbed to simulate various network scenarios. Network entities in the physical network, such as end-points and routers, are configured to match the scenario being simulated. For example, Rizzo's dummynet extensions to the FreeBSD operating system [11] allow packets to be selectively inserted into queues that have user-specified bandwidth and latency limits. These queues can be inserted into the packet processing path in the kernel network stack, thereby creating the illusion of a communication link with the appropriate properties. Even using this approach, simulation speed is limited by the actual packet processing times in the system. Also, other processes executing on the system can cause the network simulation to schedule events at incorrect times, thereby causing inaccuracies in simulation [11].

2.2 Asynchronous VLSI

Traditional circuit design uses clock signals to implement sequencing. If two operations have to be sequenced, then the clock signal is used to indicate when one operation is complete and the next operation can begin. The clock determines when the circuit advances from one step to the next, and all parts of the circuit must have completed their local computations in the time budget provided by a clock cycle. It is this functionality—namely, ordering operations that have to be executed one after the other—that is important, not the use of the clock signal itself.

In contrast, one can design circuits where sequencing is not governed by a global clock signal. Such circuits are said to be *asynchronous*. Asynchronous design is an appealing methodology for large-scale systems. The absence of a global clock means that local signals have to be used to determine when the output of a function block is ready. Large asynchronous circuits typically use delay-insensitive codes to represent data, although other implementations are also possible.

Asynchronous circuits are designed by describing the VLSI computation using a programming notation. This programming notation is called Communicating Hardware Processes (CHP). In the notation, an asynchronous VLSI computation is described as a collection of sequential *processes* that communicate with one another by exchanging messages. There is a well-established formal, synthesis-based approach to the design of asynchronous circuits [8]. This approach has been successfully applied to the design of asynchronous microprocessors, as well as other custom asynchronous VLSI implementations [7].

The absence of a global clock also implies that the entire computation is event-driven. When inputs arrive to a circuit implementing a function, the circuit reads the inputs and produces the appropriate output. This output can then be used as an input to another function. The key observation here is that circuits in an asynchronous VLSI implementation are idle until they receive input data. The arrival of an input is an event that triggers the circuit, activates it, and makes it produce an output that in turn triggers other events in the asynchronous computation. Logically, we can think of an asynchronous computation as a highly concurrent event-driven network of communicating components. It is this event-driven nature of asynchronous circuits that makes them resemble networks.

2.3 Asynchronous VLSI and Networks

We use the simple example of a first-in first-out buffer (FIFO) to illustrate how an asynchronous VLSI circuit operates, and use this to discuss several features that asynchronous circuits and networks have in common.

A FIFO that can hold upto three one-bit data items is described by the concurrent composition of three CHP processes, as shown in Figure 1. Each process, shown as a box, has one input port, L, and one output port, R. The process receives an input value on L and sends the received value out on R. The receive operation blocks if there is no data to be received, and the send operation blocks until the process being sent to is ready to receive data. The connections between the VLSI implementation of these processes uses multiple wires (three per port, two for data and one for control), even though each process in the example only has one input and one output port. The multiple wires are required to implement synchronization protocols for communication [8]. While the FIFO shown in Figure 1 has three concurrently executing processes, it logically behaves as one, three-place FIFO.

This example illustrates several similarities between asynchronous VLSI circuits and networks.

- Each elementary process in an asynchronous VLSI circuit is sequential; an elementary component of a network node can only execute one action at a time.
- A collection of concurrently executing CHP processes can behave like one logical entity; a network node can consist of multiple, concurrently executing components that together provide the functionality of a single node.
- Different CHP processes exchange information by sending messages to each other; network

nodes also exchange information by message-passing.

- In both asynchronous VLSI computations as well as networks, there is no centralized control that sequences all events in the system.

It is this set of similarities that allows us to draw an analogy between asynchronous VLSI and communication networks. In particular, we are interested in the following question: can we model networks by designing them in silicon and simply observing the behavior of the asynchronous VLSI circuit implementation?

3 Network on a Chip

Inspired by the similarity between networks and asynchronous VLSI systems, we discuss how we can model networks in silicon. While asynchronous VLSI systems share properties with communication networks, there is one major difference: message-passing speeds in silicon are many orders of magnitude faster than those in networks. Our approach exploits this difference to construct network models in silicon that can produce simulation results faster than the networks themselves, and thus faster than real-time. Such a custom asynchronous VLSI approach will be energy-efficient compared to a traditional software-based solution, and the resulting implementation could be integrated into existing networking devices.

Modeling a wireless network in silicon is similar in spirit to existing component-based software models of wireless networks [3, 14, 13]. The main point of departure is that several components in the simulation infrastructure are directly modeled in VLSI. The NoC architecture combines these dedicated VLSI components with software models that are used to program the hardware to provide a flexible simulation infrastructure.

Our proposed architecture includes, but is not limited to, the following major components:

- Channel models
- Data Link and Medium access control
- Routing
- Node models

The channel model is responsible for determining the quality of the communication links between network nodes. This model simulates effects such as signal attenuation and fading. We require an explicit channel model because the silicon implementation contains reliable communication links; additional circuits are required to make these silicon communication channels behave like wireless channels. The channel model is directly implemented in hardware with parameters that are software-controlled, so as to avoid incurring software overhead in channel modeling.

Once a message arrives at a node after passing through a channel model, it enters the hardware model of the data link and medium access control layer. This layer models the link delay and bandwidth, including potential overhead incurred due to media access control algorithms. Like the channel model, this model contains programmable registers that can be externally set to control its behavior.

Hardware support for routing in the NoC architecture is used for two purposes: (a) to provide a fast routing technique for multi-hop routing protocols in ad hoc networks; (b) to provide a flexible mechanism by which a static hardware structure can be programmed to support multiple network topologies as well as node mobility. Silicon is a static medium; if we designed our simulator architecture to have the same topology as an existing network, we could only simulate one topology with a single chip. In addition, this restriction would severely limit the amount of mobility that could be modeled by such a simulator architecture. Instead of limiting the simulator's capabilities in this manner, the NoC architecture treats the network being simulated as a virtual topology that must be mapped to a physical VLSI implementation by configuring the hardware implementation. This choice allows us to model mobility by simply reprogramming the hardware. Figure 2 shows how a small network can be mapped to a static VLSI array by programming the hardware links in the appropriate way. The highlighted links correspond to those with network traffic. When a node moves, the network topology changes and the hardware can be reconfigured to reflect the new topology. The NoC architecture borrows this idea from field programmable gate arrays (FPGAs) where software control can reprogram both routing and logic functions to change the behavior of a single chip. The primary difference between the NoC architecture and an FPGA is that our approach allows a programmer to precisely control network link delays, and is customized for network simulation. Also, existing off-the-shelf FPGA solutions all use a global clock which introduces an artificial global synchronization constraint that could affect the accuracy of a network simulation that used a pure FPGA-based model of the entire network.

The node model is the most flexible part of the NoC architecture. This model corresponds to the rest of the behavior of a local network node which includes the network protocol, transport layer, application layer, as well as node mobility. The node model can also control the behavior of the other hardware components described above, thereby allowing the simulation of adaptive protocols.

Another way to think about this implementation strategy is to treat each piece of the VLSI network model as a component that would traditionally be an entity in a parallel discrete-event simulation. Indeed, the similarities between the strategy we have described and strategies for component-based software network simulation are striking [13, 14]. The key difference between the VLSI implementation when compared with a parallel simulation environment is that the asynchronous VLSI implementation mimics the network itself and has adjustable delay parameters. The net result is that the time at which an event occurs in the asynchronous circuit will be a *scaled* version of the actual time at which the event occurs in the network. Therefore, the VLSI simulation does not suffer from the event-ordering problem, because the event-ordering problem is an artifact of the discrete-event simulation algorithm and is not a physical phenomenon.

Our simulation architecture can also take advantage of the technological scaling properties of VLSI, sometimes referred to as Moore's Law. Moore's Law states that the number of devices that fit on a single chip doubles every eighteen months. In addition, the speed of transistors on a chip is also increasing at an exponential rate. What this means is that with technology scaling, not only will the performance of our network simulator improve, but the number of nodes we can simulate in a fixed amount of area will increase as well.

4 Evaluation

We created various network topologies and simulated them using both a regular network simulator and an asynchronous VLSI simulator. For network simulation we used ns-2 [10]. The circuits we designed were simulated with SPICE as well as `chpsim`, a compiled simulator for asynchronous circuits that reports accurate timing information.

Asynchronous VLSI model. An important property of the VLSI network simulator that we must preserve is absence of deadlock. It is well known that any hardware routing strategy that permits routing cycles cannot be deadlock-free. Networks, on the other hand, could have cycles in their routing protocols. To circumvent this problem, we adopt the same behavior as a network: when routing resources are exhausted, network packets are dropped. Our hardware model uses a credit-based flow-control system: every node begins with a certain number of credits for each link, corresponding to the amount of data buffering available for that link. Nodes only send packets on links for which they have credit. Nodes drop packets when incoming traffic is too high, just like a communication network. Whenever a node receives a packet that completely fills its buffer for that port, the node drops one packet in that buffer and returns credit to the sender. The combination of credits and dropping packets ensures the VLSI circuit will be deadlock-free for arbitrary routing tables. Different queue management strategies drop different packets when overflow occurs, and sometimes even before overflow is observed. In the simulation runs that we report below, we modeled the droptail scheme used by ns-2 that always drops the latest packet in the buffer.

The VLSI implementation of a network node consists of a routing table, message buffers, and models for local packet generation. When a message arrives from the network, it passes through the routing table if necessary and is either routed to the next hop or is locally processed. If a link is congested, the message waits in a buffer until routing resources are available. The local node model injects messages into the network for different destination nodes. Figure 3 shows an example topology and node configuration, showing how an incoming message is processed by a node. The high-level description of the behavior of the VLSI node can be directly translated into an asynchronous circuit implementation using well-known techniques [8]. As an example of such a translation, the FIFO credit buffers that we use can be translated into the simplified circuit shown in Figure 4 (the staticizer circuit elements are omitted for clarity).

Network Configuration and Traffic. We wrote several topology generators that are capable of generating ring, grid, and random network topologies. The topology generator can create an ns-2 simulation script as well as a hardware description of the VLSI model for the network nodes and channels in the CHP language. Every VLSI network node was implemented using a collection of CHP processes. Routing decisions are made with the same system as the ns-2 simulations so as to mimic ns-2 behavior.

Our simulation setup can generate both fixed traffic patterns as well as random traffic patterns. For instance, in the ring topology, each node can be configured to send packets to nodes a fixed distance away, or to random destinations. For grids, each node can send messages to a fixed distance away in both dimensions. Using fixed and pseudo-random traffic patterns was useful when comparing the ns-2 simulations with the results obtained from `chpsim`, because both software and hardware networks had sources that produced identical traffic patterns. An example regular traffic pattern on a grid is shown in Figure 3.

Results. A number of simulations were performed using both ns-2 and `chpsim` for various network topologies and traffic patterns described above. We measured end-to-end latency and overall network throughput in both ns-2 as well as `chpsim`. We found that the network statistics measured by `chpsim` were scaled versions of the statistics generated from ns-2, confirming our belief that the two networks behave in an identical manner.

To illustrate our results, we show the statistics we gathered from twenty different traffic pattern/topology combinations. These simulations correspond to a network where the average number of hops traversed by each packet ranges from four to twenty four. Each simulation corresponds to one minute of real time. Figure 5 shows the ratios of latency and throughput measured from ns-2 and the hardware simulation for the twenty different scenarios. The ratios are all scaled by the same factor so that a ratio of one means that the software and hardware simulations produce identical results. All the measured data points lie within the two horizontal lines on the graph that bound the region corresponding to a $\pm 1\%$ deviation from ideal behavior, showing that both latency and throughput measurements from the hardware simulation match those reported by ns-2.

Figure 6 shows the simulation times measured for the software simulation on an 800 MHz Pentium III PC running FreeBSD 4.2. Since the traffic patterns we chose had different average hop counts per packet, the time taken by the ns-2 simulation varied as a function of the average hop-count as should be expected. As opposed to this, the time reported by our circuit simulations showed that the hardware simulation took the same amount of time to complete independent of the level of traffic. This is consistent with the fact that each simulation corresponded to a fixed real time interval. Finally, the second graph in Figure 6 shows the speedup that we should expect by using the hardware implementation, assuming the circuits were fabricated in a $0.35\mu m$ CMOS technology. The speedup graph has the same shape as the ns-2 simulation time graph because the hardware takes a fixed amount of time to simulate the different traffic patterns. Note that the magnitude of the speedup is in the 10^5 range, which roughly corresponds to the ratio of hardware message-passing cost to network message-passing cost.

While our graphs contain the results from twenty different simulated scenarios, our other results were similar to the ones we have shown in Figures 5 and 6.

5 Summary

We have taken the first steps toward designing a network on a chip. Our investigation to date has produced encouraging results, showing that our intuition that asynchronous VLSI systems and communication networks behave in a similar manner is indeed correct. Our simulations demonstrated that it is possible to predict network latencies and throughput using a hardware simulation approach, and that such an approach can lead to architectures that can simulate network behavior faster than real-time.

Acknowledgments

This work was supported by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564. The authors would like to thank Lang Tong for extensive discussions about wireless networks, and the anonymous reviewers for their helpful comments.

References

[1] Rajive Bagrodia, and Wen-Toh Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, **20**(4):224–238, April 1994.

[2] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xian Zeng, Jay Martin, and Ha Yoon Song. PARSEC: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, **31**(10):77–85, October 1998.

[3] James Cowie, Hongbo Liu, Jason Liu, David Nicol, and Andy Ogielski. Towards Realistic Million-Node Internet Simulations. *Proceedings of the 1999 International Conference on*

Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 28–July 1, 1999, Las Vegas, Nevada.

[4] F. Desbrandes, S. Bertolotti, and L. Dunand. Opnet 2.4: an environment for communication network modeling and simulation. *Proceedings of the European Simulation Symposium*, pp. 609–614, Delft, Netherlands, October 1993.

[5] Polly Huang, Deborah Estrin, and John Heidemann. Enabling Large-scale simulations: selective abstraction approach to the study of multicast protocols. *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–248, July 1998.

[6] David Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, **7**(3):404–425, July 1985.

[7] Rajit Manohar. A Case for Asynchronous Computer Architecture. *Proceedings of the ISCA Workshop on Complexity-Effective Design*, June 2000.

[8] Alain J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. In *Formal Development of Programs and Proofs*, UT Year of Programming Series, pages 59–80, 1989.

[9] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED. *Proceedings of ACM SIGCOMM 2000*, pp. 151–160.

[10] S. McCanne and S. Floyd. The ns network simulator. Available on the web from the following site: `http://www.isi.edu/nsnam/ns/`.

[11] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, **27**(1) January 1997.

[12] D. Schwetman. Hybrid Simulation Models of Computer Systems. *Communications of the ACM*, September 1978.

[13] Joel Short, Rajive Bagrodia, and Leonard Kleinrock. Mobile wireless network system simulation. *Proceedings of Mobicom '95*, pp. 195–209.

[14] Hung-Ying Tyan and Chao-Ju Hou. JavaSim: A component-based compositional network simulation environment. *Western Simulation Multiconference–Communication Networks And Distributed Systems Modeling And Simulation*, January 2001.

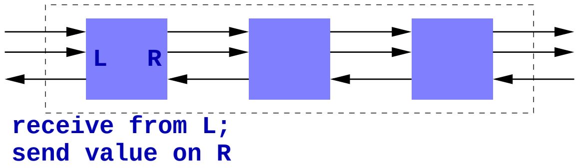

Figure 1. Three stage one-bit FIFO

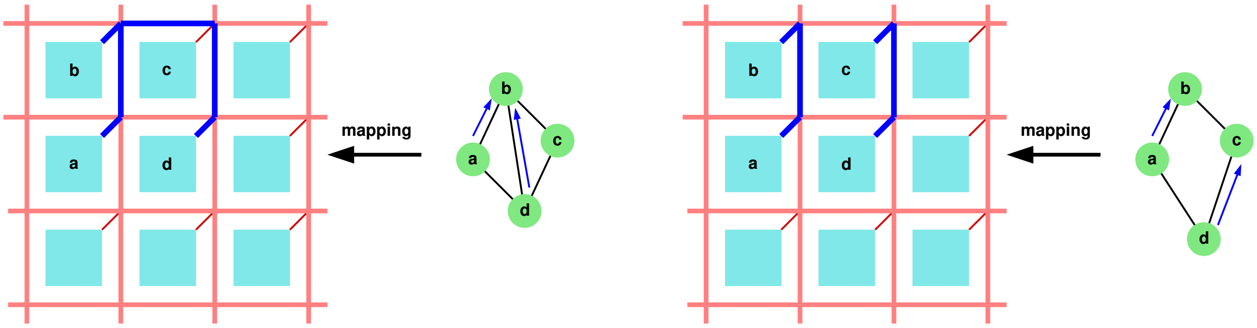

Figure 2. Mapping strategy, showing initial configuration followed by configuration after node motion.

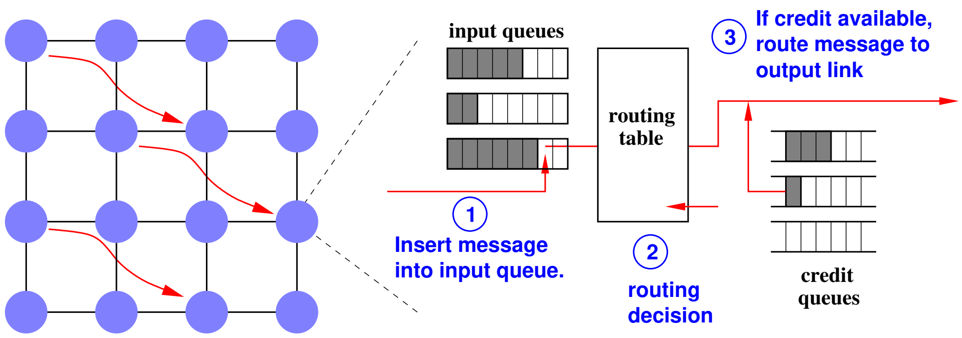

Figure 3. Grid topology showing sample regular traffic pattern. Edges represent bidirectional links, and sample packet paths are shown in red.

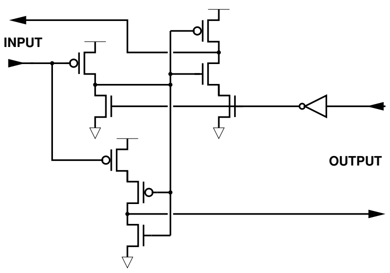

Figure 4. Circuit implementing credit FIFOs.

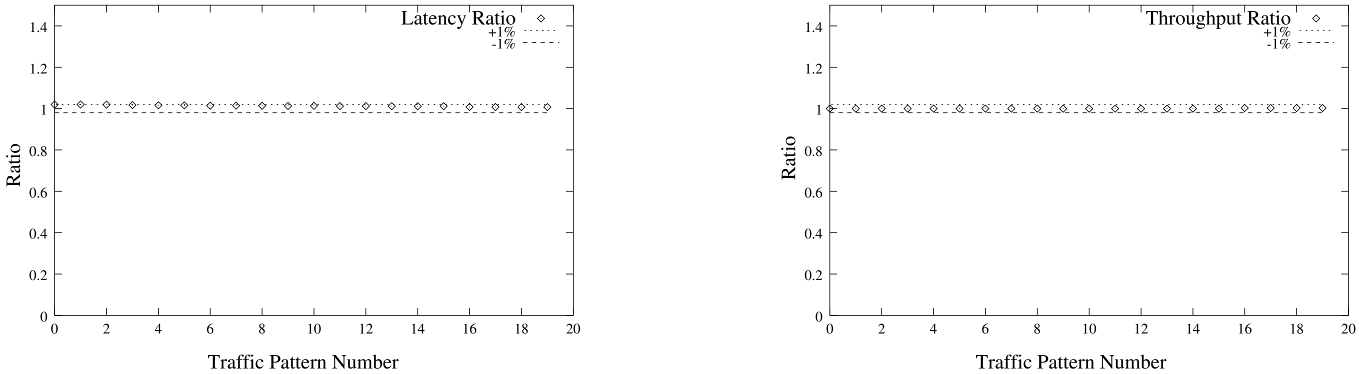

Figure 5. Ratio of normalized ns-2 and `chpsim` latency and throughput predictions for different traffic patterns

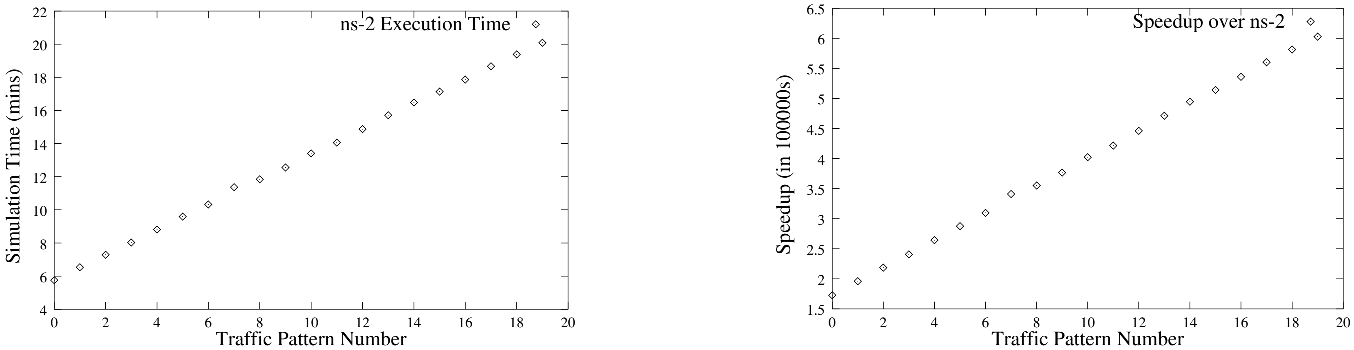

Figure 6. ns-2 execution times for different traffic patterns and the corresponding speedup