

WIRELESS NETWORK SIMULATION DONE FASTER
THAN REAL TIME

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Clinton Wayne Kelly, IV

August 2003

© Clinton Wayne Kelly, IV 2003

ALL RIGHTS RESERVED

ABSTRACT

We present a new kind of network simulator capable of simulating mobile ad hoc wireless networks faster than real time. Our design consists of two parts: a new synchronization protocol, continuous time-based synchronization (CTBS), for parallel discrete-event simulation, and a new piece of hardware, the Network on a Chip (NoC), designed to efficiently execute a CTBS-based simulator. We present preliminary studies that suggest our simulator will be able to simulate some ad hoc networks faster than real time.

Biographical Sketch

I was born in Fairfax, Virginia on 19 December, 1977. I lived in nearby Reston, Virginia with my parents—Alberta Allen Kelly and Dr. Clinton Wayne Kelly, III—for three years. When my sister, Emily Lindsay Allen Kelly, was born in 1981 we moved to Bethesda, Maryland, where my family still resides.

Throughout my life I have had the pleasure of visiting many different areas of the world. My parents have taken my sister and me to the Grand Canyon, the Galapagos Islands, the Swiss Alps, Kenya, Tansania, Zimbabwe, Batswana, South Africa, Namibia, Australia, New Zealand, London, the French countryside, Manchu Picchu, and the Amazon rainforest, among other places. I am truly blessed to have seen such a large portion of the earth at such a young age.

In 1996 I came to Cornell University as a member of the College of Engineering. I spent my entire junior year “studying” at the University of Sussex in southern England. Although I had traveled extensively throughout the world before my junior year, I had never *lived* in a foreign country; doing so was very educational.

In May 2000 I received a degree in Computer Science. I had originally intended to do my graduate work in computer graphics, but I so thoroughly enjoyed Professor Mark Heinrich’s ECE 475 class and the independent-study project that I did as a senior with Professor Rajit Manohar that I decided to do systems research. In the

fall of 2000 I enrolled as a graduate student at Cornell University, and the following spring I began working on the Network on a Chip. During my first year in graduate school I also joined the Cornell Bhangra team, and I met Smrita Sinha, who has made my life as a Ph.D. student very enjoyable.

For my maternal grandmother, Alberta Wood Allen.

Acknowledgements

Thanks to my advisor, Professor Rajit Manohar.

Thanks to Mom and Dad and Emily for their constant support. Thanks to Smrita for her love, patience, and understanding.

Thanks to Professor Lang Tong for his feedback. Thanks to Avneesh Bhatnagar for showing me how to use `ns-2`. Thanks to my compilers group—David Biermann, John Teifel, and Avneesh—for doing our first project while I wrote my first paper. Thanks to Mainak Chaudhuri for his helpful discussions. Thanks to Professor Mark Heinrich for his useful comments about my work.

Thanks to Jasjeet Singh Thind for introducing me to Bhangra, which gave me a creative outlet without which I would not have survived graduate school.

Thanks to Giuseppe Bianchi and Ilenia Tinnirello for giving me the source code for their IEEE 802.11 simulator, and to Jason Liu for giving me the source code for his stochastic MAC model. Thanks to Gokhan Mergen for giving me the source code for his Slotted Aloha simulator.

The work described in this thesis was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564.

Table of Contents

1	Introduction and Motivation	1
2	Traditional Network-Simulation Techniques	4
2.1	Discrete-Event Simulation	4
2.2	Parallel Discrete-Event Simulation	6
2.2.1	The Null-Message Protocol	10
2.2.2	Time Warp	11
2.2.3	Lookahead	12
2.2.4	Simulators Currently Used for Wireless Networks	14
2.3	Events in Wireless Network Simulations	15
3	Our Technique	18
3.1	Continuous Time-Based Synchronization	18
3.2	Examples using CTBS	20
3.3	Limitations of CTBS	23
4	Parallel Architectures	25
4.1	Networks of Workstations	25
4.2	DSM Machines	27
4.3	Message-Passing Multicomputers	28
5	The Network on a Chip	31
5.1	The Processor	32
5.1.1	The Processor Core	34
5.1.2	The Timer Coprocessor	35
5.1.3	Sending and Receiving Messages	37
5.2	The Interconnect	39
5.3	Deadlock Avoidance	41
5.4	Multi-Chip Simulation	42
5.5	A Simple Example	43

6	Evaluation of Scaling Properties	47
6.1	Simulation Setup	47
6.2	Network Scenarios	48
6.3	Optimizations	49
6.3.1	Moving Computation	50
6.3.2	Performing Radio Calculations	55
6.3.3	Our Final Critical Path	56
6.4	Results	58
6.5	Pros and Cons	60
6.5.1	Disadvantages	60
6.5.2	Advantages	61
7	Summary	63
7.1	New Protocols	63
7.2	Sensor Networks	64
7.3	The NoC Test bed	64
7.4	Future Work	65
A	The NoC Processor ISA	66
A.1	Introduction	66
A.2	State of the Processor	66
A.3	Description of Instructions	67
A.3.1	ALU Instructions (Reg, Reg)	67
A.3.2	Carry Instructions	68
A.3.3	Short Immediate Instructions	69
A.3.4	Event-scheduling Instructions	69
A.3.5	Bit-field Instructions	70
A.3.6	ALU Instructions (reg, imm)	70
A.3.7	Memory Instructions	71
A.3.8	Unconditional Branches	71
A.3.9	Conditional Branches	71
A.3.10	Miscellaneous	72
A.4	Opcode Encodings	72
A.5	Comments	72
	Bibliography	76

List of Tables

6.1	Events and their timestamps from a simulation of a MANET. Times are in simulated μs	54
A.1	Single-word instructions.	73
A.2	Double-word instructions.	74

List of Figures

5.1	The NoC with host.	32
6.1	Number of neighbours for a 100-node simulation.	49
6.2	Moving computation for an ack. The shaded boxes represent computation done for the events listed below. The optimization moves the computation that analyzes the incoming message and potentially creates an ack. Times are in simulated nanoseconds.	51
6.3	Moving computation for forwarding a route-reply packet. The shaded boxes represent computation done for the events listed below. The optimization moves all of the non-ack computation. Times are in simulated nanoseconds.	52
6.4	The time scales for simulations of CTBS-based simulations, running on the NoC, for MANETs of various sizes.	59
6.5	The time scales for simulations of CTBS-based simulations, running on the NoC, for various transmitter-turn-on times.	59

Chapter 1

Introduction and Motivation

A *mobile ad hoc network* (MANET) is a collection of mobile wireless nodes that combine to form a temporary network without any infrastructure or centralized control. Because of the mobility of the nodes in the network and the ensuing unreliability of the links between the nodes, MANETs require routing protocols that are different from those used in other, more stable networks. For example, protocols designed for wireless networks with stationary nodes typically require nodes to set up static, bidirectional links that exist for long periods of time. In a MANET, however, these links may fail frequently as nodes move out of transmission range of one another, causing routing protocols that rely on such links to perform poorly. The development of new, MANET-specific protocols has been the subject of a great deal of research over the past several years [29, 1, 42, 41, 43, 25, 48, 27].

Researchers typically cite three major uses for MANETs: emergency situations, military operations, and sensor networks [31, 48, 35, 29]. Networks used in such situations could easily contain many nodes—on the order of several hundred or even several thousand [35, 57, 4, 19]. A researcher designing a protocol to be used in one of these situations, therefore, would like to be able to test how well the protocol scales

as the number of nodes using the protocol increases. The two most frequently-used methods for evaluating scalability are analytic techniques and simulation. Because of the complexity (for example, frequent changes in network topology, and the importance of modeling the physical layer accurately [50]) of MANETs, however, the use of analytic techniques is often too complicated to be useful; analytic models of MANETs are often wrong [24]. This leaves simulation as the most viable option for evaluating routing protocols for MANETs.

Unfortunately, the combination of modern simulation methodologies and computer systems (i.e. processors, memory systems, etc.) is not sufficient to enable fast, detailed simulation of large MANETs. Most recent papers simulate networks containing no more than one- or two-hundred nodes [43, 16, 26, 27]; some actually simulate networks containing no more than *twenty* nodes [48, 29].

In this thesis, we present a novel network simulator that will use fine-grained parallelism to enable large-scale simulation of MANETs. This thesis makes two principal contributions to the field of wireless-network simulation:

1. Our simulator uses a new conservative synchronization protocol for parallel discrete-event simulation. We call this protocol *continuous time-based synchronization* (CTBS).
2. Our simulator runs on a collection of custom chip multiprocessors. Each processor contains hardware constructs that facilitate efficient execution of a simulator using CTBS. We call this hardware *the network on a chip* (NoC).

There are several potential benefits to this work. The first benefit is that researchers studying MANETs will be able to simulate larger networks than they can simulate now. This should enable them to better evaluate the scaling properties of certain network protocols and architectures, and to evaluate protocols on realistic

network sizes. The other benefits are less obvious: Consider that our simulator may not only be faster than traditional simulators, but also faster than *real time*. In this case, we believe that our simulator will enable an entirely new class of routing protocols that modify themselves based on in-situ simulation. Moreover, we plan to create sensor network nodes with designs based on the processors in the NoC. We further discuss these benefits in Chapter 7.

The rest of this thesis proceeds as follows. In Chapter 2, we discuss conventional network-simulation techniques, including two common synchronization protocols for parallel simulation; we describe our protocol in Chapter 3. In Chapter 4 we discuss the suitability of CTBS-based simulations for existing hardware platforms; we describe our hardware in chapter Chapter 5. In Chapter 6 we provide an evaluation of the scaling properties of our simulator for different network scenarios. In Chapter 7 we discuss our future work and the potential benefits of our simulator. Finally, Appendix A contains the ISA of the NoC processor.

Chapter 2

Traditional Network-Simulation Techniques

Researchers studying mobile ad hoc wireless networks typically use *discrete-event simulators* such as Glomosim [57, 4], Opnet [18], or ns-2 [38] to simulate network scenarios. In this chapter, we give a brief overview of discrete-event simulation, explain why it does not scale well, and examine ways to speed up discrete-event simulation through parallelization. We then discuss recent work in parallel simulation of MANETs. Finally, we provide a short description of the typical way in which discrete-event simulators model wireless signals.

2.1 Discrete-Event Simulation

A discrete-event simulator (DES) consists of several elements: (1) the state variables $S = (s_1, s_2, \dots, s_n)$, which define the current state of the system the DES is simulating; (2) a time-ordered *event queue*, which contains pending, timestamped *events*;

and (3) a global *clock*, which indicates the current simulated time. An event’s timestamp indicates the time at which the event would occur in the actual system the DES is simulating. In a simulation of a wireless network, an event could represent the beginning of a transmission, the expiration of a timer, or the movement of a node, for example.

The *simulation kernel* drives the simulation by repeatedly removing the first event (i.e. the event with the earliest timestamp) in the event queue, updating the clock to the timestamp of that event, and simulating the effects of the event (for the rest of this thesis, instead of saying, “simulating the effects of” the event, we will say, “executing” the event) by doing any of the following:

- Changing one or more of the state variables.
- Scheduling new events at times equal to or later than the time of the event the kernel is currently executing.
- Canceling previously-scheduled events already in the event queue.

It is crucial for the simulation kernel to always pick from the event queue the event, E_{min} , with the lowest timestamp. If the kernel were to pick another event, $E_{not-min}$, first, and if the simulation of $E_{not-min}$ modified state variables that the simulator read while simulating E_{min} , then the simulator might produce incorrect results. These errors resulting from out-of-order execution of events are called *causality errors*.

To illustrate this more clearly, consider a simulation of two tanks, $Tank_A$ and $Tank_B$, in a battlefield. Say that the event queue currently holds two events, $E_{A-fires}$ and $E_{B-fires}$, with timestamps $T_{A-fires}$ and $T_{B-fires}$, respectively. The event $E_{A-fires}$ indicates that $Tank_A$ has fired a shell at $Tank_B$. Likewise, the

event $E_{B-fires}$ indicates that $Tank_B$ has fired a shell at $Tank_A$. We assume that $T_{A-fires} < T_{B-fires}$. The simulator has some state variables that indicate which tanks are alive and which have been blown up.

If the kernel executes events in-order, then it will execute $E_{A-fires}$ first. Executing this event will change the simulation's state, since $Tank_B$ will now be destroyed. This means that the kernel must cancel $E_{B-fires}$; $Tank_B$ clearly cannot fire at $Tank_A$ if it has been blown up.

If, however, the kernel had executed $E_{B-fires}$ first, then it would have changed its state to indicate that $Tank_A$ had been blown up, and it would have canceled $E_{A-fires}$. This would have been a causality error.

If the kernel correctly models the effects of all events, and if all events have correct timestamps, then the DES will correctly predict the behavior of the given system. This makes discrete-event simulation attractive to researchers studying MANETs. Unfortunately, sequential discrete-event simulation does not scale well with the size of the network it simulates. Moreover, discrete-event simulations of MANETs often scale especially poorly, due to the broadcast-like nature of wireless networks: To simulate a single transmission by a node, the simulator may have to schedule many other events, each representing the arrival of the transmitted signal at a receiving node's antenna. Because of this poor scaling, researchers wishing to perform large-scale simulations of MANETs often turn to *parallel* discrete-event simulators (PDESs).

2.2 Parallel Discrete-Event Simulation

During the past several decades, researchers have devoted a great deal of time to improving the efficiency of parallel discrete-event simulation. Fujimoto [20] provides

an excellent overview of this work. While there are several different ways of parallelizing any simulator, the most common approach is to perform a *space-parallel decomposition*, in which the simulation model is divided—in the space domain—into several components, called *physical processes*.¹ For the rest of this thesis, whenever we discuss a parallel discrete-event simulation, we refer to a simulation based on space-parallel decomposition. For example, a simulation designer could choose to decompose a MANET into a set of physical processes where one process represents one node. In the simulator, there is a one-to-one mapping between each physical process and a corresponding *logical process* (LP).

We say that a parallel discrete-event simulator is composed of N logical processes, $LP_0, LP_1, \dots, LP_{N-1}$. These processes communicate by sending timestamped event *messages*; the k th message is a tuple (E_k, T_k) , where E_k is an event and T_k is its timestamp. Each logical process LP_i consists of the following elements: (1) the state variables that correspond to the physical processes the LP represents; (2) a time-ordered event queue; and (3) a local clock, $Clock_i$, that equals the timestamp of the LP's most-recently-executed event. Note that the components—state variables, an event queue, and a clock—of a logical process are similar to the components of a sequential DES. You can therefore think of a PDES as a composition of sequential DESs, each simulating a portion of a system, that communicate by sending messages containing events [2].

To avoid confusion between messages transmitted by wireless nodes in a real network and messages sent between logical processes in a parallel simulation, we will refer to the former as “packets” and the latter as “messages” for the remainder of this thesis. Moreover, we will not consider any parallel simulators that use shared

¹The authors of [32, 33] discuss partitioning the simulation model based on channel frequency.

variables. If two LPs need to read and write the same piece of data, they can emulate shared variables by passing messages.

Fujimoto [20] defines the *local causality constraint* for parallel discrete-event simulation. We say that a PDES satisfies this constraint if and only if every LP executes events in nondecreasing timestamp order. Put more formally, whenever LP_k executes an event E_i with timestamp T_i , $Clock_k \leq T_i$. Note that this constraint is sufficient but not always necessary: It is possible for a simulator to disobey the local causality constraint and still produce the same results as a sequential DES, as long as the events that it executes in incorrect order are independent.

To see how causality errors can occur in a PDES, we revisit our previous example. We divide our simulation into two physical processes, one representing $Tank_A$ and one representing $Tank_B$. We will have two kinds of events in our system, $E_{i-fires-at-j}$ and $E_{j-blows-up}$. The first event represents $Tank_i$ firing a shell at $Tank_j$, and the second represents a shell hitting $Tank_j$. This is a more realistic example than our previous one, because we model the difference between the time when a tank fires a shell at a target and the time when the shell hits the target. We represent our tanks with LP_A and LP_B .

In our new example, LP_A has an event, $E_{A-fires-at-B}$, in its event queue. This event has timestamp 10. Likewise, LP_B has an event, $E_{B-fires-at-A}$, in its event queue. This event has timestamp 12. We say that a shell needs only one time unit to move between the tanks. For the simulator to simulate this scenario correctly, LP_A should execute $E_{A-fires-at-B}$ first. It should then send a message, $(E_{B-blows-up}, 11)$, to LP_B , which should execute $E_{B-blows-up}$ and then cancel $E_{B-fires-at-A}$. If LP_B executed $E_{B-fires-at-A}$ before receiving the message from LP_A , it would clearly cause a causality error, since $Clock_B$ would be 12 when LP_B executed $E_{B-blows-up}$.

But how can LP_B know to wait to receive the message from LP_A , therefore avoiding a causality error? Avoiding causality errors by controlling when LPs execute events (or detecting when LPs execute events out-of-order, and then rolling back to a previous, correct state) is the job of the simulation’s *synchronization protocol*.

Synchronization protocols fall into two categories: *conservative* and *optimistic*. Simulations using protocols from the former category avoid causality errors by strictly adhering to the local causality constraint; logical processes in a conservative simulation only remove events from their local event queues when they are sure that doing so will not result in a causality error. Optimistic simulators, on the other hand, execute events without such assurances. Instead, an optimistic simulator executes until it detects a causality error, at which time it uses a “rollback” mechanism to return itself to an earlier, correct state.

In our previous example, a conservative simulator could use a protocol such as the null-message protocol [12] to ensure that LP_B would not execute $E_{B-fires-at-A}$ until it could *guarantee* that no other message would arrive from another LP with a timestamp greater than 12. In an optimistic simulator, on the other hand, LP_B might execute $E_{B-fires-at-A}$ before receiving the message from LP_A . In this case, the simulator would detect a causality error, since the timestamp of $E_{B-blows-up}$, 11, would be less than $Clock_B$, 12, when LP_B executed $E_{B-blows-up}$. The simulator would then roll back the state of the entire simulation to an earlier time, before the error occurred, and continue execution, making sure that LP_B waits for $E_{B-blows-up}$ to arrive this time. We now give short descriptions of one conservative protocol and of one optimistic protocol.

2.2.1 The Null-Message Protocol

Chandy and Misra [12] and Bryant [8] independently developed several conservative synchronization protocols for parallel discrete-event simulation, including the null-message protocol. Simulations using this protocol must statically specify the links between logical processes (if LP_i can send a message to LP_j , then we say there is a link between LP_i and LP_j) and requires that messages sent over these links have nondecreasing timestamps. Because of this ordering constraint, if logical process LP_i receives a message with timestamp T from LP_j , then LP_i knows it will *never* receive a message with a timestamp less than T from LP_j for the remainder of the simulation.

Each link is a queue that stores messages in first-in-first-out (FIFO) order and has a “clock.” If a link is nonempty, its clock is equal to the value of the message at the link’s head. If the link is empty, its clock is equal to the value of the last message removed it. We can think of the local event queue as a link from LP_i to LP_i .

A simulation can avoid causality errors if each LP always executes the event at the head of the link (or the local event queue) with the lowest clock; if the link with the lowest clock is empty, then the LP stalls. Although this behavior guarantees that every LP will execute events in nondecreasing timestamp order, a simulation using this protocol can easily deadlock.

To prevent deadlock, Chandy, Misra, and Bryant introduced *null messages*. These messages are used only to update the clocks of links, and do not represent any events in the physical processes the simulator is simulating. In general, if LP_i can *guarantee* that it will not send a message to LP_j with a timestamp earlier than T_{null} , then LP_i can send a null message to LP_j with timestamp T_{null} . If the link

from LP_i to LP_j is empty, then this null message will advance the clock of the link, thus preventing deadlock.

The logical process LP_i can determine the value of T_{null} by determining the smallest value among the clocks of all of its incoming links, and using some information about the simulation. For example, a network designer may know that there is a minimum delay, ΔT , between the timestamp of any message arriving at an LP and the timestamp of a resulting outgoing message. Therefore, if the smallest clock value at an LP is T_{min} , the LP can send a null messages on all of its outgoing links with timestamps $T_{min} + \Delta T$.

A proof exists that a simulation using null messages will avoid deadlock as long as no cycles exist in which the total timestamp increment of a message could be zero [20]. In other words, a simulation using null messages will only deadlock if a cycle of links exists where every link has the same clock value.

2.2.2 Time Warp

Time Warp [28] is the canonical optimistic PDES. In Time Warp, a message with timestamp T_s is said to be a *straggler* if it arrives at LP_i when $Clock_i > T_s$. When this happens, the simulator recovers by performing a *rollback* operation.

The arrival at an LP of a straggler indicates that the LP has already executed at least one event, E_{oops} , with a timestamp later than T_s . The early execution of E_{oops} may have had two kinds of undesirable effects: (1) The LP may have changed some of its state variables and (2) the LP may have sent messages to other LPs. LPs in Time Warp periodically save their state, so they can recover easily from effects in the former category. Correcting effects in the latter category requires sending a corresponding *anti-message* for every erroneous message that the LP sent after

executing E_{oops} (and any other events that followed E_{oops}). Messages containing actual simulator events are called *positive messages*. Whenever an LP receives an anti-message that corresponds to a positive message that it has already received, but not executed, it simply destroys both messages. If, however, the LP has already processed the positive message, then it must perform a rollback to reverse the effects of processing this message, and possibly send anti-messages itself. This process of restoring state and sending anti-messages continues until Time Warp cancels all of the erroneous effects of the straggler.

2.2.3 Lookahead

One of the most important concepts in parallel simulation is that of *lookahead*. If an LP at simulated time $Clock$ can predict with complete certainty that it will not send any messages until at least time $Clock + T$, then the LP has lookahead equal to T . Good lookahead improves the performance of both optimistic and conservative simulators [20]. A person can easily see how a conservative simulator benefits from lookahead: A logical process in a conservative simulator can use its lookahead to determine the timestamps for null messages. An LP with greater lookahead can send null messages with higher timestamps, allowing the receivers of the null messages to process events sooner (and therefore improving the overall performance of the simulator).

Optimistic simulators can take advantage of lookahead by using a technique called *lazy cancellation*[21]. The idea behind this technique is that a logical process may not need to send an anti-message for every positive message sent with a timestamp greater than that of a straggler. For example, say an LP sends a message, E_p , with timestamp T_p , and then receives a straggler, E_s , with a timestamp, T_s ,

such that $T_s < T_p$. In conventional Time Warp, the LP would immediately send an anti-message corresponding to E_p , and then restore its state to the latest saved state before T_s . In a simulator using lazy cancellation, however, the LP would perform the same state-restoration, but it would not immediately send the anti-message. Instead, the LP would wait until its clock advanced again to T_p , and then check to see if it was going to send a message identical to E_p . If it was going to do so, then it would not need to send an anti-message.

What does this have to do with lookahead? Simulators using lazy cancellation perform better than conventional optimistic simulators when the transmission of an event from one LP to another is often independent of recent, earlier events executed by the sender. This “independence” is the lookahead of the LP sending the message. Moreover, an optimistic simulator using lazy cancellation can take advantage of average-case behavior. Say, for example, that the sending of E_p is *usually*, but not always, independent of the execution of E_s . An optimistic simulator using lazy-cancellation will send E_p early, without penalty, most of the time. A conservative simulator, however, will not be able to *guarantee* that it will send E_p , so it will not be able to send a null message with timestamp T_p . Also, notice that, although both optimistic simulators using lazy cancellation and conservative simulators using null messages can take advantage of lookahead, a designer of one of the latter simulators must explicitly program his simulator to exploit lookahead, while a designer of one of the former simulators need not have any knowledge of the lookahead inherent in the physical system for which he is designing a simulator. The requirement that the simulation designer recognize the lookahead in the physical system is a major drawback to conservative simulators. We will revisit this when we evaluate our technique in Chapter 6.

2.2.4 Simulators Currently Used for Wireless Networks

While there has been a great deal of work devoted to the study of parallel simulation of cellular wireless networks [34, 23, 9, 10, 11, 5, 6, 7, 37, 49], parallel simulation of MANETs is a relatively-unexplored field. The only examples of simulations of large (i.e. around 10,000 nodes) MANETs occur in works from the UCLA Parallel Computing Laboratory [54] and from Dartmouth College [17].

The UCLA Parallel Computing Laboratory has developed a parallel simulator called the Global Mobile Information System Simulator (Glomosim) [57, 4] using a parallel simulation language called the Parallel Simulation Environment for Complex Systems (Parsec) [3]. The authors of [4] aim for Glomosim to be able to simulate networks containing as many as 100,000 nodes with a “reasonable” execution time, and claim to have already used Glomosim to simulate networks as large as 10,000 nodes. The paper contains both a case study that simulates a network with 81 nodes for 400 simulated seconds, and speedup results for simulations of larger networks. The authors claim a speedup of four times with six processors for the largest network—10,000 nodes—that they simulate.

The authors of [57] give speedup results for Glomosim simulations of MANETs with at most 3,000 nodes. The greatest speedup in the paper is a factor of nine, with sixteen processors, using an IBM 9076 SP (a distributed-memory multicomputer). Similarly, [3] contains the results of simulations of wireless networks up to 3,000 nodes, with a maximum speedup of slightly less than eight, on sixteen processors.

Because Glomosim is built on top of Parsec, it can use any of Parsec’s parallel synchronization protocols [3] (Parsec has both conservative and optimistic protocols). However, the authors of [57] report results for Glomosim when using only conservative protocols; they may have chosen not to use optimistic protocols be-

cause of the large cost of saving the state for an LP (which may be simulating several hundred to several thousand nodes) in a simulation of a MANET [39].

Although more papers than are possible to cite in this thesis have used Glomosim to evaluate various network protocols and architectures, the parallel version of Glomosim is not freely available; we therefore conclude that the researchers using Glomosim have only used it as a sequential DES. However, the commercial successor to Glomosim, Qualnet [45], is available in parallel form.

The authors of [35] use the Simulator for Wireless Ad Hoc Networks (SWAN) to simulate a network with 10,000 nodes using a simplified MAC-layer model. They report that simulating 1,000 simulated seconds took more than ten hours to complete with five processors. The paper does not include speedup results (presumably because a one-processor simulation would take several days, and because the paper was meant as merely a “proof-of-concept”). Because SWAN is not yet publicly-available, no other researchers have used it for their research.

Though the performance achievable with Glomosim and SWAN is impressive, we believe that simulators using our synchronization protocol and running on the NoC will be able to simulate many large MANET scenarios even faster. In the next chapter we discuss our synchronization protocol.

2.3 Events in Wireless Network Simulations

Before describing our protocol, however, we give an overview of the way that simulation designers typically use events to model wireless networks. Specifically, we give a short description of the radio layer, since LPs in simulations using our protocol only exchange events through this layer.

A MANET simulator usually uses two events per node to model a wireless transmission: The first event represents the beginning of the transmission, and the second event represents the end of the transmission. To see how this works, we use an example, using three nodes, node A , node B , and node C . Say that node A transmits a packet at time $t = 5\mu s$. Naturally, the packet reach node B or node C only after some time, called the *propagation delay*, has elapsed. Let us say that node B is closer to node A than node C is, so the propagation delay to node B from node A is $2\mu s$, and the propagation delay to node C from node A is $3\mu s$. This means that we will have three events simulating the beginning of the transmission of the packet: one for node A with its timestamp equal to $5\mu s$, one for node B with its timestamp equal to $5 + 2 = 7\mu s$, and one for node C with its timestamp equal to $5 + 3 = 8\mu s$. If we were using PDES to simulate this network, then LP_A would send messages to LP_B and LP_C containing their respective events.

The second event at each node represents the end of the transmission. Say that the transmission of the packet lasts for $100\mu s$. This means that node A , node B , and node C will schedule these events with timestamps equal to $105\mu s$, $107\mu s$, and $108\mu s$, respectively. If we were using PDES to simulate this network, then each logical process would schedule this second event *itself*: Neither LP_B nor LP_C would receive a message from LP_A telling them to simulate the end of the transmission. Presumably, the first message would contain a field indicating the duration of the simulated transmission.

In most MANET simulators, an LP performs all radio calculations when it executes the first event (which represents the beginning of the transmission). The calculations can include determining path loss and fading, for example. If an LP executes an event representing the beginning of a transmission after it has already

executed *another* event representing the beginning of *another* transmission (but before executing the event representing the end of the transmission), then the LP must decide whether it should simulate a collision. This decision usually involves determining the strengths of the signals of both incoming transmissions.

In the next chapter, we describe our technique for MANET simulation. The previous discussion of how typical simulators use events to model wireless networks should be useful in understanding how our technique can be used to simulate MANETs.

Chapter 3

Our Technique

In this chapter we propose a new synchronization protocol, continuous time-based synchronization (CTBS), for parallel discrete-event simulation. We developed this protocol with the goal of enabling massively-parallel simulation of large-scale MANETs. By “large-scale,” we mean containing 1,000 to 100,000 nodes; when we say we want a “massively-parallel” simulation, we mean that we desire a simulation with one processor—and therefore one LP—for every node in the simulated network.

Since our aim is to simulate such large-scale networks, our chief concern is how simulations using our protocol will *scale*. A protocol that has a high overhead, and therefore performs poorly relative to a sequential DES for small simulations, will still be acceptable as long as it scales well when we use thousands of nodes.

3.1 Continuous Time-Based Synchronization

The main factor limiting the scalability of simulations using either optimistic or conservative protocols is the increase in the number of *synchronization messages*

(anti-messages or null messages) as the number of LPs in each simulation increases. This flooding of synchronization messages leads us to create our protocol, which does not use *any* synchronization messages: Every message in a CTBS represents an actual event in the corresponding physical system.

There are two key rules behind CTBS:

1. Simulation proceeds at a scaled version of real time: An LP will not execute an event until the event’s timestamp is greater than or equal to the scaled version of the current time.
2. Every message must arrive at its destination LP before the scaled version of the current time equals or exceeds the timestamp of the message’s enclosed event. Upon receiving the message, the destination LP places the message’s enclosed event in its local event queue.

A simulation using CTBS clearly satisfies the local causality constraint: Since every LP’s local event queue is ordered, for a causality error to occur a message containing an event, E_1 with timestamp T_1 would have to arrive at an LP *after* that LP has already executed an event E_2 with timestamp T_2 such that $T_2 > T_1$. However, our first rule states that the LP should not have executed E_2 until the scaled version of the current time was equal to or greater than T_2 . Therefore, according to our second rule, the message containing E_1 should have already arrived, and E_1 should have been in the event queue when the LP executed E_2 . This is a contradiction, however: The LP would not execute E_2 before E_1 if both events were in its event queue. Therefore, a simulation using our protocol satisfies the local causality constraint.

For the rest of this thesis, we call the time scale of a simulation s . We say that the “scaled version of the current time” is equal to $s \times T_{real}$, where T_{real} is the actual current time. Therefore, if s is greater than one, our simulator will proceed faster

than the system it is simulating. If s is exactly one, then our simulation should proceed at the same speed as the system it is simulating.

3.2 Examples using CTBS

We now use two examples to demonstrate how our protocol works. Our first example compares how a simulator using the null-message protocol and a simulator using our protocol would simulate the same scenario. We compare a simulator using CTBS against a simulator using the null message protocol instead of, for example, a simulator using an optimistic protocol, because both CTBS and the null-message protocol are *conservative* protocols. Our second example provides a bit more insight into the real-time constraints imposed on a simulator using our protocol.

Consider a logical process, LP_A , simulating $Tank_A$. Its clock, $Clock_A$ is equal to T_0 . At the head of the event queue of LP_A is an event, $E_{A-fires-at-B}$, with timestamp T_{fire} . This event corresponds to $Tank_A$ firing a shell at $Tank_B$.

In a simulation using the null-message protocol, LP_A cannot remove from its event queue $E_{A-fires-at-B}$ until the clocks of all of its incoming-message queues are greater than or equal to T_{fire} . In the worst case, this means that LP_A must receive a null message with a timestamp of at least T_{fire} from *every* LP capable of sending a message to LP_A . In this simulation, the LPs capable of sending messages to LP_A correspond to the tanks capable of firing shells that would collide with $Tank_A$.

In a simulation using our technique, LP_A must simply wait until $s \times T_{real}$ is equal to T_{fire} to remove from its event queue $E_{A-fires-at-B}$. That is, the simulation must run for T_{fire}/s time units of *actual* time before LP_A can execute an event with a timestamp equal to T_{fire} .

This example should provide a good insight into the scaling properties of our protocol. In a simulation using the null-message protocol, LP_A must wait for the clock of all of its incoming-message queues to advance before it can execute $E_{A-fires-at-B}$. In a simulation of a large number of tanks, this could take a long time, since all of the LPs with links to LP_A may also be waiting for *their* incoming-message queues' clocks to advance. A person can easily see how this overhead could increase as the number of tanks increases. In a simulation using CTBS, however, the time that LP_A must wait until it can execute E_{send} is independent of the number of tanks and independent of the state of the logical processes with links to LP_A . Moreover, LP_A does not need to know the set of LPs that can send messages to it.

Our next example shows some of the real-time constraints of a simulation using CTBS, and also makes clear the difference between the scaled version of the current time and the *Clock* of a logical process.

Say that a logical process has the following events in its event queue (the subscript of an event indicates the event's timestamp, in simulated μs): $E_{10}, E_{12}, E_{20}, E_{22}$. We assume our logical process requires $4\mu s$ of real time to execute any event. For simplicity, we assume that the time scale is one: The current time and the scaled version of the current time are the same. We refer to the current time as T and the simulated time as *Clock*.

When T is $10\mu s$, the LP knows that it will not receive any messages from other LPs with timestamps less than $10\mu s$, so it can safely execute E_{10} . This execution takes $4\mu s$, after which, $Clock = 10\mu s$ and $T = 14\mu s$.

The LP can then execute E_{12} , since any messages with timestamps less than $12\mu s$ would have arrived already. Now imagine, however, that a message with event E_{21} arrives while the LP is executing E_{12} . The LP places E_{21} into the queue after

it executes E_{12} . It then executes E_{20} , E_{21} , and finally E_{22} . Now consider what happens if executing E_{22} results in the transmission of an outgoing message. Since $T = 20\mu s + 2 \times 4\mu s = 28\mu s$ when the LP begins executing E_{22} , the message will only arrive at its destination LP on-time as long as the timestamp, T_{msg} , of the message is greater than T plus the time spent doing computation, T_{comp} that creates the message plus the latency, $T_{latency}$, to send the message to the destination LP. If $T_{msg} < T + T_{comp} + T_{latency}$, a causality error may occur at the destination LP. A simulation designer can avoid such causality errors by decreasing the time scale.

From this example, we point out two factors that can dramatically affect the performance of a simulation using CTBS. The first, and most evident, is the value of $T_{latency}$. Decreasing the latency to send a message between LPs allows a higher time scale. The second factor is the amount of lookahead in the system. We can write our constraint for correctness from the previous example as $T_{msg} - T_{comp} - T > T_{latency}$. An LP in a simulation with greater lookahead can send messages earlier, decreasing T in our equation and therefore increasing the left-hand side of the inequality. This will allow a higher time scale.

Another important point to notice is the difference between the *simulation time*, or *Clock*, of a particular LP, and the elapsed *real time* for the entire simulation. Remember that, for a logical process LP_i , $Clock_i$ is equal to the timestamp of LP_i 's most-recently-executed event. At any given real time, each LP can have a different *Clock*. On the other hand, the elapsed real time is a property of the entire simulation and is always equal for every LP. (It should be obvious that there will never exist a *Clock* that is greater than the scaled version of the elapsed real time.)

3.3 Limitations of CTBS

We cannot use CTBS for simulations of arbitrary physical systems, however. Recall that one of our rules for correctness is that every message must arrive at its destination at a time earlier than its timestamp. Assuming that every LP runs on its own processor (in a chip multiprocessor or in a cluster, for example), the time to send a message from one LP to another will always be nonzero. Therefore, the scaled version of the real time at which every logical process sends every message must be earlier than the messages' timestamps. Hence, we cannot simulate systems without lookahead.

In simulations of MANETs however, there is always some lookahead: Messages in such simulations represent the beginning of wireless signals. There are typically at least two delays between the time at which a transmitting node decides to broadcast a signal and the time at which a receiving node receives the signal: the transmitter-turn-on time and the propagation delay. We will explore the relationship between these delays and the time scale of our simulations further in Chapter 6.

The performance of a simulation using one synchronization protocol relative to the performance of a simulation using another synchronization protocol depends upon the time spent determining when it is safe to execute certain events (or, for an optimistic protocol, the time spent saving state, detecting causality errors, and rolling back). An LP in a simulation using the null-message protocol must wait until the link¹ with the lowest clock contains a message. An LP in a CTBS must wait until a scaled version of an event's timestamp before executing the event. The main purposes of this thesis are to explore the scaling properties of CTBS and to describe the hardware needed to allow such simulations to operate efficiently. A

¹When we say "link," we include both the incoming-message links *and* the local event queue.

study directly comparing the null-message protocol and our technique is a part of our future work; we address it further in Chapter 7. In the next chapter we discuss currently-existing hardware platforms that could support PDES using CTBS.

Chapter 4

Parallel Architectures

We mentioned in Chapter 1 that one of the two major contributions of this thesis is the design of a chip multiprocessor (CMP) that exists specifically for executing simulators using CTBS. In this chapter, we examine other parallel architectures and discuss why none of them is an optimal platform for CTBS-based simulation of MANETs. We also discuss some features of these architectures, especially the message-passing architectures in Chapter 4.3, that we will include in our design. These parallel architectures fall into three categories: networks of workstations (NoWs), distributed shared-memory (DSM) machines, and message-passing multicomputers.

4.1 Networks of Workstations

Networks of workstations have several features that make them desirable for running MANET simulations using CTBS:

- NoWs are usually composed of commodity, off-the-shelf (COTS) components, which are very affordable.

- Most universities and corporations already have large NoWs.
- NoWs are useful for many operations other than running MANET simulations using CTBS.
- PCs in a NoW communicate by passing messages, just as LPs in a simulation do.

Unfortunately, however, there are several reasons why we cannot use NoWs for our simulations. The most important reason is that the latency to pass messages in a NoW is large. For example, the latency for short messages in a system using a M3F-PCI64C Myrinet interface [40] is $7.6\mu s$. (This number does not count the message-reception software overhead, which could be much greater.) We can use this latency figure to estimate the time scale of a MANET simulation running on a NoW. Consider that the lookahead in such a simulation is typically equal to the simulated radio's transmitter-turn-on time plus the propagation delay between two nodes. Given the transmitter-turn-on time specified by the IEEE in the specification for the IEEE 802.11 DCF is $5\mu s$, and using a propagation delay of $2\mu s$ we get the following constraint:

$$5\mu s + 2\mu s > s \times 7.6\mu s. \quad (4.1)$$

We can then solve for the timescale, s :

$$s < 7/7.6 = 0.92 \quad (4.2)$$

This is unacceptably slow.

Essentially, a NoW provides too much computational power per node and not enough message-passing efficiency. A NoW is well-equipped for parallel applications with high computation-to-communication ratios. Traditional latency-hiding

techniques, such as prefetching, would not be effective in CTBS-based simulations of MANETs because the communication patterns in such simulations are unpredictable. A CTBS-based simulation of a MANET is not such an application, however.

4.2 DSM Machines

DSM machines are immediately *unattractive* for our simulations for a simple reason: Such machines are designed to hide message-passing from programmers, and instead provide a coherent, shared memory. This design is wasted for our simulations, however, because we do not need a coherent memory. Because each processor simulates a single LP, and because LPs do not share variables, the processors will never modify each other's memories. Nevertheless, a programmer can easily design message-passing programs (such as our simulator) to run on DSM machines. Our simulator is somewhat different than many other message-passing programs, however, in that its message-passing is not predictable: A shared-memory implementation of our simulator would require frequent asynchronous receives.

The latency for a remote read in a state-of-the-art DSM machine, such as the AlphaServer GS320 [22], is 136 ns (for a 2-hop, clean, pipelined, independent read). However, this number does not include the software overhead for the asynchronous reception of a message. This overhead will be on the order of tens of microseconds [13], giving us a value for the timescale similar to that we calculated for a NoW above. Moreover, there are several other problems with a DSM-based approach. The first is that even the largest DSM machines are only on the order of 512 nodes [13]; we would like to simulate wireless networks containing several times as many nodes. Moreover, such machines are not inexpensive. Second, using a DSM

to run an application that does not use any shared memory is highly inefficient: A processor in a typical DSM machines is far more powerful and contains far more memory than it requires to simulate a single node in a wireless network. Finally, we can design a machine that will provide an efficient mechanism for determining the time when an event's timestamp is equal to the scaled version of the current time.

4.3 Message-Passing Multicomputers

Finally, we discuss other parallel architectures, such as MIT's RAW project [55, 51] and J-machine [14], and the Caltech Mosaic [46]. Both of these architectures contain processors that communicate by passing messages, making them natural choices for CTBS-based simulation of MANETs. In this chapter we will discuss the RAW microprocessor in more detail than we discuss Mosaic, since the former is closer to the chip we present in Chapter 5.

The Raw microprocessor is a tiled collection on a single chip of 16 simple, message-passing processors. The present Raw implementation supports the "glueless" connection of up to 64 Raw chips, providing a total of 1,024 tiles (processors) [51]. Clearly, the Raw microprocessor is a very attractive system to use for our simulator. Unlike DSM machines, Raw does not have any coherence protocol. It also supports efficient, dynamic routing of variable-length messages: In an uncongested network, a message's header (which contains the length of the message) will reach its destination in $5 + X + Y$ cycles, where X and Y are the number of horizontal and vertical hops, respectively. Since the Raw microprocessors runs at 255 MHz, this is a very low latency.

We can also use a feature in the Raw microprocessor's architecture [52] to efficiently determine when the scaled version of the current time is greater than or

equal to the timestamp of the event at the head of the event queue. Each Raw microprocessor contains a 64-bit timer that tracks the number of clock cycles since reset. The current value of this timer is stored in the WATCH_VAL register. An exception occurs whenever the value of WATCH_VAL equals the value of another register, WATCH_MAX. We can therefore use WATCH_MAX as the value of the timestamp (divided by the time scale) of the event at the head of the event queue.

So why not just use Raw for our simulator? There are several reasons:

- Raw is by no means a COTS component.
- The Raw processors are more powerful than processors designed for our simulator would be. We do not need floating point units or integer multiply/divide units, for example. (Radio calculations require floating point and multiplication and division; we explain how to perform these calculations before the simulation in Chapter 6.)
- Because the Raw processors are more powerful than we need, they are also much bigger. While the Raw design supports the connection of up to 64 chips, containing 1024 processors, the networks we wish to simulate could require 10-to-100 times as many processors. Simpler processors would be smaller, allowing us to fit more on one chip. And an architecture designed for our simulator could support configurations with more than 64 chips.
- Our design would be more elegant if the arrival of a message caused an interrupt in the processor, as does the scaled version of the current time equalling the timestamp of the event at the head of the event queue. Moreover, we would like to have multiple versions of the WATCH_MAX register in the Raw microprocessor. If our processor had one register for every event in the event

queue, then our simulation would have almost *zero* overhead related to ordering the event queue: Inserting a new event in the queue would be as simple as setting an unused register to the value of the event's timestamp.

In the next chapter, we discuss the design of a chip multiprocessor that is similar in many ways to Raw, but with several differences addressed above.

Chapter 5

The Network on a Chip

Based on the discussion in our previous chapter, the ideal computer for our simulator would have the following characteristics:

- It would be composed of a collection of small, simple, message-passing (*not* shared-memory) processors, several of which can fit on a single chip.
- We should be able to gluelessly assemble enough chips to create a computer with 100,000 processors.
- Two types of events should interrupt our processors: (1) the scaled version of the current time becoming equal to the timestamp of any event in the event queue and (2) a message arriving.

With this in mind, we present in this chapter the *Network on a Chip* (NoC): A collection of simple, message-passing processors, designed to facilitate CTBS-based simulation of MANETs. Figure 5.1 gives a conceptual picture of what the NoC will look like (the NoC will actually have on the order of 100 processing elements per chip, not just nine); this figure also depicts the *host*, an off-chip workstation that

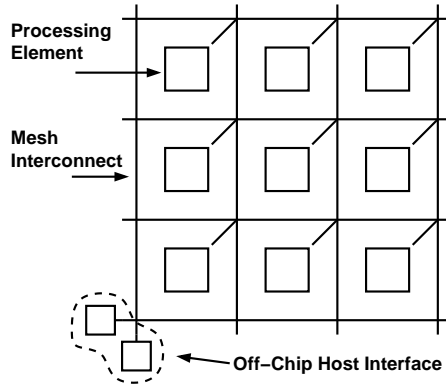


Figure 5.1: The NoC with host.

initializes the NoC, tracks statistics, handles errors, and provides a way for a user to interact with the NoC.

We begin this chapter by discussing the architecture of the NoC processors; we then describe the links that allow the processors to exchange messages. After that we explain how we can guarantee the NoC will be deadlock-free and we discuss simulations that use more than one chip. Finally, we give a short example illustrating how a CTBS-based simulation of a MANET would execute on the NoC.

5.1 The Processor

An LP running on our processor needs an efficient way to manage its event queue: The LP needs to be able to easily insert new events into its queue, cancel previously-inserted events, and receive notification if the scaled version of the current time has reached or exceeded the timestamp of the event at the head of the queue. To this end, we provide every processor with a *timer coprocessor* and the appropriate extensions to the processor's instruction set architecture (ISA) to use the coprocessor. In Chapter 5.1.2, we will describe both the coprocessor and the instructions an LP running on the processor can use to communicate with it.

We also provide every processor with incoming- and outgoing-message queues, and simple ways to insert and remove messages into and from these queues. The processor execution is event-driven:

- If a new message arrives, a *message-arrival notification* occurs.
- If the scaled version of the current time equals the timestamp of an event, a *timestamp notification* occurs.

We call the code that the processor executes after a message arrival or a timestamp notification a *handler*. Every handler ends with a `DONE` instruction. Note that these “expirations” are different from the “interrupts” of a conventional processor. Consider a situation where one of our processors is executing an event. If a timestamp notification occurs before the processor is finished executing the current event, we certainly do not want the processor to stop executing the current event and handle the timestamp notification! Rather, the processor should finish executing the current event, and then execute the event corresponding to the timestamp notification. A similar situation occurs if an incoming message arrives while the processor is executing an event.

In general, we would like our processor to execute events in an order determined by the time at which their corresponding timestamp notification occurred. The processor should not begin executing one event until it has finished executing all preceding events. If an incoming message arrives, we should place it in the local event queue immediately. Our processor achieves this performance by keeping a notification FIFO queue of all message arrivals and timestamp notification.

We call the collection of the timer coprocessor, the processor, and the processor’s memory a *processing element*. We now describe each piece of a processing element

in detail, beginning with the processor itself, which we now refer to as the *processor core*.

5.1.1 The Processor Core

Our processor core uses a RISC ISA, given in Appendix A, similar to that of the MIPS processors [30], but with additional instructions for placing events into the event queue and sending and receiving messages. It has sixteen general-purpose registers, one of which is always zero, and two of which are reserved for message passing. We use several techniques to minimize the area of a processing element:

- The processor core lacks a cache.
- The processor core does not support virtual memory or exceptions.
- The processor core does not have a multiply/divide unit or a floating-point unit.
- The processor core has a 16-bit data path and variable-length (16-bit or 32-bit) instructions.
- The processor core’s memory is made from DRAM ($400 \lambda^2/\text{bit}$), instead of faster-but-less-dense SRAM ($1200\lambda^2/\text{bit}$) [44].

When the NoC boots, each processing element places the first message that it receives—referred to as its *startup message*—into a fixed address in memory. It then sets the program counter to this address and begins executing the code contained in the startup message. This message can come from another processor, or it may come from the host. The handlers and the initialization sequence must be able to inform the core that they have completed execution so that the next event can be

executed. The processor’s ISA includes a `DONE` instruction that tells the processor when it can execute the next event or handle the next incoming message.

5.1.2 The Timer Coprocessor

We designed the timer coprocessor to make scheduling events in a simulation running on the NoC as similar as possible to scheduling events in a typical DES. Our mechanism for this is simple. The timer coprocessor contains a set of *timestamp registers* and an *incrementer*. The incrementer keeps track of the scaled version of the current time: Whenever the value in a timestamp register is equal to the value of the incrementer, a timestamp notification occurs. If the processor core is not currently executing a handler, it will jump to a particular address, depending on which timestamp register produced the timestamp notification, and begin executing a handler. If the processor core is busy executing another handler, then the timer coprocessor will place a token representing the current timestamp notification in the notification FIFO queue. When the processor executes the event’s handler, it can determine the current simulated time by reading the value of the timestamp register that produced the timestamp notification. The host determines the rate of the incrementer at compile time. If, for example, the smallest unit of simulated time is 100 nanoseconds, and the time scale is equal to one, then the incrementer will increment itself once every 100 nanoseconds. Likewise, if the time scale is two, the incrementer will increment itself once every 50 nanoseconds.

Unlike the general-purpose registers in our processor, the timestamp registers and the incrementer are both 32-bit. Therefore, instructions that read the current simulation time must move the time into a *pair* of general-purpose registers.

To schedule an event, an LP running on the NoC uses a timer-coprocessor instruction to set a timestamp register to the timestamp of the event it wishes to schedule. It can also use another timer-coprocessor instruction to cancel a previously-set timer. Descriptions of these instructions follow:

SCHEDULE *id*, *timestampLow*, *timestampHigh*. Gives the timestamp register specified by *id* the value achieved by concatenating the values of the general-purpose registers specified by *timestampLow* and *timestampHigh*.

CANCEL *id*. Cancels the timestamp register specified by *id*.

READHIGH *id*, *tgt*. Puts the highest 32 bits of the timestamp register specified by *id* into the general-purpose register specified by *tgt*.

READLOW *id*, *tgt*. Puts the lowest 32 bits of the timestamp register specified by *id* into the general-purpose register specified by *tgt*.

The only complication with this scheme arises because of the size of the incrementer. Whenever this register overflows, it resets itself back to zero. Hence, the incrementer stores a scaled version of the elapsed real time of the simulation, *modulo the maximum value that the incrementer can hold*. For a 32-bit register, this maximum value is $2^{32} - 1 = 4,294,967,296$. This means that a **SCHEDULE** instruction cannot ever use a timestamp more than 4,294,967,296 simulated time units greater than the current value of the incrementer.

In Glomosim, the smallest time unit is a nanosecond [57]. In a simulation with such a smallest unit of time, the width of the incrementer implies that no event can be scheduled more than 4.295 simulated seconds into the future. For simulations of MANETs, this limit should not be a problem: Only application-layer events, which occur much less often than mac-layer or radio-layer events, for example, will be

scheduled more than a second in advance. If an LP wishes to schedule an event that will occur more than 4.295 seconds in the future, however, it can do so by using chains of events. For example, an LP could schedule an event eight seconds in the future by scheduling one event four seconds in the future; the handler for this event would simply schedule the original event an additional four seconds in the future.

Although the number of timestamp registers is limited, the software running on the processor core can still emulate an event queue with a large number of events by using the timestamp registers for the events with the earliest timestamps, and storing the events with later timestamps in a data structure such as a heap. Whenever a timestamp notification occurs, freeing up a timestamp register, the software can then use the newly-available hardware timer for the event at the top of the heap.

We stated earlier that the rate at which the incrementer changes controls the time scale of the simulation. The `RATE` instruction changes this rate. Every processing element has a 40-bit incrementer. The value of the incrementer as seen by the timer coprocessor, is a subset of 32 consecutive bits taken from this 40-bit value. (See figure.) The `INS` rate instruction can slow down the rate at which the incrementer changes by sampling sets of higher-order bits from the physical incrementer.

5.1.3 Sending and Receiving Messages

A processing element sends and receives messages via its outgoing-message buffer and incoming-message buffer, respectively. Two of the general-purpose registers in the processor core map to these buffers: The processor core places any value written to r14 into the outgoing buffer; whenever an instruction reads r15, the processor core removes the value of the head of the incoming buffer and places it into r15 (the buffers are 16 bits wide). This is similar to the method used in the

Mosaic Element [36] and in the Raw microprocessor [52]. We will discuss later in this chapter the message format that the NoC interconnect uses.

Just as every message sent between two LPs in a PDES contains an event and a timestamp, every message sent between processing elements in the NoC contains event-dependent fields (such as the duration of the simulated packet, the identity of the sender, whether the simulated packet is a protocol packet or a data packet, etc.) and a timestamp. The LP running on the processing element sending the message can calculate the message's timestamp by adding to its current simulation time. For example, when an LP in a simulation of a MANET sends a message representing the beginning of a packet transmission, the expression for the timestamp of the event is typically an addition:

$$timestamp = currentSimTime + txTurnOnDelay + PropagationDelay \quad (5.1)$$

A sender running on a processing element can use the `READ` instruction to determine *currentSimTime*, add the delays to *currentSimTime*, and send the sum in the outgoing message. (Note: We have to modify the processor core's ISA slightly to allow for 32-bit adds. For example, we could store the overflow bit of the last add in a status register, and then optionally use this as an input to the next add.) When the receiver executes its incoming-message handler, it will copy the event-dependent fields into its memory and then use the timestamp stored in the message to schedule the event.

If the NoC interconnect is congested, an instruction writing to r14 will fail—the processor will execute the instruction and store the result in r14, but it will not copy the result into the interconnect. A program can avoid this, however, by checking the outgoing-message buffer's *status register*, which contains the number of free spaces remaining in the outgoing buffer. Before a handler begins to send a message, it

should check to make sure that there is enough space in the buffer for the entire message. We designed this scheme to prevent deadlock in the NoC; we will discuss it further in Chapter 5.3.

5.2 The Interconnect

The NoC interconnect is a 2-D mesh that allows any processing element to send a message directly to any other processing element in a fast and efficient manner. Every message that travels through the interconnect consists of two parts: a *header* and a *payload*. The header contains two 16-bit words. The first word consists of two 8-bit, signed values: the horizontal and vertical differences (referred to below as the dx and the dy) between the “coordinates” in the mesh of the source and destination processing elements (a negative horizontal difference, for example, means that the destination processing element is to the west of the source). The second word in the header indicates the size, in 16-bit words, of the remainder of the message.

At every node in the mesh we place a router that is connected to the interconnect and a processing element (referred to below as the “local” processing element) at that node. The routers use dimension-order routing [15] to prevent deadlock. (We address deadlock avoidance in greater detail in Chapter 5.3.) The routers are very simple, repeating the following behavior:

1. Wait for a header to arrive from the interconnect or from the local processing element.
2. Read the first word of the header to determine whether to forward the message (the router can forward the message north, south, east, or west) or to deliver it to the local processing element. Whenever the router forwards the message

to another router, it should decrement the magnitude of the dx or the dy by one. The router forwards the message to the local processing element when dx and dy are zero.

3. Read the second word of the header to determine the length of the payload.
4. Forward the header.
5. Forward the payload.

This is similar to the mechanism described in [47] and also resembles the dynamic network described in in [51, 52].

The interconnect connects to the host in two places at the edge of the chip (Figure 5.1). The interconnect's use of dimension-order routing necessitates two connections [47], so that every processing element can communicate with the host.

Because the number—around 100—of processing elements capable of fitting on one chip is likely to be smaller than the largest network a user would like to simulate, we have designed the NoC such that a user can connect multiple chips together. This will limit the speedup of the simulation, however, because the latency to pass messages between processing elements on different chips is much higher than that to pass messages between processing elements on the same chip. When assigning nodes to processing elements in such a simulation, a user should attempt to map nodes that he believes will communicate amongst themselves most often to processing elements that share a chip or to processing elements that are on adjacent chips. This mapping process is somewhat similar to the node aggregation scheme described in [4].

5.3 Deadlock Avoidance

We state in Chapter 5.2 that the NoC avoids deadlock by employing dimension-order routing in the interconnect. For the NoC to be deadlock-free, however, each processing element must be a perfect sink [15]; that is, each processing element must *eventually* remove messages from the interconnect. This is the reason that instructions that write to r14 are non-blocking (they fail if buffer space is unavailable rather than blocking until the space becomes available). If the programmer of a NoC simulation is not sure that his simulation will not congest the interconnect, he should use an if-clause before every send instruction to check that the outgoing-message buffer is not full. If the if-clause evaluates to false (meaning the buffer is full), the program should schedule a “retry” event for a short time into the future and issue a DONE instruction. Any processing element that does this will be a perfect sink, and if all of the elements are perfect sinks, then eventually the interconnect will have low-enough congestion such that the outgoing-message buffers can copy their values into the interconnect. After the buffers have done so, they will have enough space such that the if-clauses will evaluate to true, and the simulation can continue.

If the time a processor has to wait for its outgoing-message buffer to become free is so long that an outgoing message does not reach its destination before the scaled version of the message’s timestamp, then a causality error may occur. The receiving node can detect this error and send a message to the host. The person running the simulation may then retry the simulation with a lower time scale.

5.4 Multi-Chip Simulation

We have designed the NoC such that we can assemble any number of NoCs together for large simulations. Because the processing elements address their messages based on *changes* in the x- and y-coordinates, rather than absolute coordinates, the size of a multi-chip simulation is not limited by the number of address bits.

Details of the interchip communication scheme we will use can be found in [53]. We estimate a latency of 100 ns to pass a typical (around ten 16-bit words) message between two NoCs in a multi-chip simulation.

Each NoC will use a single clock for all of its incrementers. Problems like clock skew that would be associated with a synchronous chip multiprocessor are non-issues for the NoC, however. If the incrementers of the various processing elements on a single chip change at slightly different times, the simulation will still be accurate, as long as the incrementers still change at the same *rate*. The same goes for incrementers in elements on different chips in a multi-chip simulation. Recall that our simulation will be correct if it obeys the local causality constraint. If two processing elements have incrementers that are off by a few cycles, the local causality constraint will still be satisfied as long as all incoming messages arrive before the scaled version of real time equals their timestamps. A lack of perfect synchronization between incrementers may require a slightly lower time scale (to account for the situations when the sender has an incrementer that is behind the receiver's), but the simulation should be accurate as long as the differences between incrementers do not increase during the simulation.

5.5 A Simple Example

To further illustrate how a CTBS-based simulation could run on the NoC, we provide a short example of the operations a processing element could take upon receiving a message. We assume this processing element is part of simulation of a MANET using the IEEE 802.11 DCF MAC protocol. This protocol specifies that upon receiving a data packet, the destination node should wait for a short delay, the Short Inter-Frame Spacing (SIFS), and reply with an acknowledgment packet (ACK). If a node wishes to send a data packet, it must wait for its surrounding channel to be idle for a longer period of time, the Data Inter-Frame Spacing (DIFS), plus a short, random backoff time. In this simulation, we assume DIFS is $50\mu s$, SIFS is $10\mu s$, the propagation delay between any two nodes is T_p , and the time scale is one (the scaled version of real time and the unscaled version of real time are the same). We refer to the current time as t .

We begin our example at $t = 0$ with a single event, representing the node simulated by our processing element sending a packet, in our processing element's event queue. The event's timestamp is $10\mu s$, so when $t = 10\mu s$, a timestamp notification occurs and the processor begins executing the appropriate handler. The real node would have to wait for at least DIFS before sending a packet, so the processor schedules an event with timestamp $t = 10\mu s + 50\mu s = 60\mu s$. The processor also makes appropriate changes to some variables to represent the change in the state of the MAC, which has gone from the "idle" state to the "trying to send a packet" state.

At $t = 50\mu s$, the processing element receives a message. This message has a timestamp of $52\mu s$, and represents the arrival of the beginning of a packet from another node. The processor executes its incoming-message handler, which copies

whatever information is in the incoming message (such as the simulated incoming packet’s contents, the length of the packet, and the location and radio power of the sender) into memory and schedules an event with timestamp $52\mu s$.

At $t = 52\mu s$ the timestamp notification corresponding to the simulated arrival of the packet occurs. The processor executes the appropriate handler, which cancels the previously-scheduled event that corresponded to the end of DIFS (the handler cancels this event since the simulated received packet means that the simulated channel was not idle for DIFS; the node will need to wait again, once the channel becomes idle). The handler also changes the state of the channel around the simulated node from “idle” to “busy.” We assume that the just-received message contained the duration, T_d , of the simulated packet; the processor schedules a new event for $t + T_d$. This event corresponds to the end of the simulated incoming-packet transmission.

If we assume the simulated incoming packet is 512 bytes long, and that nodes in our simulated network transmit at 2Mbps, then we can calculate T_d :

$$T_d = (512 \times 8b) / (2 \times 10^6b) \times sec = 0.0020480sec = 2048\mu s. \quad (5.2)$$

Therefore,

$$t + T_d = 2048 + 52\mu s = 2100\mu s. \quad (5.3)$$

Hence, the next timestamp notification occurs at $t = 2100\mu s$. We assume no other messages simulating packets have arrived since the beginning of the simulation of the first incoming packet; therefore, the processing element simulates the packet’s arrival without any errors from collisions. The handler checks to see what the destination of the packet is (we assume the destination was one of the fields in the received message). In this example we assume the destination is the node our

processing element is simulating, so the processor creates a message representing the transmission of the ACK, sets this message's timestamp to $t + SIFS + T_p$, and sends the message. Finally, the processor schedules events for the times at which the transmission of the ACK would begin and end. Assuming the ACK is 32 bytes, the total time for its transmission would be $128\mu s$. Therefore, the new events will have timestamps $t + SIFS$ and $t + SIFS + 128\mu s$.

This example contains a subtle optimization that can make a big difference in the performance of a CTBS. Although the actual wireless node that our processing element represents would not begin transmitting its ACK until $t = 52\mu s + T_d + SIFS$, the processing element sent the corresponding message at $t = 52\mu s + T_d$. It can send the message early because the simulation designer could *guarantee* that the node being simulated would begin transmitting at $t = 52\mu s + SIFS$, regardless of any events that might occur between $t = 52\mu s$ and $t = 52\mu s + SIFS$. In other words, he took advantage of *lookahead*. In Chapter 6 we will discuss similar optimizations and attempt to quantize the importance of lookahead to the performance of CTBS-based simulations of MANETs.

This example also demonstrates the enormous amount of time that a NoC processing element can spend doing nothing. Consider that the time spent simulating the reception of the first packet and to transmission of the ACK were *much* greater than SIFS, for instance. After simulating the reception of the packet from another node, the processing element in our example did not execute another handler until $t = 2100\mu s$. If our example had used a time scale of 20, the $2048\mu s$ to simulate the transmission of the incoming packet would have been $102.4\mu s$ of real time. If our processor core operates at 500 MHz, then this would have been enough time to execute 51,200 instructions, which is far more than we would require to perform

the state updates after simulating the beginning of an incoming packet. Thus, the processor core would have been idle for most of those 51,200 cycles. We will discuss better ways (such as speculation and post-computation) to use these extra cycles in Chapter 6.

Chapter 6

Evaluation of Scaling Properties

In this chapter we examine how the performance of CTBS-based simulation of MANETs scales with increasing network sizes. We present results for simulations of the NoC running CTBS-based simulations of MANETs ranging in size from 100 to 900 nodes. Our preliminary results indicate that a CTBS-based simulator can run on the NoC at 22 times faster than real time, regardless of the network size. In this chapter we explain how we describe our simulation setup, we describe the networks that we simulated simulating, we discuss ways to optimize typical DES code for performance in a CTBS-based simulator, and we present our final scaling results. Finally, we use these results to evaluate how well our technique works for the simulation of different kinds of networks.

6.1 Simulation Setup

None of the results in this thesis are from a detailed, cycle-accurate simulator of the NoC. Rather than designing such a simulator, we modified Glomosim to generate logs containing, for every simulated node, every event (with its timestamp) executed

in the simulation of that node. The log file for a particular node corresponds to the workload of the processing element that would simulate this node in a CTBS-based simulation running on the NoC. We created a script that used these logs, along with worst-case estimates of the time to execute the events' handlers and worst-case estimates of the time for messages to move through the interconnect, to estimate the largest time scale we could safely use without generating any causality errors.

We obtained our worst-case instruction counts by writing handlers in C, compiling them for the MIPS ISA, and then running them in a MIPS simulator to obtain dynamic instruction counts. We assumed 500 MHz processing cores and a worst-case delay of 100 ns to send a message from one node to another. This delay is great enough to include the time to send a message between chips.

6.2 Network Scenarios

We simulated networks ranging from 100 to 900 nodes using the IEEE 802.11 MAC protocol and the Ad-hoc On-Demand Distance Vector (AODV) routing protocol [43]. A fraction of the total nodes were constant-bit-rate sources, a fraction were sinks, and the rest forwarded packets between the sources and sinks. Although we chose to place the nodes such that a given node could typically only communicate clearly with four to eight other nodes, the simulated nodes in Glomosim typically copied their messages to an average of sixteen other nodes (see Figure 6.1). These “extra” nodes are nodes that could not receive signals clearly, but rather as noise.

We simulated simulating networks with static nodes and networks with mobile nodes. For the latter networks, we used standard random-waypoint models of node mobility. The Glomosim simulations that created our logs used the two-ray pathloss model, 2Mbps-bandwidth channels, and the simple, “no noise” radio model. The

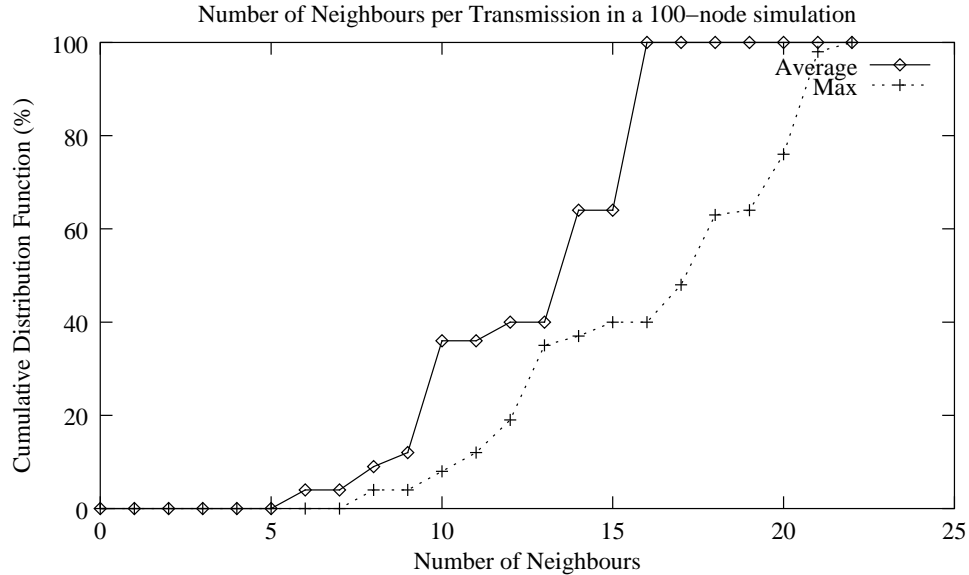


Figure 6.1: Number of neighbours for a 100-node simulation.

formula for the two-ray pathloss between two nodes is

$$pathloss = -(tx + rx) + 20 \times \log_{10}(4\pi \times d/\lambda), \quad (6.1)$$

where tx and rx are the transmitter's and receiver's antenna gains (in dB), respectively, d is the distance (in meters) between the nodes, and λ is the wavelength (in meters) of the radio signals.

6.3 Optimizations

While simulating the performance of the NoC when simulating the networks described above, we performed several optimizations designed to increase our time scale. We began by assuming that our simulator would execute all events in exactly the same manner as Glomosim does. We then optimized our (simulated) simulator until the only factor limiting its performance was the physical delay to send messages through the NoC's interconnect. The next several paragraphs describe how

we refined our unoptimized simulator until we were limited by only this physical delay.

The “critical path” for our unoptimized simulator quickly became apparent. The time scale was limited by the time between two events: an event indicating that the channel near a node has become idle and an event indicating the same node sending a new packet. Our goal was to move as much computation as possible off of this path. The simulated time between these two events depends on whether the new packet is a data or acknowledgment packet. For a data packet, the length of the critical path is DIFS, plus some random backoff time, plus the transmitter-turn-on time and the propagation delay. For an acknowledgment packet, the length of the critical path is SIFS plus the propagation delay. In the rest of this section, we provide several examples of our optimizations, we discuss how we will perform radio calculations without having a multiply/divide unit in any of the processing elements, and we discuss our final “critical path” and calculate the highest time scale that it allows us.

6.3.1 Moving Computation

Our first optimization increases the speed of the critical path for acknowledgment messages. In our unoptimized simulator, a logical process does not examine a simulated incoming packet until it can be sure that it will simulate that packet arriving without any errors. If the LP does simulate the packet arriving without any errors, it must examine the packet’s fields (source, destination, type, etc.) and determine what the node it is simulating would do. If the packet is a unicast data packet meant for the node the LP is simulating, then the LP must assemble a message representing the acknowledgment packet that the node would transmit. Checking

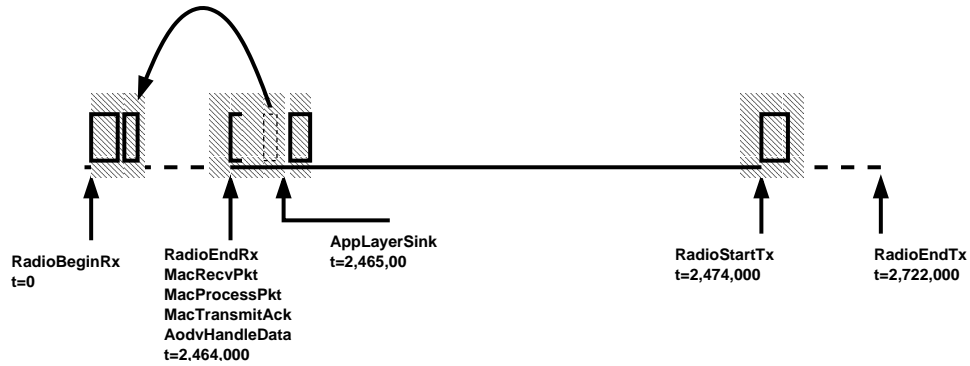


Figure 6.2: Moving computation for an ack. The shaded boxes represent computation done for the events listed below. The optimization moves the computation that analyzes the incoming message and potentially creates an ack. Times are in simulated nanoseconds. to see whether or not the packet will require an ack, and then building the message requiring the ack, does not require an enormous amount of computation, but is still demanding enough to slow down the time scale if we do not remove it from the critical path.

The optimization, therefore, is for the LP to immediately examine the simulated incoming packet and potentially create an ack *as soon as the message containing the packet arrives*, rather than waiting until the LP is sure that it will simulate the packet arriving without any errors. When the LP executes the event representing the end of the transmission of the packet, it can send the ack immediately, if appropriate. The downside of this optimization, of course, is that the LP will perform wasted calculation every time it examines a packet that will later be involved in a simulated collision. However, the LP is normally idle during the time when it will perform this computation, so this optimization will never slow down the simulation. Figure 6.2 shows this optimization applied to a timeline of events from an actual simulation. By moving some of the computation off of the critical path, we can decrease the time scale.

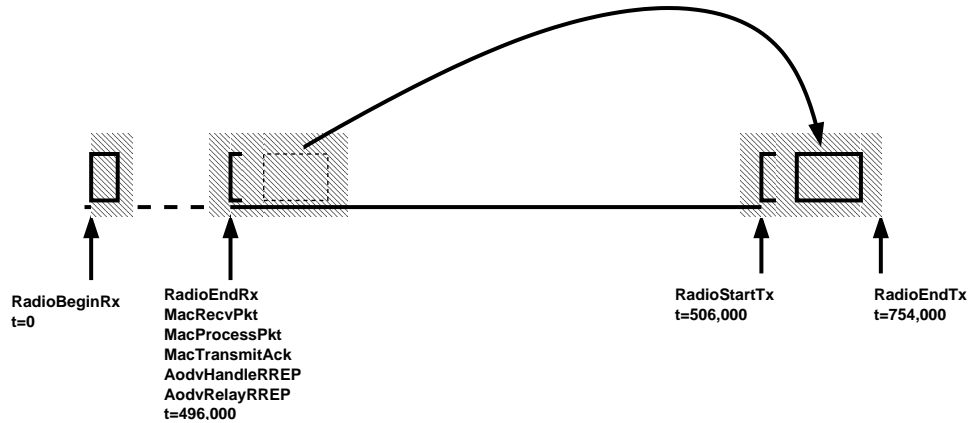


Figure 6.3: Moving computation for forwarding a route-reply packet. The shaded boxes represent computation done for the events listed below. The optimization moves all of the non-ack computation. Times are in simulated nanoseconds.

Our second optimization decreases the computation on our critical path by moving *forward* operations on the AODV routing table. AODV control packets can be either unicast or broadcast packets. The IEEE 802.11b protocol declares that unicast packets require replies, via acks, from their destinations, but that broadcast packets do not. Therefore, the method we choose to optimize the simulation of the reception of these packets varies depending on whether the packets are unicast or broadcast.

Optimizing the actions that an LP performs after simulating the reception of a unicast AODV packet is simple: If a logical process simulates the reception of a unicast AODV packet, such as a route reply (RREP) packet, it will immediately simulate the transmission of the corresponding ack. After sending to its neighbors messages representing this ack, the LP will be idle for an amount of time corresponding to the simulated transmission of the ack. The LP can perform any necessary updates to the AODV routing table during this time. Figure 6.3 shows this optimization.

We can use the same precomputation technique to optimize the simulation of the reception of broadcast AODV packets that we used to optimize the creation of acks: An LP checking a simulated incoming packet to see if it requires an ack can also determine whether the packet is a broadcast AODV packet. For example, if a node receives a route-request (RREQ) packet, it will perform several lookups in its routing table and, depending on the results of these lookups, it may immediately relay the request to all of its neighbors (after waiting for DIFS plus a random backoff time, of course). Likewise, the LP simulating this node can perform these checks and possibly assemble a copy of the RREQ packet (the LP must slightly modify the original RREQ packet before relaying it) to relay before it is sure that it will simulate the errorless reception of the packet. The LP can perform any further calculations, such as updating its routing tables, once it has begun simulating the transmission of the RREQ. As before, if the incoming broadcast AODV packet is later involved in a collision, then the speculative precomputation will be wasted.

The rest of our optimizations followed the trend shown above by moving computation from the critical path in two ways:

1. Perform whatever computation that may influence *the next outgoing message* speculatively.
2. Perform whatever computation that does not influence the next outgoing message during the time when the LP is simulating the transmission of the packet that the next outgoing message represents.

We have designed the simulator that will run on the NoC this way because of the great deal of (scaled) time that it uses to simulate the transmission of messages. The unoptimized simulator was idle for these times.

To give a quantitative look at how large these idle times can be, we provide Table 6.3.1, which contains the timestamps of several events that occurred during one of our (simulated) simulations. This table demonstrates the ample time ($496.0\mu s$ for data packets, $248.0\mu s$ for acks) that exists between the events representing the beginning and ending of transmissions. Our optimizations move as much calculation as possible into the time between these events. Table 6.3.1 also shows the number of cycles between events. These cycle counts are for a timescale of 22 and 500 MHz processor cores.

Table 6.1: Events and their timestamps from a simulation of a MANET. Times are in simulated μs .

Event	Time	Time since last event	Cycles until next event
IncomingTransmisBegin	1346784.167	n/a	11273
IncomingTransmisEnd	1347280.167	496.0	227
TransmitAckBegin	1347290.167	10.0	5636
TransmitAckEnd	1347538.167	248.0	0
StartDifs	1347538.167	0	1022
EndDifs	1347583.167	45.0	0
StartBackoff	1347583.167	0	772
EndBackoff	1347923.167	34.0	113
TransmitDataBegin	1347928.167	5.0	n/a

6.3.2 Performing Radio Calculations

Recall from Chapter 5 that none of the processing elements on the NoC have floating points units or integer multiply/divide units. While this does not prevent the processing elements from simulating any MAC or routing-layer protocols, it does prevent them from performing any radio-layer computations efficiently. For example, calculating the two-ray pathloss between two nodes in Glomosim involves performing several multiplies and divides, a square root, and a log; all of these calculations involve floating-point numbers. Glomosim performs a connectivity test for every pair of nodes (within a partition) for every simulated packet sent. We need a method with which an LP can know to which other LPs it should send messages.

We compensate for the lack of floating-point hardware in our processing elements by precomputing all radio information on the host before the simulation begins. The host can include in the startup message for every processing element a *connectivity table* of other processing elements to which the element receiving the startup message should copy all of its messages. Naturally, the connectivity of a network with mobile nodes will change with time; therefore, the connectivity table should contain different lists for different times. Software running on a processing element should always use the current simulated time to lookup the proper processing elements to which to copy an outgoing message.

This table should also include the power levels of simulated incoming packets. Whenever an LP simulates the reception of an incoming packet, it needs to compare the power of that packet against several values, such as the power of other incoming packets and the threshold of its antenna. In our simulator, the LP can look up the power of an incoming packet at a given time from a given simulated node in its connectivity table. This table cannot offer exact values, however, for all

times, because some nodes in the simulation may be moving at the times when they transmit packets. Therefore, the connectivity table contains several power values for different times during the periods when a mobile node is moving; for a given simulated incoming packet, the LP should pick the power value with the time closest to the current simulated time.

Some simulations may require connectivity tables that are too large to fit in the processing elements' memories. For example, a very long simulation, or a simulation with highly-mobile nodes, might require each node to have many entries in its table. In such a situation, the host could periodically update the tables of every processing element. The host could send update messages during the simulation, or it could pause the simulation (by sending every processing element a "pause" message) and then send the update messages. Such a scheme would also enable data-dependent mobility.

6.3.3 Our Final Critical Path

After our optimizations, the factor that ultimately limited our time scale was the physical time necessary to transmit messages through the NoC interconnect. Since we have to choose our time scale such that *every* message will arrive on-time, we must account for messages that travel from one chip to another. We assume that, in the worst-case, the total delay for such a message is $100ns$, that a node will have to send 16 messages for every transmission, and that each packet will be no bigger than 10 words. We assume a processing element will need one cycle per word to transmit a message. Because the message transfers are pipelined through the interconnect, we choose our time scale such that the first word of the last message sent will arrive on-time.

In simulated time, the messages have the transmitter-turn-on delay ($5\mu s$) plus the propagation delay to reach their destinations. Because we have mobile nodes, however, the propagation delay may approach zero for some sender-receiver pairs. Therefore, we have the following constraint for our time scale:

$$s \times \textit{physical_delay} < \textit{simulated_delay} \quad (6.2)$$

$$s \times (100 \textit{ ns} + (15 \times 10 + 1) \textit{ cycles} \times 2 \textit{ ns/cycle}) < 5\mu s \quad (6.3)$$

$$s < 12.438 \quad (6.4)$$

We can, however, perform one final optimization that will allow us to almost double our time scale. Consider that any message we send in our simulation represents the beginning of a wireless transmission. No LP that receives a message will therefore itself send a message until the scaled version of the time (for example, $496\mu s$ or $248.0\mu s$) the wireless transmission would take in the real network. For example, say that an LP is simulating a node in its backoff period, with a packet ready to transmit. If this node started to receive a packet from another node, it would not transmit its own packet until it has finished receiving the packet.

Our final optimization is to break up each message sent by an LP into *two* messages sent in the NoC. The first message is only four words long: It contains the two header words and a two-word timestamp. Whenever an LP (processing element) receives such a message, it can schedule an event corresponding to the beginning of a wireless transmission with the timestamp included in the message. That the LP does not know which LP sent the message or what kind of packet the message represents does not matter. When the LP executes the event corresponding to the first message, it will then wait for a second message, which will contain the rest of the information pertaining to the event. This optimization changes our constraint

for s :

$$s \times \textit{physical_delay} < \textit{simulated_delay} \quad (6.5)$$

$$s \times (100 \textit{ ns} + (15 \times 4 + 1) \textit{ cycles} \times 2 \textit{ ns/cycle}) < 5\mu\textit{s} \quad (6.6)$$

$$s < 22.523 \quad (6.7)$$

This optimization is safe even if an LP receives several of the four-word messages before receiving any of the corresponding longer messages. It will execute the event corresponding to the event with the earliest timestamp first. If it receives the longer messages corresponding to later events before it receives the longer message corresponding to the first event, it can simply store the contents of these messages in memory and then access them when it executes their events.

6.4 Results

Finally, we present results for simulations of simulations of large MANETs. Because CTBS-based simulations of MANETs scale so well, however, these results are fairly uninteresting. Figure 6.4 shows that the time scale does not vary with the size of the simulated network. Adjusting factors such as the average size of the data packets and the frequency with which nodes send packets does not change the time scale, either (as long as the packets are still large enough such that the simulated time to transmit or receive a data packet is long enough to overlap the time spent doing computation).

Changing the transmitter-turn-on time, however, does affect the time scale of our simulations. Recall our bound on s :

$$s < \textit{transmitter_turn_on_time}/(222 \textit{ ns}) \quad (6.8)$$

The value of s for various turn-on-times is shown in Figure 6.5.

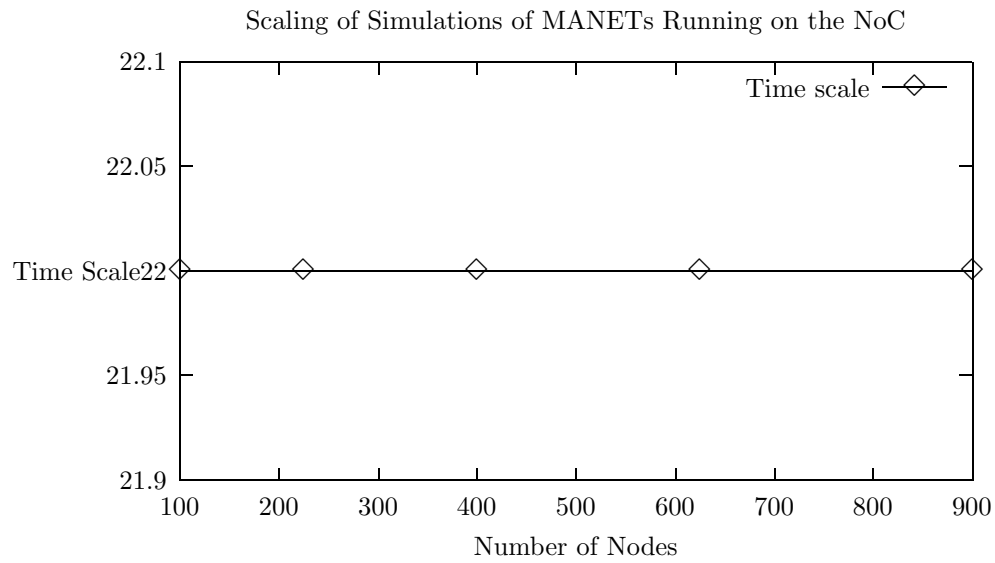


Figure 6.4: The time scales for simulations of CTBS-based simulations, running on the NoC, for MANETs of various sizes.

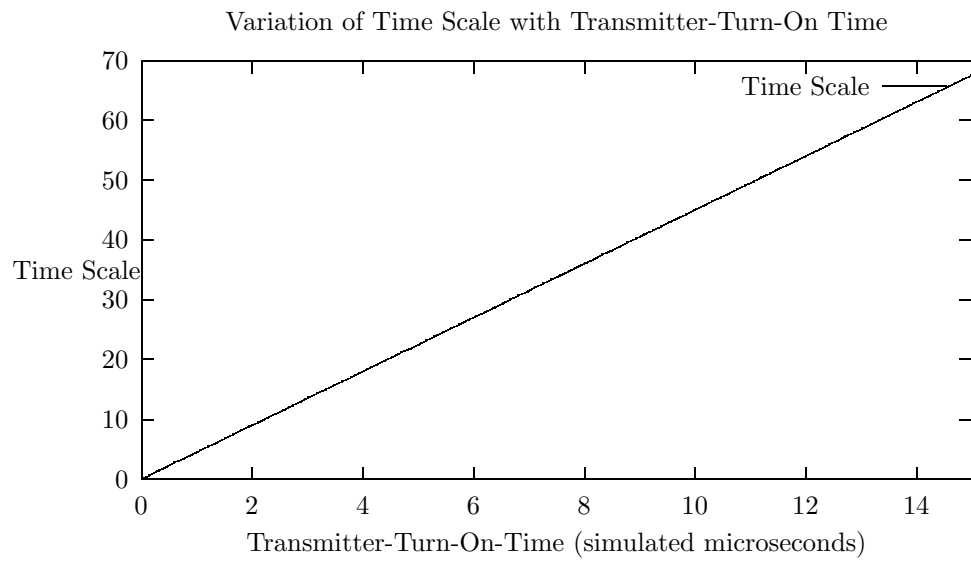


Figure 6.5: The time scales for simulations of CTBS-based simulations, running on the NoC, for various transmitter-turn-on times.

6.5 Pros and Cons

We will now discuss some of the advantages and disadvantages of performing research on networks using the NoC versus using other simulation techniques. We discuss the utility of CTBS-based simulations of MANETs and in general.

6.5.1 Disadvantages

The most obvious disadvantage of our system is its cost: If one NoC chip contains 100 processors, then simulating a 10,000-node network would require 100 chips. Assembling such a test bed would probably be much more expensive than purchasing a high-end DSM machine to run a PDES using a more conventional synchronization mechanism. We will address the cost problem in Chapter 7.

Another disadvantage to using a CTBS-based simulation running on the NoC is the amount of effort required to optimize such a simulation. The transformations we outlined earlier in this chapter were not enormously complicated, but did require intimate knowledge of how our simulator behaves. The amount of work required to obtain this knowledge and to make the optimizations we outlined may be too much to ask of users who are not skilled in parallel simulation.

The NoC's lack of any floating-point hardware requires it to use pre-computed information for its radio calculations. Our ability to precompute these values depends on our being able to accurately predict the movement of all of the nodes in the simulated network. Although we can do this easily for networks using the random-waypoint mobility model, we could not do this for networks in which the movement of the nodes depends on factors that we cannot predict before the simulation begins. For example, a researcher might wish to study a protocol he has designing in which nodes base their movement patterns on the rate of collisions at

different locations in the simulated environment. We have not yet studied how to simulate such a network using our simulator; one possible method would be to use frequent connectivity-table updates from the host.

Finally, for many simulations, many of the NoC processing elements may be idle for long periods of time. This is not necessarily a *disadvantage*, but part of our future work (discussed in Chapter 7) is to determine something useful for these nodes to do while they are idle.

6.5.2 Advantages

The most obvious advantage of the simulator we have described is the enormous speedup and perfect scaling that it provides for simulations of large MANETs. Because we can gluelessly assemble any number of NoC chips, we can simulate networks of arbitrary size with perfect scaling, as long as the connectivity of the nodes remains the same (higher connectivity would increase the number of copies of each message, which would decrease s).

One of the reasons that our simulator can simulate such large networks so quickly is that the hardware we have designed is very appropriate for our application domain. The simulation of large-scale MANETs is an inherently *parallel* task: None of the LPs share any variables, and they communicate only by message-passing. We can take advantage of the lack of shared variables by designing the NoC's memory system without any memory coherence between processing elements.

Moreover, because each processing element will represent only a single node in the simulated network, its memory does not have to be very large. Although we said in Chapter 5 that our processing elements lack caches, their memories are so small that we can think of our processing elements as having *nothing but* caches:

No memory access ever takes more than one cycle. Although Glomosim avoids context-switching overheads by simulating a large number of nodes with a single logical process [57], it still may suffer from poor cache behavior. A logical process that simulates many nodes may have very little “locality” between events in its event queue; the LP may rarely consecutively execute two events that affect the same node. This random distribution of affected nodes will probably lead to many cache misses, since the relevant data structures for all simulated nodes will not all fit simultaneously in a typical cache.

A second advantage, which we discuss further in the next chapter, is that we can use the NoC’s processing elements as the basis for a test bed of actual sensor-network nodes. Because these real nodes would execute the same protocol code as the simulated nodes on the NoC, we would be able to easily judge how accurate our simulated radio models are.

Chapter 7

Summary

We have presented the architecture of a network-on-a-chip. We described the NoC's components in detail, showed how to map various network protocols and topologies to it, and argued that it can correctly model various network scenarios. Finally, we gave preliminary results that suggest the NoC will be able to simulate networks faster than real-time.

In this chapter, we discuss two implications of our work: the possibility of new protocols that use in-situ simulation to modify themselves on the fly, and the possibility of creating sensor networks with nodes based on the NoC processing elements. We also discuss how to deal with the cost of assembling a large test bed of NoC chips. Finally, we discuss our future work.

7.1 New Protocols

The ability to simulate large networks faster than real time could make possible entirely new protocols for mobile ad hoc wireless networks. For example, consider a network in which every node contains a collection of NoC chips. If a node decides

to move (for example, if it is part of a battlefield scenario and it needs to avoid destruction), it could use its local NoC to quickly simulate how the behavior of the network will change when the node is in several possible new positions, and then move to the position which will the simulations predict will result the best performance.

7.2 Sensor Networks

Our Asynchronous VLSI group at Cornell is currently attempting to build nodes for sensor networks from the processing elements in the NoC. The nodes will be able to run exactly the same code as the processing elements the simulator do (albeit at a slower rate, since the physical network cannot run faster than real time), except they will have real radios instead of simulated radio layers. Such a test bed would prove valuable for many reasons. For example, researchers could validate the results of NoC simulations by comparing the performance of a real network with the predicted performance from the NoC. Since the only difference between the simulation and the physical system would be the radio layer, this would allow researchers an easy way to test the validity of various simulated radio models.

7.3 The NoC Test bed

Because the simulator we have described in this thesis relies on custom hardware, we cannot assume that it will be affordable to all researchers wishing to study large-scale MANETs. Therefore, we envision that CTBS-based simulation on the NoC will be most useful in a service similar to that of Netbed [56]: If one university purchased enough NoC chips to create a test bed capable of simulating 100,000

nodes, it could create an automated web interface that would allow researchers from other universities to schedule time to use the test bed remotely. Users could control the test bed by simply logging into the host. Since simulations running on the NoC should complete very quickly, sharing the NoC among many researchers should be possible.

7.4 Future Work

Our work in the near future focuses on laying out and fabricating a single processing element. This will allow us to run test code and verify all of the timing assumptions we made in our simulations.

Our long-term goals are to evaluate more thoroughly what is the best way to simulate large-scale MANETs. To do this, we will evaluate many different synchronization protocols on parallel architectures similar to the NoC. We will also explore the possibility of adding integer-multiply/divide and floating-point units in the NoC processing elements.

Finally, we will examine different uses for the long idle times that the processing elements in the NoC experience during some simulations. There may exist efficient methods to bypass some of these idle times.

Appendix A

The NoC Processor ISA

A.1 Introduction

This document contains an architectural description of the Network on a Chip (NoC) processing element.

A.2 State of the Processor

The processor state is composed of the following:

- A set of 16-bit general-purpose registers, `reg[0] ... reg[15]`. `reg[0]` is always zero. `reg[r14]` maps to the outgoing-message queue. `reg[r15]` maps to the incoming-message queue.
- A memory with two banks, `mem1` and `mem2`. Each word in the memory is 16 bits.
- A program counter, `pc`.

- A status register, `carry`, which contains the carry-out of the last arithmetic operation.
- A 32-bit incrementer, `incrementer`.
- A set of 32-bit event registers, `timestamp register0` ... `timestamp register7`. Each event register must be “turned on.” The 8-bit register `enable` keeps track of whether these registers contain actual event time stamps.
- A set of registers, `hand[0]` ... `hand[7]` , which indicate the addresses of the handlers for the eight event registers, and `hand[8]` , which indicates the address of the handler for incoming messages.
- A status register, `buf`, which contains the number of available spaces in the outgoing-message buffer.

A.3 Description of Instructions

Here are the details of each instruction. Keep in mind that, if the target of instruction is `reg[0]`, the value of `reg[0]` will not change. If the target of an instruction is `reg[r14]`, then the processor will actually just write to the outgoing-message buffer. If one of the sources of an instruction is `reg[r15]`, the processor will read the value from the incoming-message buffer.

A.3.1 ALU Instructions (Reg, Reg)

- ADD `dst src1 src2`.

$$\text{reg}[\text{dst}] := \text{reg}[\text{src1}] + \text{reg}[\text{src2}].$$

$$\text{carry} := \text{carryout}.$$

- SUB dst src1 src2.
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] - \text{reg}[\text{src2}].$
carry := carryout.
- OR dst src1 src2.
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \text{ or } \text{reg}[\text{src2}].$
- AND dst src1 src2.
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \text{ and } \text{reg}[\text{src2}].$
- XOR dst src1 src2.
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \text{ xor } \text{reg}[\text{src2}].$
- NOR dst src1 src2.
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \text{ nor } \text{reg}[\text{src2}].$
- SLLV dst src1 src2. (shift left logical variable)
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \ll \text{reg}[\text{src2}].$
- SRLV dst src1 src2. (shift right logical variable)
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \gg \text{reg}[\text{src2}].$
- SRAV dst src1 src2. (shift right arithmetic variable)
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \ggg \text{reg}[\text{src2}].$

A.3.2 Carry Instructions

- ADDC dst src1 src. (add with carry)
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] + \text{reg}[\text{src2}] + \text{carry}.$
carry := carryout.

- SUBC *dst src1 src2*. (subtract with carry)
 $\text{reg}[\text{dst}] := \text{reg}[\text{src1}] - \text{reg}[\text{src2}] + \text{carry}.$
 $\text{carry} := \text{carryout}.$

A.3.3 Short Immediate Instructions

- LSI *dst imm*. (load short immediate)
 $\text{reg}[\text{dst}] := \text{imm}.$
- INC *dst imm*. (increment)
 $\text{reg}[\text{dst}] := \text{reg}[\text{dst}] + \text{imm}.$

A.3.4 Event-scheduling Instructions

- SCHEDULE *id, hi, lo*.
 $\text{timestampregisterid}[31 : 16] := \text{reg}[\text{hi}].$
 $\text{timestampregisterid}[15 : 0] := \text{reg}[\text{lo}].$
 $\text{enable}[\text{id}] := 1.$
- CANCEL *id*.
 $\text{enable}[\text{id}] := 0.$
- READHI *id, hi*.
 $\text{reg}[\text{hi}] := \text{timestampregisterid}[31 : 16].$
- READLO *id, lo*.
 $\text{reg}[\text{lo}] := \text{timestampregisterid}[15 : 0].$

A.3.5 Bit-field Instructions

- BFS *dst src1 hi lo*. (Bit-field set)

$\text{reg}[\text{dst}][hi : lo] := \text{reg}[\text{src1}]$.

The 16-bit immediate indicates the range of $\text{reg}[\text{dst}]$ that should be set to the value contained in $\text{reg}[\text{src1}]$. This instruction should be useful for constructing messages.

- BFR *dst src1 hi lo*. (Bit-field read)

$\text{reg}[\text{src1}] := \text{reg}[\text{dst}][hi : lo]$.

The 16-bit immediate indicates the range of $\text{reg}[\text{dst}]$ that should be placed in $\text{reg}[\text{src1}]$. This instruction should be useful for reading messages.

A.3.6 ALU Instructions (*reg, imm*)

- ADDI *dst src1 imm16*

$\text{reg}[\text{dst}] := \text{reg}[\text{src1}] + \text{imm16}$.

- ANDI *dst src1 imm16*

$\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \textit{ and } \text{imm16}$.

- ORI *dst src1 imm16*

$\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \textit{ or } \text{imm16}$.

- XORI *dst src1 imm16*

$\text{reg}[\text{dst}] := \text{reg}[\text{src1}] \textit{ xor } \text{imm16}$.

A.3.7 Memory Instructions

- `LOAD0 dst src1 imm16`
 $\text{reg}[\text{dst}] := \text{mem0}[\text{reg}[\text{src1}] + \text{imm16}]$.
- `STORE0 dst src1 imm16`
 $\text{mem0}[\text{reg}[\text{src1}] + \text{imm16}] := \text{reg}[\text{dst}]$.
- `LOAD1 dst src1 imm16`
 $\text{reg}[\text{dst}] := \text{mem1}[\text{reg}[\text{src1}] + \text{imm16}]$.
- `STORE1 dst src1 imm16`
 $\text{mem1}[\text{reg}[\text{src1}] + \text{imm16}] := \text{reg}[\text{dst}]$.

A.3.8 Unconditional Branches

- `JALR dst src1`
 $\text{pc} := \text{reg}[\text{src1}], \text{regdst} := \text{pc} + 2$.
 This is the standard jump instruction. If you do not wish to link, let $\text{dst} := 0$.
- `JAL dst imm16`
 $\text{pc} := \text{imm16}, \text{reg}[\text{dst}] := \text{pc} + 2$.
 Standard jump instruction. Once again, if you do not wish to link, let $\text{dst} := 0$.

A.3.9 Conditional Branches

- `BEQ src2 src1 imm16` (branch equal)
 $(\text{reg}[\text{src1}] = \text{reg}[\text{src2}]) ? \text{pc} := \text{imm16} : \text{pc} := \text{pc} + 1$.
- `BNE src2 src1 imm16` (branch not-equal)
 $(\text{src1} \neq \text{src2}) ? \text{pc} := \text{imm16} : \text{pc} := \text{pc} + 1$.

- BGEZ `src1 imm16` (branch greater-than-or-equal-to zero)
 $(src1 \geq 0) ? pc := imm16 : pc := pc + 1.$
- BLTZ `src1 imm16` (branch less-than zero) $(src1 < 0) ? pc := imm16 : pc := pc + 1.$

A.3.10 Miscellaneous

- SETADDR `src imm16` (set address)

`hand[src] := imm16.`

`NUMFREE dst`

`reg[dst] := buf.`

Used to determine the number of free spaces in the outgoing-message buffer.

A.4 Opcode Encodings

Because the NoC uses 16-bit processor cores, the instruction encoding is somewhat complicated. There are several different formats for instructions, and some instructions use two words.

Any instruction with an opcode = 1111 uses a second 16-bit word. Table A.1 and Table A.2 list the encoding for every instruction.

A.5 Comments

- There was no space for shift-by-immediate instructions.
- Main memory consists of two “banks.” Unlike traditional banked memory, however, in which one bank holds even address and one holds odd (to allow

Table A.1: Single-word instructions.

Instruction	15..12	11..8	7..4	3..0
	“op”	“F0”	“F1”	“F2”
ADD	0000	dst	src1	src2
SUB	0001	dst	src1	src2
ADDC	0010	dst	src1	src2
SUBC	0011	dst	src1	src2
OR	0100	dst	src1	src2
AND	0101	dst	src1	src2
XOR	0110	dst	src1	src2
NOR	0111	dst	src1	src2
SLLV	1000	dst	src1	src2
SRLV	1001	dst	src1	src2
SRAV	1010	dst	src1	src2
SCHEDULE	1011	dst	src1	src2
JALR	1110	dst	src	0000
CANCEL	1110	—	id	0001
WAIT	1110	—	—	0010
NUMFREE	1110	dst	—	0101
READHI	1110	dst	hi	0110
READLO	1110	dst	lo	0111

Table A.2: Double-word instructions.

Instruction	15..12	11..8	7..4	3..0
	“op”	“F0”	“F1”	“F2”
ADDI	1111	dst	src1	0000
ADDIC	1111	dst	src1	0001
ORI	1111	dst	src1	0010
ANDI	1111	dst	src1	0011
XORI	1111	dst	src1	0100
NORI	1111	dst	src1	0101
BFS	1111	dst	src1	0110
BFR	1111	dst	src1	0111
LOAD0	1111	dst	src1	1000
STORE0	1111	src2	src1	1001
LOAD1	1111	dst	src1	1010
STORE1	1111	src2	src1	1011
BEQ	1111	src2	src1	1100
BNE	1111	src2	src1	1101
BGEZ	1111	src1	0000	1111
BLTZ	1111	src1	0001	1111
SETADDR	1111	src1	0010	1111
JAL	1111	dst	0011	1111

parallelism in memory operations to consecutive addresses), in the NoC, one bank holds the first half of the memory, and one holds the second. This allows a convention in which the processor core uses the first bank and the message coprocessor uses the second.

- Register zero is always zero.
- The first instruction in a handler cannot be `wait`.

Bibliography

- [1] Garcia Luna Aceves. Loop Free Routing Using Diffusing Computations. *ACM Transactions on Networking*. 1(1):130-141, 1993.
- [2] Rajive L. Bagrodia. Perils and Pitfalls of Parallel Discrete-Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference*, 1996.
- [3] 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.
- [4] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, Ken Tang, Rajive Bagrodia, Mario Gerla. GloMoSim: A Scalable Network Simulation Environment. *UCLA Computer Science Department Technical Report 990027*, May 1999.
- [5] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildiz. Efficient parallel simulation of large-scale PCS networks. *Transactions of the Society for Computer Simulation International*, 16(3): 113-125, 1999.
- [6] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildiz. Exploiting model independence for parallel PCS network simulation. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pages 166-173, 1999.
- [7] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildiz. Performance analysis of a parallel PCS network simulation. *Proceedings of the 6th International Conference on High Performance Computing (HiPC'99)*, 1999.
- [8] Bryant, R.E. Simulation of packet communications architecture computer systems. MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977.
- [9] C. D. Carothers, R. M. Fujimoto, Y. B. Lin, and P. England. Distributed simulation of large-scale PCS networks. *Proceedings of International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2-6, 1994.
- [10] C. D. Carothers, R. M. Fujimoto, and Y. B. Lin. A case study in simulating PCS networks using time warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS'95)*, pages 87-94, 1995.

- [11] C. D. Carothers, R. M. Fujimoto, and Y. B. Lin. Simulating population dependent PCS network models using time warp. *Proceedings of the 1995 Winter Simulation Conference*, pages 555-562, 1995.
- [12] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM* 24, 11 (November 1981), 198-205.
- [13] Mainak Chaudhuri. `mainak@csl.cornell.edu`. Personal communication.
- [14] William J. Dally et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, pages 23-39, April 1992.
- [15] William J. Dally and Charles L. Seitz. Deadlock-free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, Vol. 36, No. 5, pp. 547-553, May 1987.
- [16] Samir R. Das, Charles E. Perkins, Elizabeth M. Royer and Mahesh K. Marina. Performance Comparison of Two On-demand Routing Protocols for Ad hoc Networks. *IEEE Personal Communications Magazine special issue on Ad hoc Networking*, February 2001, p. 16-28.
- [17] The DaSSF Homepage.
<http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/>.
- [18] 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.
- [19] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pp. 263-270. Seattle, Washington, USA, ACM. August, 1999.
- [20] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10): 30-53, October 1990.
- [21] A. Gafni. A rollback mechanism for optimistic distributed simulation systems. *Proc. of the SCS Multiconference on Distributed Simulation*, Jul, 1988, Vol. 19, No. 3, pp. 61.
- [22] Kouros Gharachorloo, Madhu Sharma, Simon Steely, Stephen Van Doren: Architecture and design of AlphaServer GS320. *ASPLOS 2000*: 13- 24.
- [23] A. G. Greenberg, B. D. Lubachevsky, D. M. Nicol, and P. E. Wright. Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communications. *Proceedings of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 45-47, 1995.

- [24] Piyush Gupta, Robert Gray, and P. R. Kumar. An Experimental Scaling Law for Ad Hoc Networks. May 16, 2001.
- [25] Z.J. Haas and M.R. Pearlman, The Zone Routing Protocol (ZRP) for Ad Hoc Networks. *Internet Draft, draft-ietf-manet-zone-zrp-02.txt*, June 1999.
- [26] Zygmunt J. Haas and Marc R. Pearlman. The Performance of Query Control Schemes for the Zone Routing Protocol. *IEEE/ACM Transactions on Networking*, August 2001, pp. 427-438.
- [27] Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, August 2000, Boston, Massachusetts.
- [28] David Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, **7**(3):404-425, July 1985.
- [29] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, edited by Tomasz Imielinski and Hank Korth, Chapter 5, pages 153-181, Kluwer Academic Publishers, 1996.
- [30] Gerry Kane, Joe Heinrich, Joseph Heinrich. Mips Risc Architecture. Prentice Hall. 1991.
- [31] Barry M. Leiner, Robert J. Ruth, and Ambatipudi R. Sastry. Goals and Challenges of the DARPA GloMo Program. *IEEE Personal Communications*, **3**(6):34-43, December 1996.
- [32] M. Liljenstam and R. Ayani. A model for parallel simulation of mobile telecommunication systems. *Proceedings of 4th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. Pages 168-173, 1996.
- [33] M. Liljenstam and R. Ayani. Partitioning PCS for parallel simulation. *Proceedings of the 5th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Pages 38-43, 1997.
- [34] Y. B. Lin and P. Fishwick. Asynchronous parallel discrete event simulation. *IEEE Transactions on Systems, Man, and Cybernetics*, **26**(4), 1996.
- [35] J. Liu, L. F. Perrone, D. M. Nicol, M. Liljenstam, C. Elliott, and D. Pearson. Simulation modeling of large-scale ad-hoc sensor networks. *European Simulation Interoperability Workshop*, 2001.
- [36] Lutz, Chris; Rabin, Steve; Seitz, Charles L. and Speck, Don (1983) Design of the Mosaic Element. <http://resolver.library.caltech.edu/caltechCSTR:1983.5093-tr-83>

- [37] B. A. Malloy and A. T. Montroy. A parallel distributed simulation of a large-scale PCS network: keeping secrets. *Proceedings of the 1995 Winter Simulation Conference*, pages 571-578, 1995.
- [38] S. McCanne and S. Floyd. The ns network simulator. Available on the web from the following site: <http://www.isi.edu/nsnam/ns/>.
- [39] Richard A. Meyer, Rajive Bagrodia. Path Lookahead: A Data Flow View of PDES Models. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS '99)*, May 1-4, 1999 in Atlanta, Georgia.
- [40] Myrinet Performance. <http://www.myri.com/myrinet/performance/>
- [41] Vincent D. Park and M. Scott Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In *Proceedings of INFOCOM'97*, pages 1405-1413, April 1997.
- [42] C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proc. of the ACM SIGCOMM*, October 1994.
- [43] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc On-Demand Distance Vector Routing. *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, New Orleans, LA, February 1999, pp. 90-100.
- [44] Poulton, J. An embedded DRAM for CMOS ASICs. *Proceedings. Seventeenth Conference on Advanced Research in VLSI*. pp. 288-302, 1997.
- [45] Scalable Network Technologies. Qualnet. <http://www.scalable-networks.com/>.
- [46] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, and Wen-King Su. The design of the Caltech Mosaic C multicomputer. In *University of Washington Symposium on Integrated Systems*, March 1993.
- [47] Charles L. Seitz and Wen-King Su. A Family of Routing and Communication Chips Based on the Mosaic. *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pp 320-337, MIT Press, 1993.
- [48] Sivakumar, Sinha, Bharghavan. CEDAR: A Core-Extraction Distributed Routing Algorithm. *IEEE Journal on Selected Areas in Communications*. Vol 17, No. 8, August 1999.
- [49] S. Skold, R. Ronngren, M. Liljenstam, and R. Ayani. Parallel simulation of mobile communication networks using time warp. *Proceedings of the 1995 EUROSIM Simulation Congress*, pages 559-564, 1995.
- [50] Mineo Takai, Jay Martin and Rajive Bagrodia. Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks. *Proceedings of MobiHoc 2001*, October 2001.

- [51] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpfen Matt Frank, Saman Amarasinghe and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, Mar/Apr 2002.
- [52] Michael Taylor. The Raw Prototype Design Document. <ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf>. 2002.
- [53] John Teifel. Interchip Communication in Asynchronous VLSI Systems. *Cornell Computer Systems Lab Technical Report CSL-TR-2002-1027*, October 2002.
- [54] UCLA Parallel Computing Laboratory. <http://pcl.cs.ucla.edu/>.
- [55] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. *IEEE Computer*, September 1997, pp. 86-93.
- [56] Brian White, Jay Lepreau, Shashi Guruprasad. Lowering the Barrier to Wireless and Mobile Experimentation. *Hotnets-I*, October 2002.
- [57] X. Zeng, R. Bagrodia, M. Gerla. GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. *Proceedings of the 12th Workshop on Parallel and Distributed Simulations—PADS '98*, Banff, Alberta, Canada, May 1998.