Self-stabilizing wormhole routing

Anthony Brandon Kenitzki
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1415

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI
SELF-STABILIZING WORMHOLE ROUTING

by

Anthony B. Kenitzki

Bachelor of Arts
College of Engineering
University of Nevada, Las Vegas
1997

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2002
The Thesis prepared by

Anthony B Kenitzki

Entitled

Self-Stabilizing Wormhole Routing

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Examination Committee Chair

Examination Committee Chair

Dean of the Graduate College
ABSTRACT

Self Stabilizing Wormhole Routing

by

Anthony B. Kenitzki

Ajoy K. Datta PhD. Examination Committee Co-Chair
Professor of Computer Science
University of Nevada. Las Vegas

Sébastien Tixeuil PhD. Examination Committee Co-Chair
Professor of Computer Science
Université de Paris-Sud. France

Parallel and distributed systems are composed of individual processors that communicate with one another by exchanging messages through communication links. When the sender and the receiver of a message are not direct neighbors, intermediate processors must cooperate to ensure proper routing.

Wormhole routing is most common in parallel architectures in which messages are sent in small fragments called flits. We assume that each processor will contain a single fixed-size flit buffer for each incoming link. A processor must forward the flit in a given link buffer to another processor before receiving another flit on that link. This permits messages to wind through the entire network from source to destination, resembling a worm. Wormhole routing is a lightweight and efficient method of routing messages between parallel processors.

Our purpose is to modify existing wormhole routing algorithms in familiar topologies to make
them self-stabilizing. Self-stabilization is a technique that guarantees tolerance to transient faults (e.g. memory corruption or communication hazard) for a given protocol. Transient faults would typically place the network in an illegitimate state, while Self-stabilization guarantees that the network recovers a correct behavior in finite time, without the need for human intervention. Self-stabilization also guarantees the safety property, meaning that once the network is in a legitimate state, it will remain there until another fault occurs.

This paper presents self-stabilizing network algorithms in the wormhole routing model, using the unidirectional ring and the two-dimensional mesh topologies. We chose the ring topology to illustrate the numerous difficulties of self-stabilization in a wormhole routing environment, even in one of the most simple network topologies. We then extend the results of the ring topology to a more complex two-dimensional mesh network.
# TABLE OF CONTENTS

**ABSTRACT** ................................................................. iii

**LIST OF FIGURES** ......................................................... vii

**LIST OF ALGORITHMS** .................................................. viii

**ACKNOWLEDGMENTS** ................................................... ix

**CHAPTER 1 INTRODUCTION** .............................................. 1
  Processor Networks ......................................................... 1
  Topology ........................................................................ 1
  Self-stabilization ........................................................... 2
  Routing Protocols .......................................................... 2
  Related Work ................................................................. 3
  Our Contribution ........................................................... 3
  Thesis Outline ............................................................... 4

**CHAPTER 2 SELF-STABILIZING WORMHOLE ROUTING IN RINGS** .......... 5
  Network Model and Topology ............................................. 5
  Wormhole Routing .......................................................... 5
  Specification ..................................................................... 6
  Problems Encountered ...................................................... 6
  Solutions and Ideas .......................................................... 7
  Assumptions and Conventions .......................................... 8
  Programming Conventions ............................................... 10
  Data Structures .............................................................. 10
  Algorithm ........................................................................ 12
    Helper Functions .......................................................... 12
    Algorithm Description ................................................... 13
  Proof of Correctness ....................................................... 18
    Legitimacy Predicates ...................................................... 18
    Correct Behavior .......................................................... 21
    Convergence ................................................................ 25
  Model Scalability ............................................................ 27

**CHAPTER 3 SELF-STABILIZING WORMHOLE ROUTING IN 2D MESHES** ........ 29
  Topology .......................................................................... 29
  New Problems Encountered .............................................. 29
  Solutions and Ideas .......................................................... 31
  Adaptive Wormhole Routing in a 2D mesh ...................... 31
    Problem Specification ..................................................... 32
    Model and Hypothesis ..................................................... 32

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
# LIST OF FIGURES

2.1 Wormhole Routing Transmission ................................................. 14
2.2 Wormhole Routing Deadlock ....................................................... 19
3.1 West-First Channel Numbering Scheme for an $n \times m$ Mesh ............................................................................ 33
3.2 West-First Channel Numbering Example on a $3 \times 3$ Mesh .............................................................................. 34
3.3 A Processor in a Mesh Topology .................................................. 42
3.4 Network Stabilization Part 1 ........................................................ 63
3.5 Network Stabilization Part 2 ........................................................ 63
## LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Self-stabilizing Wormhole routing on rings (Main program)</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Self-stabilizing Wormhole routing on rings (Receive actions)</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Self-stabilizing Wormhole routing on rings (Send actions)</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>The CHECK-PATH function</td>
<td>38</td>
</tr>
<tr>
<td>3.2</td>
<td>The FORCE-PATH function</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>The PROCESS-BODY function</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>The FREE-CHANNEL function</td>
<td>42</td>
</tr>
<tr>
<td>3.5</td>
<td>Self-stabilizing Wormhole routing on 2D Mesh (Main program)</td>
<td>44</td>
</tr>
<tr>
<td>3.6</td>
<td>Self-stabilizing Wormhole routing on 2D Mesh (Receive actions)</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Self-stabilizing Wormhole routing on 2D Mesh (Send actions)</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Self-stabilizing Wormhole routing on 2D Mesh (Error correction actions)</td>
<td>48</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

First, my thanks go first to Dr. Sébastien Tixeuil for mentoring me and making this thesis possible. I will always remember his patience with me every time I encountered what I thought to be an insurmountable problem. I have Dr. Ajoy K. Datta to thank for his direction and most importantly his fastidiousness with regards to this paper’s ledgibility and logical flow. My thanks to Dr. Burke for teaching me the math required for my proofs. I would also like to thank my thesis committee members Dr. Kazem Taghva and Dr. John Minor.

Thank you to all of the faculty of the UNLV Computer Science Department for a superb education and all of the support I received over the years. Finally, my sincerest thanks go to Dr. Maria Gradinariu, for some of the initial ideas that set this paper in motion.
CHAPTER 1

INTRODUCTION

1.1 Processor Networks

A massively parallel multi-computer consists of a number of processors directly connected through physical wires or shared memory registers. There are two main parallel architectures currently in use [Jur98]. In a physically-shared memory parallel computer, all processors access a shared pool of memory on an interconnection network. In a physically distributed memory parallel computer, each processor on the network has its own local memory and software programs. This paper utilizes the physically distributed parallel computer interconnection model in which processors will send messages to one another through directed channels.

1.2 Topology

The topology of a processor network tells us how all of the processors are interconnected, and most importantly which processors are adjacent to one another. Since it is expensive in terms of hardware and software complexity, processor networks are rarely completely connected. Therefore, it is necessary to develop message routing protocols in order for non-adjacent processors to communicate with one another.
1.3 Self-stabilization

In 1974, Dijkstra pioneered the concept of self-stabilization in a distributed network [Dij74]. A distributed system is self-stabilizing if it will return to a legitimate state in a finite number of steps, regardless of the initial state, and the system will remain in a legitimate state until another fault occurs. Thus a self-stabilizing algorithm will tolerate transient processor faults. These transient faults include variable corruptions, program counter corruptions (which temporarily cause a processor to execute any code in any order), and communications channel corruptions.

1.4 Routing Protocols

There are many routing protocols for interconnected processor networks, and some of the most popular schemes are store and forward, virtual cut-through, and wormhole routing. In the store and forward protocol, messages are broken into packets, and each packet is forwarded in full to each processor along a path. A processor cannot forward a message packet until the entire message packet is received. In 1979, Kermani and Kleinrock proposed an improvement on the store and forward routing scheme called virtual cut-through [Ker79]. Virtual cut-through is a protocol similar to store and forward, except that a packet is only stored at a processor if the required outgoing channel is not available. Wormhole routing uses a cut through routing technique with a few differences. In wormhole routing, message packets are broken into even smaller fragments called flow control digits (or flits). Routing and control information is stored in the first flit (also called the header flit). As the header flit moves through the network toward its destination, every processor it passes through will reserve a channel for the remaining flits of the message. When the last (tail) flit of the message passes through a processor, the channel reservation is released. If a header flit reaches a processor where there is no available output channel resource, the other flits in the message packet remain where they are until the header advances. Thus the flits of the packet wind from the header to the source like a worm.

A routing protocol needs to be low latency, simple, high throughput, and robust [Glass92].
Latency refers to the time that it takes for a packet to travel from the source to its destination. Wormhole routing has an extremely low transmission latency, since a flit of a message packet does not have to wait for the entire packet to arrive at a processor before it can be transmitted again. The protocol is simple in that the packet buffers required at each processor need only be a few flits large (a few bytes). High throughput is achieved through adaptive routing, in which a message may take many paths from the source to the destination. A message may make many adaptive turns in order to avoid congestion, meaning that if a header flit reaches a processor where an outgoing channel is blocked, it is allowed to move in another direction.

1.5 Related Work

Considerable research has been done in making wormhole routing robust (fault tolerant). Papers such as [Da90] add virtual channels to the network to handle faults. Virtual channels divide a single physical channel into many, sharing the bandwidth between them. Papers such as [Glass93] use an adaptive turn-based model to avoid faults. If a faulty processor is encountered on the network, a message will choose a path around the failed processor. All of these wormhole routing papers are written to tolerate fail-stop faults, meaning that one or more processors will cease to function entirely on the network, while the remainder will faithfully execute their programs. Papers such as [BDT99] present self-stabilizing network algorithms in a virtual cut through setting, but not in a wormhole routing environment.

1.6 Our Contribution

Self-stabilization takes the concept of a robust wormhole routing algorithm one step further. This paper presents self-stabilizing wormhole routing algorithms for the ring and mesh topologies. We show that even in a simple ring topology, self-stabilizing wormhole routing is not trivial. A local processor fault can cause message flits to be lost or introduced at random, leaving fragmented and corrupted messages on the network. Data flits can flood all of the processor buffer flits on the
network. Misrouted leader flits can cause the network to deadlock. All of these problems have to be resolved without human intervention. In a self-stabilizing environment the mesh algorithm is beleaguered not only by faulty messages, but also by faults in the channel reservations of active message paths (also known as circuits). A message circuit can branch, sending flits in all directions across the network. A cycle can be created in the circuit, thus causing a deadlock of the network. Circuits can be broken in the middle, leaving unroutable message fragments in the network. Finally, stale circuits can remain in the network forever if no tail flit passes through them. Once again, all of these faults must be corrected in finite time.

1.7 Thesis Outline

In chapter two, we first illustrate the possibility of self-stabilizing wormhole routing using a unidirectional ring. Chapter three extends the results of chapter two to a more complicated network topology and routing algorithm. In chapter three we implement a self-stabilizing adaptive wormhole routing algorithm in a mesh topology. This algorithm is a modified version of the two-dimensional mesh adaptive routing algorithms presented by Glass and Ni in both [Glass91] and [Glass92]. Finally in chapter four, we present conclusions and future research opportunities.
In this chapter we present a self-stabilizing wormhole routing algorithm for a ring topology. First, we formally define the network model, wormhole routing, and self-stabilization. Next we illustrate the inherent difficulties wormhole routing presents even in a topology as simple as a ring. Last, we present the self-stabilizing algorithm and proof.

2.1 Network Model and Topology

Our network model is a clockwise unidirectional ring \( G = \{V, E\} \), where \( V \) is a set \( \{1, 2, 3, \ldots, n\} \) of processors, and \( E \) is the set \( \{(1, 2), (2, 3), (3, 4), \ldots, (n, 1)\} \) of channels connecting the processors in a ring shape. An individual Processor \( P \) can only receive messages on its incoming (right) channel \((\text{predecessor}(P), P)\) and it can only transmit messages on its outgoing (left) channel \((P, \text{successor}(P))\).

2.2 Wormhole Routing

In wormhole routing, message packets are broken into flow control digits (or flits), each flit is only a few bytes in size. All routing and message control information is stored in the first flit (also called the header flit). As the header flit moves through the network toward its destination, every processor it passes through will reserve a channel for the content (data) flits of the message to pass through. The other flits of the message will thus follow the header flit in a pipe-line fashion. When
the last (tail) flit of the message passes through a processor, the channel reservation for that message is released. If a header flit reaches a processor where there is no available output channel resource, the other flits in the message packet remain where they are until the header flit advances. Thus the flits of the packet wind from current processor containing the header flit, all the way back to the source processor (much like a worm).

2.3 Specification

Our algorithm is self-stabilizing, meaning that we will prove that the following three properties hold:

1. **Convergence**: Regardless of initial state, the network must return to a legitimate state in finite time. Local processor faults can corrupt variables to completely arbitrary values. Thus message flits and wormhole paths can be lost or corrupted. Convergence guarantees that the algorithm will recover from these faults.

2. **Liveness**: A fault must not cause the network to deadlock, livelock, or starve. A network is in a deadlock state when one or more processors are waiting for a resource that will never be released. Livelock happens when all processors are executing as normal, but the algorithm fails to progress. Starvation occurs when a processor is prevented from performing a critical function forever.

3. **Closure**: Once the network is in a legitimate state, it will remain in a legitimate state until a fault occurs. The legitimate state of the network is defined in Section 2.10.1

2.4 Problems Encountered

In the self-stabilizing environment, network faults can corrupt the local variables of any network processor. Thus messages flits and their wormhole routing paths can be spontaneously introduced, lost, or corrupted. There are two kinds of corrupted messages that we may encounter:
1. Messages that are structurally not correct. A transient fault can cause message fragments to be corrupted beyond usefulness or lost altogether. These messages will not contain both a header flit and a tail flit and are of one of the following types:

   (a) **Headerless Message Fragments**: This happens when several message flits are in the network without a header.

   (b) **Header Message Fragments**: A header without a tail moves alone in the network.

   (c) **Headerless Flooding**: A single message without a header and without a tail occupies all the network flits and moves throughout the network.

   (d) **Misrouted Messages**: A message header flit is forwarded onward rather than delivered by the destination processor. It is then possible to deadlock the network.

2. Messages that are logically not correct. These messages will contain both a header and a tail, but the contents of the message will be corrupted from an application point of view or from a routing point of view.

2.5 Solutions and Ideas

Our algorithm implements the following solutions to these problems:

**Headerless Message Fragments.** If the header of a message is lost before it reaches its destination, we must handle and discard this corrupted message. When a header flit of a message is received in the incoming channel of a processor, the channel is locked for that message until the tail of that message is encountered. Whenever a processor receives a non-header message fragment on an incoming channel that is not reserved for that message, then the fragment is discarded.

**Header Message Fragments.** Corruption can cause the network to be flooded with message headers without tails. To correct this, we can implement a maximum number of hops on a message header. When a processor receives a header, it will know how long the header has been active on
the network. A global maximum time can be specified by the application, e.g. if an upper bound on \( n \) is known to the application, this bound can be used as the maximum number of hops.

**Headerless Flooding.** Since the network can start in any arbitrary state, it is possible to have every processor filled by a non-header value. All processors believe that they are forwarding a valid message. The solution to this is to have every processor count how many flits have been forwarded in a message. The application layer will specify a maximum message length. Since the headerless message has no end, at least one processor eventually decides to begin discarding the message fragments.

**Misrouted Messages.** Program counter corruption can cause a processor to simply forward a message rather than deliver it. This can be dealt with in the same manner as header message fragments. So long as the maximum number of hops for a message is set to \(|V'| - 1\), a message can never be routed again by its originator.

**Messages that are logically not correct.** It is possible for a header flit to contain a destination that does not exist in the network. Since each header flit has a timeout stamp in the header, the message is eventually dropped. The message will then be a headerless message, which was handled above.

Incomplete or complete messages with corrupted data can be delivered at a destination processor to the application layer. It is the responsibility of the application layer to recognize and discard the message in this case.

2.6 Assumptions and Conventions

Our algorithm uses the following assumptions with regards to the system:

**Atomic Actions**: All enabled actions within a single processor in the network are executed atomically. This does not prevent other processors from executing actions at the same time.
**Connection Management**: There are two types of network communications, connectionless and connection-oriented [Gouda98]. In a connectionless communication, a Processor $P$ can flood another Processor $Q$ with message packets without regard for the readiness of $Q$ to accept those messages. The Processor $Q$ is allowed to discard any messages that it cannot process or hold in its local buffer. Wormhole routing requires connection-oriented strict flow control, since only one flit can be held by a given processor at any time. We must assume a self-stabilizing alternating bit protocol such as the algorithm described in [AB98]. Thus we can prevent a Processor $P$ from sending more than one flit at a time to a Processor $Q$ that is ready to accept one.

**Crash Faults**: A processor that crashes will instantly reset, with all variables set to arbitrary values. Processors are always active and available on the network.

**Fair Scheduler**: We model a large local multi-processor system. All processors in the network move at nearly the same speed. Channels can be modeled by physical wires with a known-bound delay, or by read-modify-write shared registers. Thus we assume a fair asynchronous environment for all processors. By fair, we mean that if a processor has a guarded command that is continuously enabled, then this guard is eventually executed.

**Hard-Coded Constants**: Constants are hard coded and cannot be corrupted. Constant values occupy static and read-only memory. Typically, constants for our algorithm are inputs from the application layer.

**Rare occurrence of Errors**: In any infinite execution, the number of faulty actions is finite. Put simply, all faults have to eventually stop in order for the error correction code to return the network to correct behavior. Those failures are transient failures: after some time, they cease to occur.

**Single Sender**: Only one processor on the network contains the code to introduce new messages onto the network.
Timeout Actions: Since we assumed that we have a fair scheduler, that all processor execution speeds are similar, and that communication delays are bounded, then we can assume that all timeout actions are accurate based on the local clock at each processor.

Variable Domains: Each variable has a set of valid values that it may take. The variable cannot be corrupted to a value outside of the legal domain of that variable.

2.7 Programming Conventions

This program is written as a series of guarded commands. This concept was first introduced by Dijkstra [Dij75] as ...constructs that allow non-deterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. An action is of the form \(<guard> \rightarrow <statement>\). A guard is a boolean expression over processor variables and an input (such as a message). A statement is a sequence of program statements separated by the semicolon character. A single action can only be executed if the guard is evaluated to be true, and all statements in an action are executed atomically.

The syntax for branching is borrowed from [Gouda98]. An if condition along with any statements preceded by [] symbols are evaluated for truth in parallel. A single true guard is selected nondeterministically for execution from that list.

2.8 Data Structures

In this section we present the data structures and local variables used by the algorithm.

Flits are data structures a few bytes in length each. Programatically, we reference a member of a flit data structure as \(<flit>.<variable>\). We will use the following data structures for the three types of flits:

1. **Header Flits (hflit)** consist of a global unique message identifier (mid), a time to live (ttl), and a destination (dest). For example: send hflit(mid, ttl, dest)
2. **Data Flits** (dflit) consist of a message id and a fragment of the actual message payload to be sent. *For example:* send dflit(mid, dat)

3. **Tail Flits** (tflit) consist only of a message identifier. *For example:* send tflit(mid)

The maximum time to live in hops (maxttl) constant and the maximum message length constant (maxlen) are inputs supplied by the application layer. The maximum message id (maxmid) is the largest message id that is allowed by the processor software or register size.

The left channel lock (lchannel) variable holds the current message identifier to transmit, or the value 0 if the local processor is not routing a message. If a Processor \( P \) is not routing a message, then \( P \) knows that it may deliver received data and tail flits.

The total flits received (ftotal) variable is used to account the total flits received for a message. This variable is used to prevent a data flit flood, where one or more data flits can remain in the network forever moving in a circle.

Wormhole Routing flow control is guaranteed by a Clear To Send (CTS) wire that connects each processor in a uni-directional link. The CTS wire on Processor \( P \) for the link \(<\text{predecessor}(P)\) \( P \)> is set to LOW when \( P \) is ready for a new message, or it is set to HIGH when it is not. This wire can also be modeled as a read-modify-write shared register between the two Processors in the unidirectional link. A Processor can read the CTS variable of its successor, but it can only write to its own. Thus the CTS variable will allow only one flit to be in the flit buffer of a processor at any time, and that the processor will not accept another flit into its local buffer until it is empty.

Each processor will have a single CTS variable for each incoming link. This variable will simply be called CTS for the ring protocol, since every Processor only has a single incoming link. An example of wormhole flow control with two intermediate routing processors is detailed in example Figure 2.1.

The Buffer variable represents the flit buffer of a processor. The Buffer variable can only hold a flit value or no value at all (empty).
2.9 Algorithm

First we present the helper functions used by the algorithm in Section 2.9.1. The code of the main program is in Section 2.9.2. Lastly, in Section 2.9.2 we present a brief description of each action in the main program.

2.9.1 Helper Functions

The following are the functions called in the main program.

SENDNEWMESSAGE is a function that will activate when the privileged Processor $P$ is idle for too long (that is, when $P$ has nothing to forward and has nothing in its flit buffer). The processor will generate a new unique message id, an arbitrary destination, and then it will send its left neighbor a new correct message starting with a header, numerous data flits, and a tail flit. We will assume fairness, such that eventually every processor will receive a message from $P$.

DELMERMSG is a function that will deliver a message to the application layer, clear out the channel flit buffer, and set the CTS variable of the incoming channel to LOW.

DISCARD is a function that will clear out the channel flit buffer, and set the CTS variable of the incoming channel to LOW.

RECV is a function that utilizes the previously assumed self-stabilizing connection protocol. The RECV function is used to read transmitted data from incoming channel.

SEND is a function that transmits data across an outgoing channel. Since we assumed a self-stabilizing connection protocol, a Processor $P$ can only initiate a single SEND on an outgoing channel when a corresponding RECV action is activated on the other side of the link by a Processor $Q$.

TIMEOUT is a function that will wait a sufficiently long time for a network condition to hold. The
syntax of the TIMEOUT actions is borrowed from [Gouda98]. These are normal timeout actions that can be implemented with a local clock at each processor. In Gouda’s book, a TIMEOUT action is of the form \( \text{TIMEOUT: } \langle \text{protocolpredicate} \rangle \rightarrow \langle \text{statement} \rangle \). A protocol predicate is of the form:

\[
(\text{local predicate of processor } p) \land \\
(mp\#ch.p.q \leq kp) \land \\
(mq\#ch.q.r \leq kq) \land \\
\ldots \\
(mv\#ch.v.w \leq kv)
\]

The details of a protocol predicate are as follows:

1. A local predicate of Processor \( p \) is simply a regular guard similar to any other action. It is a boolean predicate built using local variables in \( p \).
2. \( kp, kq, \ldots, kr \) are integers.
3. \( mp \) are messages sent from Processor \( p \) to Processor \( q \).
   
   \( mq \) are messages sent from Processor \( q \) to Processor \( r \).
   
   \( \ldots \)
   
   \( mv \) are messages sent from Processor \( v \) to Processor \( w \).
4. \( mp\#ch.p.q \) is a count of all messages sent from Processor \( p \) to Processor \( q \) on the channel \( < p, q > \in E \).
5. Every pair consecutive conjuncts of the form \( (ms\#ch.s.t \leq ks) \) and \( (mt\#ch.t.u \leq kt) \) require that each action that can send messages from Processor \( t \) to Processor \( u \) must be preceded by a RECV action from Processor \( s \) to Processor \( t \).

2.9.2 Algorithm Description

The algorithm (presented as Algorithms 2.1, 2.2, and 2.3) is composed of nine actions total. Each action performs the following functions:
Figure 2.1: Wormhole Routing Transmission

Algorithm 2.1 Self-stabilizing Wormhole routing on rings (Main program)

program Self-Stabilizing Wormhole Routing in a Ring
process P [ i: 0..n-1 ]
inputs maxttl.
   maxmid.
   maxlen
var Ichannel: {0..maxmid}.
   ftotal: {0..maxlen+1}.
   CTS: {LOW, HIGH}
   Buffer: {<empty>, hflit, dflit, tflit}
begin
   RECEIVE actions (presented as Algorithm 2.2)
   [1] SEND actions (presented as Algorithm 2.3)
end
1. *Receive actions* are formally presented as Algorithm 2.2 and are described below:

**R1** The action (R1) allows a processor to receive header flits. Header flits are first checked to see if they have arrived at the correct destination. When a header flit is delivered the lchannel lock variable is set to 0, the *not routing* status. Header flits that are not delivered are first checked for faults (time to live). Faulty header flits are discarded, and all others are written to the local Buffer variable to be routed. Once a flit is written to the Buffer variable, the clear to send (CTS) variable is set to HIGH (not ready to receive).

**R2** The action (R2) allows a processor to receive data flits. When a data flit is received, the lchannel variable is examined against the message identifier of the data flit. If the lchannel variable is set to 0, then the flit is delivered. If the lchannel variable is not equal to the message id of the data flit, then the flit is discarded. The flit is only routable if the message id of the data flit is equal to the lchannel variable, and the total flits received ftotal variable does not exceed the maxlen constant. Routable data flits are written to the Buffer variable and the CTS variable is set to HIGH.

**R3** The action (R3) allows a processor to receive tail flits. Tail flits do not require a check against the ftotal variable, but they are handled the same in all other aspects as data flits in (R2).

**R4** The action (R4) allows the network to recover from a deadlock. This action does not activate until a sufficient time has passed such that no message flit may be on any channel in the network. Since Processor P is unable to receive a new flit for an extremely long time, and it not clear to receive new flits, then P sets Buffer to nothing, and the CTS variable to LOW (ready to receive a new flit).

2. *Send actions* are formally presented as Algorithm 2.3 and are described below:

**S1** The action (S1) allows a processor to route a header flit. A processor will lock its outgoing channel, initialize its ftotal variable to 1, transmit the flit, and set its CTS to LOW.
Algorithm 2.2: Self-stabilizing Wormhole routing on rings (Receive actions)

/* Receive a header flit. */
(R1) RECV hflit(mid, ttl, dest) ∧ CTS = LOW →
    if hflit.ttl ≤ maxttl ∧ hflit.dest = i →
        lchannel := 0:
        DELIVERMSG hflit(mid, ttl, dest):
        (D1) hflit.ttl > maxttl →
            lchannel := 0:
            DISCARD hflit(mid, ttl, dest):
        hflit.ttl ≤ maxttl ∧ hflit.dest ≠ i →
            - Save the flit to be sent later.
            Buffer := hflit(mid, ttl, dest):
            CTS := HIGH:
    fi

/* Receive a data flit. */
(R2) RECV dflit(mid, dat) ∧ CTS = LOW →
    if lchannel = 0 →
        DELIVERMSG dflit(mid, dat):
        (D2) lchannel = dflit.mid ∧ ftotal > maxlen →
            DISCARD dflit(mid, dat):
        lchannel = dflit.mid ∧ ftotal ≤ maxlen →
            Buffer := dflit(mid, dat):
            CTS := HIGH:
        (D3) lchannel > 0 ∧ lchannel ≠ dflit.mid →
            DISCARD dflit(mid, dat):
    fi

/* Receive a tail flit. */
(R3) RECV tflit(mid) ∧ CTS = LOW →
    if lchannel = 0 →
        DELIVERMSG tflit(mid):
        lchannel = tflit.mid →
            Buffer := tflit(mid):
            CTS := HIGH:
        (D4) lchannel > 0 ∧ lchannel ≠ tflit.mid →
            DISCARD tflit(mid):
    fi

/* Have not received anything for a long time. */
/* Processor P assumes that it is holding up the network. */
(R4) TIMEOUT CTS = HIGH ∧
    ∧ (flit#ch.1.2 + flit#ch.2.3 + ... + flit#ch.n - 1.n + flit#ch.n.1 = 0) →
        Buffer := <empty>:
        CTS := LOW:
    fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
/* Can send a header flit. */
(S1) if Left.CTS = LOW ∧ Buffer = hflit(mid, ttl, dest) →
  |channel := Buffer.mid;
  Buffer.ttl := P.Buffer.ttl + 1;
  ftotal := 1;
  SEND hflit(mid, ttl, dest);
  CTS := LOW;
/* Can send a data flit. */
(S2) if Left.CTS = LOW ∧ Buffer = dflit(mid, dat) →
  ftotal := ftotal + 1;
  SEND dflit(mid, dat);
  CTS := LOW;
/* Can send a tail flit. */
(S3) if Left.CTS = LOW ∧ Buffer = tflit(mid) →
  ftotal := 0;
  |channel := 0;
  SEND dflit(mid, dat);
  P.CTS := LOW;
/* Detect this local invalid condition and repair it. */
(S4) if P.CTS = HIGH ∧ Buffer = <empty> →
  CTS := LOW;
/* The following code only exists in the privileged processor. */
/* It will allow a message to be introduced. */
(S5) if TIMEOUT Buffer := <empty>
  ∧ P.|channel = 0
  ∧ Left.CTS = LOW
  ∧ (flit#ch.1.2 + flit#ch.2.3 + ... + flit#ch.n - 1.n + flit#ch.n.1 = 0) →
    |channel := 0;
    SENDNEWMESSAGE;
fi
S2 The action (S2) allows a processor to route a data flit. A processor will transmit the flit, increment the ftotal variable, and set its CTS to LOW.

S3 The action (S3) allows a processor to route a tail flit. A processor will transmit the flit, set the Ichannel variable to 0, and set its CTS to LOW.

S4 The action (S4) prevents a local fault condition in which the CTS variable is set to HIGH, and the Buffer variable is empty. A processor will merely reset its CTS variable back to LOW.

S5 The action (S5) is a TIMEOUT action that prevents a network deadlock condition. Just like action (R4), the action is not activated until enough time has passed such that every message channel on the network should be empty. The network can deadlock if all buffer flits on the network are full, and no processor has a CTS value of LOW. This is clearly illustrated in Figure 2.2.

2.10 Proof of Correctness

We will now prove that this algorithm satisfies the specification as defined in Section 2.3. First the legitimacy predicates are introduced in Section 2.10.1, then the Closure and Convergence properties are proven for this algorithm in Section 2.10.2 and Section 2.10.3 respectively.

2.10.1 Legitimacy Predicates

Legitimate State: In Section 2.1 we stated that our network is a set $G = \{V, E\}$ such that $V$ is the set of all processors, and $E$ is the set of all channels. The network $G$ is considered to be in a legitimate state if all of the messages in the network channels are structurally correct, and the network processors satisfy certain passage predicates. A transient fault can cause a message to be corrupted or a critical piece of the message to be completely lost. Local variables at each processor in the network can be set to arbitrary values within their variable domains. The network is in an
Figure 2.2: Wormhole Routing Deadlock
illegitimate state until all of the affected and associated flits are removed from the network, and each processor has corrected critical local variables. Formally, the network is in a legitimate state if the conjunction of all the following processor and message predicates is satisfied.

2.10.1.1 Processor Predicates

Local processor safety predicates are common in any self-stabilizing algorithm. All of the following local processor predicates must be satisfied by each processor on the network:

P1 No processor can have a CTS value of HIGH if it has an empty Buffer. \[ \{ P | P \in \mathcal{V} \land P.Buffer = \text{<empty>} \land P.CTS = \text{HIGH} \} = \emptyset. \]

P2 Each Processor \( P \) has an Ichannel value equal to zero (not forwarding), or equal to the message id of the last header flit received by \( P \).

P3 At least one processor has a Buffer variable = <empty> and a CTS variable = LOW. \[ \exists \alpha \in \{ P.CTS | P \in \mathcal{V} \} \text{ s.t. } \alpha = \text{LOW}. \]

2.10.1.2 Message Predicates

Message predicates are unusual in a self-stabilizing setting, however our algorithm requires all of the following predicates to be satisfied:

M1 A message is constructed with a header flit, one or more data flits, and a tail flit. The difficulty with this predicate is that many messages will not have all of their flits on the network at one time. A header flit and multiple data flits may have been legitimately delivered to the destination while a tail flit remains on the network. A header flit may be on the network while data flits and the tail flit wait to be transmitted. To get all of the flits for a single message \( M \), for a path \( C \) take:

\[ (\text{flits delivered}) \cup (\text{flits in transit, reverse-ordered in } C) \cup (\text{flits not yet transmitted}). \]
For example: Let $H$ be a header flit, $D_1$ be a data flit, and $T$ be a tail flit:

RECEIVER PROCESSOR Delivered: $H$
INTERMEDIATE PROCESSOR 2 Flit Buffer: $D_1$
INTERMEDIATE PROCESSOR 1 Flit Buffer: $D_2$
SENDER PROCESSOR Application Buffer: $T$
The correct message is: $H, D_1, D_2, T$

M2 The number of data flits in the message is less than $\text{maxlen}$.

M3 The time to live variable in a header flit will never exceed $\text{maxttl}$.

M4 The message identifiers of the header, data, and tail flits remain the same and equal throughout the life of a message.

2.10.2 Correct Behavior

First in Section 2.10.2.1 we prove that when the network is in a legitimate state, that the program exhibits liveness (no deadlocks, livelocks, or starvation). Afterwards, in Section 2.10.2.2 we prove that when the network is in a legitimate configuration, then the algorithm behavior is correct.

2.10.2.1 Progress

The first step is to prove three important features of distributed routing algorithms:

Lemma 2.1 (Deadlock) Starting from a configuration that satisfies the legitimacy predicate, the network will not deadlock.

Proof. Deadlocks occur when processors are waiting on resources that are never freed. It is proven in [Da87] that a routing algorithm in a direct network is deadlock free if there is no cycle in the channel dependence graph. Wormhole routing in a unidirectional ring will contain no cycles in the channel dependence graph since the sender cannot route its own message twice. □
Lemma 2.2 (Starvation) Starting from a configuration that satisfies the legitimacy predicate, the network will not starve.

Proof. Starvation occurs whenever a processor needs to send a message, but it too busy routing messages for other processors. Starvation cannot happen since we assumed only a single sender on the network. Therefore, the action (S4) will eventually be activated on the sender processor. □

Lemma 2.3 (Livelock) Starting from a configuration that satisfies the legitimacy predicate, the network will not livelock.

Proof. The network is livelocked when processors are executing as normal, but no progress is made. The routing algorithm progresses when message flits are removed from the network. The only time message flits are not eventually delivered is when they are stuck on the network forever. These faulty message flits are normally handled by the DISCARD actions (D1) and (D2). Header-only messages and misrouted headers are dealt with in (D1), since all header flits are limited to \( \text{max}_t \) hops. A headerless flood is eliminated by (D2), since only \( \text{max}_l \) many data flits can pass through a processor. The predicates (M1) and (M4) guarantee that every message on the network is structurally correct. Every message will have a tail flit following data flits, following a header flit all with the same message identifier, so a livelock cannot occur. □

Theorem 2.1 (Progress) Starting from a configuration that satisfies the legitimacy predicate, the network will not deadlock, livelock, or starve.

2.10.2.2 Reliable Delivery

Next we prove that every flit transmitted will be received, and every flit received will be transmitted until it is delivered.

Lemma 2.4 (Reliability 1) Starting from a configuration that satisfies the legitimacy predicate, every flit sent to a processor is eventually received.
Proof. After a Processor $P$ executes a SEND function on a left channel, the successor of $P$ ($S(P)$) will have a RECV action $r \in \{(R1), (R2), (R3)\}$. We assumed a fair scheduler, so the RECV action $r$ on $S(P)$ will be activated after a short time.

Lemma 2.5 (Reliability 2) Starting from a configuration that satisfies the legitimacy predicate, every flit received at a processor is eventually delivered or written to the local Buffer variable.

Proof. The RECV actions that can process a newly received flit are (R1), (R2), and (R3). There are only three possible outcomes from the statements within those actions: DISCARD, DELIVER, or Buffer-Write.

DISCARD statements are protected by guards (D1), (D2), (D3), and (D4) that have the following conditions (in order):

1. (D1) A Header Flit is received that has been forwarded too many times in the ring. We assume a correct network condition (M3), so the guard (D1) will not evaluate true.

2. (D2) At least one data flit too many is received for the current message. This happens when a processor receives maxlen data flits of a message before a tail flit. In (R1), the $ftotal$ variable is set to 0 for a new message passing through a processor. We assume a correct network condition ((M1) and (M2)), so the guard (D2) will not evaluate true.

3. (D3) and (D4) The outgoing channel is locked for a message id other than that of a received data or tail flit. This can only happen if the message is a headerless fragment, the channel lock has been corrupted, or the flit message id has been corrupted. In (R1), the header flit of a message sets the $Ichannel$ variable to the message id of the header. We assume a correct network condition ((P2), (M1), and (M4)), so the guards (D3) and (D4) will not evaluate true.
Therefore, we can conclude that in a correct network condition a received buffer flit will either be delivered or written to the flit buffer. The network predicate (P3) will hold in a legitimate configuration. □

**Lemma 2.6 (Reliability 3)** Starting from a configuration that satisfies the legitimacy predicate, every processor having an incoming channel Buffer variable containing a flit eventually transmits this flit.

**Proof.** Assume that a Processor \( P \) has a flit in its Buffer variable and that the flit is never transmitted. To evaluate true, the guards of actions (S1), (S2), and (S3) require the CTS variable of \( S(P) \) to be set to LOW. Thus we know that \( S(P) \) has a CTS value of HIGH.

Since the network is in a legitimate state, we can assume that (N4) holds. Thus we can safely conclude that at least one processor in \( V \) has a CTS value = LOW.

Since \( G \) is a unidirectional ring, the set of Processors \( \{ J | J \in V \land J \neq P \} \) on the network is upstream relative to \( P \).

The predicate (P1) guarantees that every Processor \( \in J \) with a CTS value of HIGH has a flit in its Buffer variable. Let \( Q \) be the first Processor upstream to \( P \), such that \( Q.\text{CTS} = \text{LOW} \).

The predecessor of \( Q \), \( P(Q) \) must have a flit in its Buffer, and thus one of \( P(Q) \)'s SEND actions must be true. Eventually (S1), (S2), or (S3) will be activated on \( P(Q) \).

The result of the actions in (S1), (S2), and (S3) are: the flit in \( P(Q).\text{Buffer} \) is transmitted to \( Q \). \( P(Q).\text{Buffer} \) is set to <empty>, and \( P(Q).\text{CTS} \) is set to LOW.

By induction, this process will repeat until \( S(P).\text{CTS} = \text{LOW} \), and one of the SEND actions in \( P \) will be activated. Therefore, any flit that is written to a Processor Buffer variable is eventually transmitted. □

**Theorem 2.2 (Reliable Delivery)** Starting from a configuration that satisfies the legitimacy predicate, every flit sent to a processor is eventually received, every flit received at a processor is eventually delivered or written to the local Buffer variable, and finally every processor having an incoming channel Buffer variable containing a flit eventually transmits this flit.
Theorem 2.3 (Closure) *Once the network is in a legitimate state, all new and structurally correct messages introduced on the network will eventually reach their destination.*

**Proof.** This follows from both Theorem 2.1 and Theorem 2.2. In Theorem 2.1 we proved that the algorithm cannot be deadlocked or livelocked, and that new messages are always introduced onto the network. Then in Theorem 2.2 we proved that every one of those introduced messages is eventually delivered. □

2.10.3 Convergence

Lastly, we prove that this algorithm will converge to a legitimate state from any arbitrary initialization in finite time. This is done via the convergent stair method from [Gouda91]. In this method the system converges to fulfill a number of predicates $A_1, A_2, \ldots, A_k$, such that each $1 \leq j < k$. $A_j - 1$ is a refinement of $A_j$ [Dol00]. A predicate $A_j - 1$ refines $A_j$ iff $A_j$ holds when $A_j - 1$ holds. Thus each $A_j$ predicate is called an attractor. Using the convergent stair method, we show that TRUE is an attractor for processor predicates. The conjunction of all processor predicates is an attractor for the message predicates. Thus we prove that the conjunction of all predicates will eventually hold in the system, and the system converges to a legitimate state.

2.10.3.1 Processor Predicates

First we prove that starting from an arbitrary configuration, all of the processor legitimacy state predicates will be satisfied in finite time.

**Lemma 2.7 (P1)** *Starting from an arbitrary configuration, (P1) eventually holds.*

**Proof.** This is guaranteed by the (S4) action, which will eventually be executed on $P$. Thus the statements in action (S4) will satisfy network predicate (P1). □

**Lemma 2.8 (P2)** *Starting from an arbitrary configuration, (P2) eventually holds.*
Proof. This follows from (R1). The lchannel variable is set to the message identifier of the received header flit if it is forwarded, or to 0 if the header flit is delivered. Thus a new header flit will satisfy network predicate (P2) for Processor P.

Lemma 2.9 (P3) Starting from an arbitrary configuration, (P3) eventually holds.

Proof. The domain of a CTS variable is \{HIGH, LOW\}. Thus \(\forall \alpha \in \{P.\text{CTS}|P \in V\}. \alpha = \text{HIGH} \lor \alpha = \text{LOW}\).

Now assume for a contradiction that \(\{P.\text{CTS}|P \in V\} \subset \{P.\text{CTS}|P \in V \land P.\text{CTS} = \text{HIGH}\}\)

This means that: \(\forall \alpha \in \{P.\text{CTS}|P \in V\}. \alpha = \text{HIGH}\).

Therefore the actions (S1), (S2), and (S3) cannot be enabled on the network, since they all require a LOW CTS variable. The network is in a deadlocked state if no SEND actions can be performed, since a RECV action requires a corresponding SEND action. The deadlocked state is illustrated in the example Figure 2.2.

These deadlocks are resolved by packet preemption as discussed in [Ni91]. After some time, the timeout action (R4) will be activated on a nonempty set \(W\) of processors, and the Buffer variables of one or more deadlocked will be discarded from the network, and the CTS variables are set to LOW.

After all of \(W\) executes (R4), we have \(\forall \alpha \in W. \alpha.\text{CTS} = \text{LOW} \land \alpha.\text{Buffer} = \text{HIGH}\). Therefore (P3) holds since \(\exists \alpha \in \{P.\text{CTS}|P \in V\}\) s.t. \(\alpha = \text{LOW}\).

2.10.3.2 Message Predicates

Next we prove that starting from an arbitrary configuration, all of the message legitimacy state predicates will be satisfied in finite time.

Lemma 2.10 (M1-M4) Starting from an arbitrary configuration, predicates (M1) through (M4) eventually hold.
Proof. We need to prove that all faulty messages will be removed from the network. Each flit type is handled individually.

**Faulty Header Flits** The action (R1) guarantees that these flits will eventually either be delivered, or that (D1) will remove them when their maximum number of hops has expired.

**Faulty Data Flits** Assume that a data flit \( d \) with message id \( i \) can remain on the network forever. We can also assume that every processor on the network thinks that it is forwarding message id \( i \). or else a processor with a different Ichannel variable will execute (D3) and drop the flit. Every time the data flit \( d \) is received by a Processor \( P \). \( P \) will increment its local \( ftotal \) variable, thus correctly recording the total number of flits forwarded for message id \( i \) so far. However, (R2) guarantees that this can happen at most maxlen times before the message is discarded by guard (D2).

**Faulty Tail Flits** These can never pass through a processor twice. Before forwarding a tail to a neighbor, a processor will reset its Ichannel and ftotal variables. Thus eventually a tail flit will reach a processor where it is delivered or discarded.

\[ \square \]

**Theorem 2.4 (Convergence)** Once the network is in an illegitimate state, it will return to a valid state in finite time. We have proven that each of the predicates will hold after finite time, so the conjunction of the predicates will hold after a finite time.

2.11 Model Scalability

We have discussed so far the self-stabilizing properties of a wormhole routing uni-directional ring. To make this algorithm more useful, we need to include multiple senders. When multiple senders are introduced onto the network, the following two things can happen:

1. It is possible to starve a processor. A processor that needs to send a message can be prevented from doing so by other processors in a unidirectional ring.
2. It is possible for two messages to deadlock. Since we have a ring topology, any two messages introduced onto the network by different processors can acquire resources in a circular-dependent manner.

Both of these problems can be avoided by adding more available channels for any processor to initiate a message upon. A simple solution presented in [Da90] is to add multiple virtual channels to the network for each physical channel. Virtual channels are logical channels which may share the same physical wire, but each virtual channel contains its own flit buffer, control program (including local variables), and data path. The flit buffers can be represented as an array of n flit buffers, along with an array of n channel lock variables. A flit sent from flit buffer(i) over the physical channel will be written to flit buffer(i) at the destination processor. If one virtual channel is allowed per sender processor, then we can make the same self-stabilizing guarantees than that of a single processor and a single channel. Mutual exclusion is another solution to these difficulties. If this stabilizing wormhole routing algorithm is layered on top of a stabilizing token passing algorithm, we can also make the same self-stabilizing guarantees. Any processor that has a token may send a message if required, and pass the token on after all messages flits have been transmitted. Thus a processor should never receive routing instructions while it has the token.
CHAPTER 3

SELF-STABILIZING WORMHOLE ROUTING IN 2D MESHES

In this chapter we present a self-stabilizing adaptive wormhole routing algorithm for a mesh topology. First, we formally define the mesh network topology, and we move on to discuss the new problems encountered in a mesh. Last, we present the self-stabilizing algorithm and proof.

3.1 Topology

A 2D mesh is an \( n \times m \) dimensional grid of processors. Let the \(-x\) direction be west, the \(+x\) direction be east, the \(-y\) direction be south, and the \(+y\) direction be north. For our purposes, we will model each bi-directional link in the network as a pair of unidirectional links. Each processor has a unique identifier, and each channel is assigned an identifier depending on the routing algorithm employed.

3.2 New Problems Encountered

In the ring topology, only faulty messages need to be removed from the network before the system can return to a legitimate state. There can only be one active sender at a time on the ring, and there is only one path that a flit can take to its correct destination.

The mesh topology offers many paths for a single flit to follow to its destination. Thus all mesh routing algorithms make heavy use of circuits. A circuit is a network path reserved for the body of a particular message by the header flit. A fault can cause invalid circuits to form in the network.
Much like wormhole messages, an invalid circuit can either be structurally invalid or logically invalid:

1. Structurally invalid circuits have at least one internal flaw that will prevent progress in the algorithm, since the routing code cannot function properly. Structural flaws include branching circuits, cyclical circuits, and broken circuits.

(a) *Branching* circuits contain processors with more than one outgoing channel reserved for a given incoming channel. Thus there is a *branch* in the circuit in at least one processor in the circuit. *Branched* circuits can cause flits to be scattered throughout the network, or it can cause flits to arrive at the destination processor out of order.

(b) *Cyclical* circuits are network paths that contain a closed loop. *Cyclical* circuits cause race conditions in which message flits move forever in the closed circuit.

(c) *Broken* circuits are those in which the circuit has one or more *holes* in it. A *hole* is a processor in a network path that no longer has an outgoing path for a particular message. Thus *broken* circuits are severed into two or more disjoint pieces before the entire message reaches the destination.

(d) *Stale* circuits are those in which the tail flit of a message is lost before it can completely clean up a circuit. The circuit can be partial or end to end complete.

2. Logically invalid circuits are structurally sound and complete, but they are constructed in such a way that a message can never reach its destination.

Among those logically invalid circuits, *Dead-End* circuits contain one or more *wrong turns* in the path. These *wrong turns* can cause the entire message to enter a processor that it cannot leave (due to channel id restrictions), thus never reaching its destination. For example, if a header flit moves one coordinate too far east, it can never move west again in the west-first routing algorithm.
3.3 Solutions and Ideas

In order to return to a legitimate global state, every faulty circuit must be torn down. We can deal with the incorrect circuits in the following manner:

**Broken and stale circuits.** They are the most difficult to deal with. Since a tail flit may never pass through these circuits, we must implement a timeout mechanism on each processor. If no new message flit is sent on an assigned outgoing channel for a sufficiently long period of time, then the channel lock will be cleared out.

**Cyclical and branching circuits.** They can be checked for whenever a packet is to be sent on an outgoing channel. Each one of these faults can be detected locally on a single processor.

A processor can detect a *branching* path while attempting to route a flit. If more than one outbound channel is assigned to the flit’s path then the circuit has a branch.

A *cyclical* circuit can be detected by a processor if the assigned route for a flit is an illegal turn. An illegal turn is one that is not allowed by the routing algorithm. In any case, the offending channel locks can be cleared out by the processor, thus severing the circuit into a *broken* circuit, which is discussed above.

**Dead-End circuits.** They can be resolved by destroying the header flit of the message when the routing algorithm determines that it needs to make an illegal turn to reach its destination. The other flits, including the tail will move into the dead end processor and eventually destroy the circuit normally.

3.4 Adaptive Wormhole Routing in a 2D mesh

Routing algorithms in a mesh can be *static* or *adaptive*. Static algorithms exhibit the same behavior regardless of network conditions (such as congestion). Adaptive algorithms will respond to adverse network conditions by sending packets through alternative routes. *Minimal* adaptive routing
requires an algorithm to send a packet along any of the shortest paths. Non-minimal adaptive routing algorithms may allow longer paths to be taken. Algorithms that lack the ability to route along every shortest path are partially adaptive.

For our purposes, we will model the West First routing algorithm presented in [Glass91]. The algorithm will statically route a message along the west axis until it reaches the destination x coordinate (if required). The algorithm will then adaptively route the packet north, east, and south if necessary. The channel numbering scheme shown in example Figure 3.1 and Figure 3.2 guarantees that the algorithm will route a packet along strictly decreasing channel numbers. Note that all outgoing channel numbers for west turns will have the highest numbers of all in each processor. No west routing is possible if a flit comes from a direction other than the east. Thus [Glass91] proves that West-First routing is deadlock-free.

3.4.1 Problem Specification

The problem specification for wormhole routing in a mesh is the same as in Chapter 2 Section 2.3. Namely, we will satisfy the self-stabilizing properties Convergence, Liveness, and Closure.

3.5 Model and Hypothesis

A wormhole routing processor in a mesh requires more hardware and software sophistication than a processor in a simple ring. Since each processor has at least two incoming and outgoing channels, complex decision making must be done in the routing code as to how and where to send a packet, shown in example Figure 3.3.

Every processor must have an input selection policy and an output selection policy determined by the underlying routing algorithm. When a processor receives flits on many incoming channels, the input selection policy determines which incoming message channel will be chosen to receive a flit. When more than one message is waiting for a single outgoing channel, the output selection policy determines which message will be chosen first. We will assume that the input selection policy is
Figure 3.1: West-First Channel Numbering Scheme for an n X m Mesh
Figure 3.2: West-First Channel Numbering Example on a 3 X 3 Mesh
round-robin, and that the output selection policy is FIFO (the oldest message will be chosen first). In this manner we can guarantee that no message waits forever to be received or transmitted by a processor. This is important in order to prove that the algorithm behaves correctly.

3.6 Data Structures

This algorithm shares most of the data structures and variables in common with the ring algorithm. The differences are as follows:

- The network is a two-dimensional array of $m \times n$ processors connected in a mesh, rather than $n$-many processors connected in a ring. Each processor in the mesh has a unique identifier equal to its $<x, y>$ coordinate pair. Each processor is aware of the size of the mesh, and consequently its position in the mesh.

- With the exception of edge and corner processors, each Processor $P$ has four incoming and four outgoing channels. These channels are called incoming-north $P^{in}$, incoming-south $P^{is}$, incoming-east $P^{ie}$, incoming-west $P^{iw}$, outgoing-north $P^{on}$, outgoing-south $P^{os}$, outgoing-east $P^{oe}$, and outgoing west $P^{ow}$.

- We will assume that along any infinite execution, there will be infinitely many processors activated that will initiate a message on the network. To accommodate this, each processor has a local virtual channel $P^{iv}$ that allows the processor to initiate messages. To initiate a message, we will assume that a processor will send itself a legitimate message one flit at a time on $P^{iv}$.

- Since there are many simultaneous senders on the mesh network, it is no longer practical for any flit to contain a unique message identifier. Unique identifiers are too strong an assumption to make in a self-stabilizing environment. Since every channel lock at a processor must correspond to a unique message, we will set the channel lock to the name of an incoming channel. This variable is removed from every flit.
• Header flits no longer need a time to live (ttl) variable. The West-First routing algorithm is deadlock free. In addition to this, channel locks must be made in descending order, so each processor in the network can instantly determine and correct if it contains an invalid channel assignment that may lead to a deadlock. The time to live variable (ttl) in the header flit was added to allow the ring network to expire header flits that have circled the network too many times. Since the West-First routing algorithm both avoids and detects cycles, there is no need for this variable.

• Each header flit will have a timestamp variable. This timestamp is set when a header flit is waiting on an outgoing channel to unlock. This allows a router to fairly select the oldest waiting header flit for an outgoing channel.

• The ftotal variable has been removed from each processor. The variable was added to the ring algorithm to remove headerless flood messages from the network. In Section 3.4 we proved that network cycles are avoided in the West-First routing algorithm. We also know that a processor can detect an illegitimate turn that will lead to a cycle. Thus every data flit must eventually come to a processor where it will be discarded or delivered.

• Each incoming and outgoing channel for a processor is labeled with a hard-coded channel identifier (ID) calculated according to the west-first channel labeling algorithm. All virtual incoming channels along with the missing channels in edge or corner processors are assigned the maximum channel id (maxmid) constant. This allows initial west movements from a sender, and it prevents messages from being sent on non-existent links. The channel identifier (ID) for channel ch on Processor P is denoted by $P^{ch}.ID$.

• There is now a channel-lock variable for each of the four outgoing channels per processor. This variable contains the name of the local incoming channel that the corresponding outgoing channel is reserved for. The channel lock (Lock) variable on processor for channel ch on Processor P is denoted by $P^{ch}.Lock$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
• Much like the ring algorithm, each incoming channel has a Buffer variable that can hold a single flit, or <empty>, and a CTS variable for flow control. These variables are denoted as $P_{\text{ch}}^{\text{Buffer}}$ and $P_{\text{ch}}^{\text{CTS}}$ respectively. The CTS variable of an adjacent processor can be read from the corresponding outgoing channel connected to it.

• The dest variable in a flit is actually a pair of integers denoting the <X,Y> coordinates of the destination processor in the grid. The individual coordinates are referenced by Dest.x and Dest.y respectively. The domain of the dest variable is limited to the possible <X,Y> pairs of the 2d mesh.

• Parameterized Actions are used wherever possible to denote a specific incoming ($P^{\text{tx}}$) or outgoing ($P^{\text{ox}}$) channel. These are used to simplify the router code, and they can apply to any channel of the appropriate type. For example, the guard of a single action may reference $P^{\text{tx}}$ rather than four separate actions with one guard written for each of $P^{\text{tx}}, P^{\text{tx}}. P^{\text{ux}}, P^{\text{un}},$ or $P^{\text{ss}}$.

### 3.7 Algorithm

First we present each of the helper functions and what they do in Section 3.7.1. The code of the main program is in Section 3.7.2. Finally, in section 3.7.3 each action in the main program is briefly described.

#### 3.7.1 Helper Functions

This section will give code and descriptions of all helper functions.

All of the computation for the routing algorithm is performed when a header flit is received. A routing path does not exist for the message yet, so one must be constructed. A processor in an adaptive network will have two different kind of path actions for a header flit:

**CHECK-PATH** (presented as Algorithm 3.1) will first examine the lock variable on the specified output channel. If the lock variable is free, then that channel will be locked, the flit will be written
to the flit buffer, and the function will return true. If the channel is locked for another message, then the function will return false. CHECK-PATH allows adaptive routing, in that a processor will examine more than one outgoing channel to find a viable path.

**Algorithm 3.1 The CHECK-PATH function**

```
function CHECK-PATH
    (P**: incoming channel.
     P^: outgoing channel,
     hflit: header flit)
    returns BOOLEAN
begin
    /* Check if the proposed circuit is valid. */
    if P^: .ID < P**: .ID — »
        /* Check if the channel is open. */
        /* If so. then lock the channel and write the flit to the buffer. */
        if P^: .Lock = 0 — y
            P^: Buffer := hflit;
            P^: CTS := HIGH:
            return TRUE;
        /* Check if the circuit is already established. */
        /* If it is. then there has been a fault. */
        /* Drop the header flit and free the channel. */
        if P^: .Lock = P**: .Lock — y
            DISCARD hflit;
            FREE-CHANNEL P^:
            return FALSE:
        /* Check if the channel is locked. */
        if P^: .Lock != 0 ∧ P^: .Lock != P**: .Lock — y
            return FALSE:
        fi
        if P^: .ID ≥ P**: .ID — y
            DISCARD(flit)
        fi
end
```

**FORCE-PATH** (presented as Algorithm 3.2) will force a header flit to wait on the desired output channel until it is free. Essentially the program will set the processor local time on the header flit, and exit with the header flit still contained in the flit buffer. When the channel is unlocked later, the FREE-CHANNEL procedure will activate and send any waiting header flits.
Algorithm 3.2 The FORCE-PATH function

procedure FORCE-PATH

(P^iz: incoming channel.
P^oz: outgoing channel.
hflit: header flit)

begin
/* Check if the proposed circuit is valid. */
if P^oz.ID < P^iz.ID →

/* Check if the channel is open or locked. */
/* If so, then lock the channel and write the flit to the buffer. */
if P^oz.Lock = 0 →

P^oz.Lock := P^iz;
hflit.time := <LOCAL-PROCESSOR-TIME>;
P^iz.Buffer := hflit;
P^iz.CTS := HIGH;
/* Check if the circuit is already established. */
/* If it is, then there has been a fault. */
/* Drop the header flit and free the channel. */
P^oz.Lock = P^iz →

DISCARD hflit:
FREE-CHANNEL P^oz
/* If the channel is locked, then write the flit to the */
/* buffer and wait for a FREE-CHANNEL call. */
P^oz.Lock != 0 ∧ P^oz.Lock != P^iz →

hflit.time := <LOCAL-PROCESSOR-TIME>;
P^iz.Buffer := hflit;
P^iz.CTS := HIGH:
fi

P^oz.ID ≥ P^iz.ID →
DISCARD hflit:
fi
end
**PROCESS-BODY** (presented as Algorithm 3.3) Every body flit (data or tail flit) that arrives at a processor will either already have an outgoing channel assigned to it, or it needs to be delivered to the local processor. There are two phases to processing a body flit in this self-stabilizing algorithm:

- The Circuit Discovery phase involves finding a valid existing output channel that is reserved in the virtual circuit for the message body. If there are multiple assigned outbound channels for a given input channel, then there is a faulty virtual circuit for that message. The processor will then call the FREE-CHANNEL procedure on each of those channels to remove them from a faulty virtual circuit. If there is no circuit out of a processor for a body flit, then the flit will be discarded.

- The Sending phase delivers the flit or writes the flit to the input channel Buffer variable. If the existing circuit is invalid (an *wrong turn*), or the last flit sent is a tail flit then the procedure will tear that part of the circuit down by calling FREE-CHANNEL on the output channel.

**FREE-CHANNEL** (presented as Algorithm 3.4) allows circuit destruction. This procedure will clear out the channel lock (Lock) variable of a specified output channel. If there are any header flits waiting in incoming channel flit buffers for this outgoing channel, then the router will fairly select the next header flit with the lowest *timestamp* value. The processor will set the channel lock variable to the selected incoming channel. Thus a SEND action will activate and transmit the header flit on the outgoing channel.

### 3.7.2 Main Program

The main body of the program contains only the adaptive routing code. If possible, a flit will be statically routed west first. If the flit can go east, then it can be adaptively routed in any direction...
Algorithm 3.3 The PROCESS-BODY function

procedure PROCESS-BODY
  (\(P^{ix}\) : incoming channel.
  \(\text{flit}\) : body or tail flit)
var: \(P^{oz}\) : outgoing channel pointer
begin
  /* Check if the circuit is faulty (multiple outbound channels). */
  if <the count of all outgoing channel locks for \(P^{ix}\) > 1 —
  - Tear down the branch in the circuit and discard the flit.
  do while exists (\(P^{oz}.\text{Lock} = P^{ix}\))
  FREE-CHANNEL \(P^{oz}\):
  od
  DISCARD flit:
  /* Check if the circuit is valid and write the flit to the buffer. */
  if \(P^{oz}.\text{ID} < P^{ix}.\text{ID}\) —
  \(P^{ix}.\text{Buffer} := \text{flit};
  P^{ix}.\text{CTS} := \text{HIGH};
  \(P^{oz}.\text{ID} \geq P^{ix}.\text{ID}\) —
  DISCARD flit:
  FREE-CHANNEL \(P^{oz}\);
  fi
end:
Algorithm 3.4 The FREE-CHANNEL function

procedure FREE-CHANNEL
  \( (P^{oz}: \text{outgoing channel}) \)
begin
  /* Free the channel lock. */
  \( P^{oz}.\text{Lock} := 0; \)
  /* Get a new header to send. */
  if \( P^{oz} = P^{ow} \)
    <Pick the oldest waiting hflit that can go west>:
    \( P^{oz}.\text{Lock} := P^{iz} \text{ of the hflit (or 0 if none)}; \)
  \( P^{oz} = P^{oe} \)
    <Pick the oldest waiting hflit that can go east>:
    \( P^{oz}.\text{Lock} := P^{ie} \text{ of the hflit (or 0 if none)}; \)
  \( P^{oz} = P^{os} \)
    <Pick the oldest waiting hflit that can go south only>:
    \( P^{oz}.\text{Lock} := P^{iz} \text{ of the hflit (or 0 if none)}; \)
  \( P^{oz} = P^{on} \)
    <Pick the oldest waiting hflit that can go north only>:
    \( P^{oz}.\text{Lock} := P^{ie} \text{ of the hflit (or 0 if none)}; \)
end:
other than west. If the flit can only go north or south, then an adaptive west movement is possible if the packet has originated at that processor.

The main program (presented as Algorithm 3.5) consists of three sets of actions:

1. **RECV** actions (presented as Algorithm 3.6) are activated when a flit is received on an incoming channel.
2. **SEND** actions (presented as Algorithm 3.7) are activated when a processor is able to transmit a flit buffer.
3. **ERROR** actions (presented as Algorithm 3.8) are activated when a local error condition is detected.

### 3.7.3 Algorithm Description

The main program (presented as Algorithm 3.5) has a total of eleven actions. There are three receive actions and three send actions that allow a processor to read and transmit header, data, and tail flits. There are five error correcting actions that allow the network to recover from transient faults. The primary functions of the 11 actions are described as follows:

1. Receive actions are formally presented as Algorithm 3.6 and are described below:

   **R1** The action (R1) is responsible for receiving header flits and performing the actual routing decisions. The first action performed is to clear out all channel locks that already exist for the incoming channel of the header flit (this is done to prevent branching). Once all of these errors are cleared, the flit will be checked to see if it has arrived at its destination. If the flit has not arrived, then it requires routing as follows:

   - If the header flit needs to go west, then static routing is applied. The result of this is that eventually the header flit will be routed west or discarded. The router will check if the packet is able to move west by comparing the incoming and outgoing
Algorithm 3.5 Self-stabilizing Wormhole routing on 2D Mesh (Main program)

program Self-Stabilizing West-First Routing Algorithm
process p [ i: 0..m-1 ][ j: 0..n-1 ]
inputs maxchannelid.

\[
\begin{align*}
P^{in}.ID & : \{0..\text{maxchannelid}\}, \\
P^{ne}.ID & : \{0..\text{maxchannelid}\}, \\
P^{se}.ID & : \{0..\text{maxchannelid}\}, \\
P^{nw}.ID & : \{0..\text{maxchannelid}\}, \\
P^{sw}.ID & : \{0..\text{maxchannelid}\}.
\end{align*}
\]

\[
\begin{align*}
\text{var} & \quad P^{on}.Lock: \{0, P^{ne}, P^{in}, P^{se}, P^{nw}, P^{sw}\}, \\
\text{var} & \quad P^{or}.Lock: \{0, P^{ne}, P^{in}, P^{se}, P^{nw}, P^{sw}\}, \\
\text{var} & \quad P^{os}.Lock: \{0, P^{ne}, P^{in}, P^{se}, P^{nw}, P^{sw}\}, \\
\text{var} & \quad P^{ow}.Lock: \{0, P^{ne}, P^{in}, P^{se}, P^{nw}, P^{sw}\}, \\
\text{var} & \quad P^{in}.Buffer: \{<\text{empty}>, \text{hflit}, \text{dflit}, \text{tflit}\}, \\
\text{var} & \quad P^{ne}.Buffer: \{<\text{empty}>, \text{hflit}, \text{dflit}, \text{tflit}\}, \\
\text{var} & \quad P^{se}.Buffer: \{<\text{empty}>, \text{hflit}, \text{dflit}, \text{tflit}\}, \\
\text{var} & \quad P^{nw}.Buffer: \{<\text{empty}>, \text{hflit}, \text{dflit}, \text{tflit}\}, \\
\text{var} & \quad P^{sw}.Buffer: \{<\text{empty}>, \text{hflit}, \text{dflit}, \text{tflit}\}.
\end{align*}
\]

\[
\begin{align*}
P^{in}.CTS & : \{\text{HIGH}, \text{LOW}\}, \\
P^{ne}.CTS & : \{\text{HIGH}, \text{LOW}\}, \\
P^{se}.CTS & : \{\text{HIGH}, \text{LOW}\}, \\
P^{nw}.CTS & : \{\text{HIGH}, \text{LOW}\}, \\
P^{sw}.CTS & : \{\text{HIGH}, \text{LOW}\}.
\end{align*}
\]

\[
\begin{align*}
\text{var} & \quad p^{in}: \{p^{ne}, p^{in}, p^{se}, p^{nw}, p^{sw}\}, \\
\text{var} & \quad p^{ne}: \{p^{ne}, p^{in}, p^{ne}, p^{nw}, p^{sw}\}.
\end{align*}
\]

begin
\[
\begin{align*}
\text{RECEIVE actions (Presented as Algorithm 3.6)} \\
\text{SEND actions (Presented as Algorithm 3.7)} \\
\text{ERROR CORRECTION actions (Presented as Algorithm 3.8)}
\end{align*}
\]
end:
Algorithm 3.6 Self-stabilizing Wormhole routing on 2D Mesh (Receive actions)

/* Receive a header flit. */
(R1) RECv hflit(time, dest) from \( P^{1z} \) ->
/* Free any channels locked for this channel. */
/* There should not be any since a header was received. */
do while exists \( (P^{1z}.Lock = P^{1z}) \)
    FREE-CHANNEL \( P^{1z} \);
od
/* Check if the flit has reached the destination. */
if dest.x = i \& dest.y = y ->
    DELIVER(hflit(time, dest));
/* Check if the flit can travel west-first. */
if dest.x < i ->
    FORCE-PATH(\( P^{1z} \), \( P^{0w} \), hflit);
/* Check if the flit can travel east at all. */
if dest.x \geq i ->
    if CHECK-PATH(\( P^{1z} \), \( P^{0e} \), hflit) \lor CHECK-PATH(\( P^{1z} \), \( P^{0n} \), hflit)
        \lor CHECK-PATH(\( P^{1z} \), \( P^{0s} \), hflit) ->
        EXIT:
    fi
    FORCE-PATH(\( P^{1z} \), \( P^{0e} \), hflit);
/* Check if the flit can travel north only. */
if dest.y > j \& dest.x = i ->
    if CHECK-PATH(\( P^{1z} \), \( P^{0n} \), hflit) \lor CHECK-PATH(\( P^{1z} \), \( P^{0w} \), hflit) ->
        EXIT:
    fi
    FORCE-PATH(\( P^{1z} \), \( P^{0n} \), hflit);
/* Check if the flit can travel south only. */
if dest.y < j \& dest.x = i ->
    if CHECK-PATH(\( P^{1z} \), \( P^{0s} \), hflit) \lor CHECK-PATH(\( P^{1z} \), \( P^{0w} \), hflit) ->
        EXIT:
    fi
    FORCE-PATH(\( P^{1z} \), \( P^{0s} \), hflit);
fi
/* Receive a data flit. */
(R2) RECv dflit(dat) from \( P^{1z} \) ->
    PROCESS-BODY(\( P^{1z} \), dflit):
/* Receive a tail flit. */
(R3) RECv tflit(dat) from \( P^{1z} \) ->
    PROCESS-BODY(\( P^{1z} \), tflit):
fi
channel identifiers according to the West-First channel numbering scheme (channel assignments must be made in decreasing order). If the flit is unable to move west, then it is discarded. Otherwise the flit is written to the incoming channel Buffer variable. If the west outgoing channel is available, then it will be locked for the incoming channel of the header flit. Otherwise the header flit will wait in the Buffer variable for a FREECHANNEL function call to select it.

- If the header flit is able to move east, adaptive routing is possible. The routing algorithm will examine the outgoing channels $P^{\text{oe}}$ (east), $P^{\text{on}}$ (north), and $P^{\text{os}}$ (south) in that order to determine if it is possible to route the flit along that channel (according to the West-First channel numbering scheme) and if the channel is available. The first available channel that the header flit can be routed along is locked. If no outgoing channel is currently available, then the flit is written to the Buffer variable in order to wait on the east channel.

- If the header flit only needs to move north, then the adaptive routing algorithm will examine the outgoing channels $P^{\text{on}}$ (north) and $P^{\text{ow}}$ (west) to determine if they are available and if it is possible to route the flit along that channel according to the West-First channel numbering scheme as in (R1). The first available channel that the header flit can be routed along is locked. If no outgoing channel is currently available, then the flit is written to the Buffer variable in order to wait on the east channel.

- If the header flit only needs to move south, then the adaptive routing algorithm will work the same as in the north case, except that the outgoing channels $P^{\text{on}}$ (south) and $P^{\text{ow}}$ (west) are examined.

R2 The action (R2) is responsible for receiving and processing data flits. When a data flit is received on an incoming channel $P^{\text{iz}}$, the processor searches for an outgoing channel locked for that incoming channel. If there are no outgoing channels locked, then the
data flit is delivered. If there are multiple channels locked for the incoming channel, the branches are destroyed and the flit is discarded. Lastly, if there is only a single outgoing channel locked for the incoming channel, the channel lock assignment is double checked for West-First correctness as in (R1) before the data flit is be written to the Buffer variable.

R3 The action (R3) is responsible for receiving and processing tail flits, but is otherwise functionally equivalent to (R2).

2. Send actions are formally presented as Algorithm 3.7 and are described below:

Algorithm 3.7 Self-stabilizing Wormhole routing on 2D Mesh (Send actions)

```c
/* Can send a header flit. */
(S1) if P..CTS = LOW \& P..Lock = P..Buffer = hflit(time, dest) →
    SEND hflit(time, dest) to P..
    P..CTS := LOW;
/* Can send a data flit. */
(S2) if P..CTS = LOW \& P..Lock = P..Buffer = dflit(dat) →
    SEND dflit(dat) to P..
    P..CTS := LOW;
/* Can send a tail flit. */
(S3) if P..CTS = LOW \& P..Lock = P..Buffer = tflit(dat) →
    SEND tflit(dat) to P..
    FREE-CHANNEL P..
    P..CTS := LOW;
```

S1 The action (S1) allows a processor to transmit a header flit.

S2 The action (S2) allows a processor to transmit a data flit.

S3 The action (S3) allows a processor to transmit a tail flit. This action will also free the outgoing channel lock for the message.

3. Error correction actions are formally presented as Algorithm 3.8 and are described below:

E1 The action (E1) will correct the local error condition in which a channel assignment in a processor violates the West-First channel lock requirements. All channel reservations
Algorithm 3.8 Self-stabilizing Wormhole routing on 2D Mesh (Error correction actions)

/* This action will correct an invalid channel lock. */
(E1) if $P_{oz}.\text{Lock} = P_{iz} \land P_{oz}.\text{ID} \geq P_{iz}.\text{ID}$
  $P_{iz}.\text{Buffer} := \text{empty};$
  $P_{iz}.\text{CTS} := \text{LOW};$
  FREE-CHANNEL $P_{oz};$

/* This action will correct a local CTS fault condition. */
(E2) if $P_{iz}.\text{CTS} = \text{HIGH} \land P_{iz}.\text{Buffer} = \text{empty}$
  $P_{iz}.\text{CTS} := \text{LOW};$

/* This action will remove unroutable header flits from the network. */
(E3) if $P_{iz}.\text{Buffer} = \text{hflit(time, dest)}$
  $\wedge ((P_{iz}.\text{Buffer}.\text{dest}.x < P.i \land P_{oz}.\text{ID} \geq P_{iz}.\text{ID})$
  $\lor (P_{iz}.\text{Buffer}.\text{dest}.x = P.i \land P_{iz}.\text{Buffer}.\text{dest}.y < P.j$
  $\land P_{oz}.\text{ID} \geq P_{iz}.\text{ID} \land P_{oz}.\text{ID} \geq P_{iz}.\text{ID})$
  $\lor (P_{iz}.\text{Buffer}.\text{dest}.x = P.i \land P_{iz}.\text{Buffer}.\text{dest}.y > P.j$
  $\land P_{oz}.\text{ID} \geq P_{iz}.\text{ID} \land P_{oz}.\text{ID} \geq P_{iz}.\text{ID})$
  $P_{iz}.\text{Buffer} := \text{empty};$
  $P_{iz}.\text{CTS} := \text{LOW};$

/* This action will remove unroutable body flits from the network. */
(E4) if $P_{iz}.\text{Buffer} = \text{dflit(dat)} \lor P_{iz}.\text{Buffer} = \text{tflit(dat)}$
  $\land \exists P_{oz}(P_{oz}.\text{Lock} = P_{iz})$
  $P_{iz}.\text{Buffer} := \text{empty};$
  $P_{iz}.\text{CTS} := \text{LOW};$

/* This action will correct broken and stale circuits. */
(E5) if TIMEOUT $P_{oz}.\text{Lock} = P_{iz} \land P_{iz} = Q_{oz} \land <Q, P> \in C$
  $\land \forall \text{edges} <a, b> \in C. (\Sigma(\text{flit}\#\text{ch.a.b}) = 0)$
  $\Rightarrow \text{FREE-CHANNEL } P_{oz};$

fi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
must be made in a strict decreasing order. If left unchecked, this error can cause the network to deadlock. The error is corrected by releasing the faulty channel lock.

**E2** The action (E2) will correct the local error condition in which an incoming channel Buffer variable is empty, but the CTS variable is set to HIGH (not ready to receive). The error is corrected by setting the CTS variable to LOW.

**E3** The action (E3) will correct the local error condition in which a processor contains a header flit in one of its Buffer variables that cannot be routed according to the West-First channel numbering scheme. The error is corrected by discarding the affected header flit.

**E4** The action (E4) will remove unroutable data and tail flits from the network. Data and tail flits are unroutable if they are written to the Buffer variable of a processors' incoming channel to be transmitted later, but there is no outgoing channel lock assignment for them. Data and tail flits are never selected for routing by the FREE-CHANNEL function. The error is corrected by discarding the affected flit.

**E5** The action (E5) is a TIMEOUT action that will correct broken or stale circuit fragments. An outgoing channel that has not sent data for a long time will activate this action to release the stale channel lock.

### 3.8 Proof of Correctness

Our proof of correctness for this algorithm will follow the model in the last chapter. First, all of the system legitimacy predicates are formally described in Section 3.8.1. Second, we proceed to prove closure in Section 3.8.2 and convergence in Section 3.8.3.

#### 3.8.1 Legitimacy Predicates

In the previous chapter, the ring algorithm had two different sets of legitimacy predicates, one set for the network processors, and one set for the network messages. The mesh algorithm has both
of these kinds of legitimacy predicates, with the addition of circuit predicates. Formally, the mesh network is in a legitimate state if the conjunction of all network, message, and circuit predicates are satisfied.

3.8.1.1 Processor Predicates

The following passage predicates can be resolved at each individual processor. Each processor is in a legitimate state iff each of the following predicates are satisfied. Remember that $G = \{V, E\}$ such that $V$ is the set of all processors, and $E$ is the set of all channels.

P1 For every processor, all outgoing channel locks must be valid according to the West-First numbering scheme. $\forall P \in V, \{P_{ox}: Lock = P_{ix} \land P_{ix}.ID \geq P_{ox}.ID\} = \emptyset$.

P2 For every processor, all incoming channels with a HIGH CTS value must have a flit in their buffer. This prevents a deadlock caused by a CTS variable corruption. $\forall P \in V, \{P_{lx}: CTS = \text{HIGH} \land P_{lx}.Buffer = \text{<empty>}\} = \emptyset$.

P3 For every processor, all incoming channels must have a unique output channel locked for it or no locked output channel locked at all. A corrupted Lock variable can cause a branching circuit. $\forall P_{ix} \in E, (\exists P_{ox}: Lock = P_{ix}) \lor (\neg \exists P_{ox}: Lock = P_{ix})$.

P4 For every processor, no incoming channel Buffer variable can ever contain an unroutable header flit. The disjunction of the following conditions indicate the presence of unroutable header flits:

- If a header flit needs to go west, but it cannot legally move west then the flit is unroutable. $\forall P \in V, \{P_{ix}: Buffer = \text{hflit}(\text{time}, \text{dest}) \land P_{ix}.Buffer.dest.x < P.i \land P_{ox}.ID \geq P_{ix}.ID\} = \emptyset$.

- If a header flit needs to go south only, but it legally cannot move south or west then the flit is unroutable. $\forall P \in V, \{P_{ix}: Buffer = \text{hflit}(\text{time}, \text{dest}) \land P_{ix}.Buffer.dest.y < P.j \land P_{ox}.ID \geq P_{ix}.ID \land P_{ow}.ID \geq P_{ix}.ID\} = \emptyset$.
• If a header flit needs to go north only, but it legally cannot move south or west, then the flit is unroutable. \( \forall P \in V, \{P^{12}, P^{12}.\text{Buffer}.\text{dest}.x = P.i \land P^{12}.\text{Buffer}.\text{dest}.y > P.j \land P^{12}.\text{ID} \geq P^{12}.\text{ID} \land P^{12}.\text{ID} \geq P^{12}.\text{ID} \} = \emptyset. \)

**P5** For every processor, no incoming channel Buffer variable can contain a data or tail flit unless there is an outgoing channel locked for it. \( \forall P^{12} \in E, (P^{12}.\text{Buffer} = \text{dflit(dat)} \lor P^{12}.\text{Buffer} = \text{tflit(dat)}) \Rightarrow \exists P^{12}(P^{12}.\text{Lock} = P^{12}). \)

### 3.8.1.2 Message Predicates

Much like the ring topology, the mesh network cannot contain structurally incorrect messages in a legitimate state. A message is structurally correct if and only if:

**M1** A message is constructed with a header flit, one or more data flits, and a tail flit. The difficulty with this predicate is that many messages will not have all of their flits on the network at one time. A header flit and multiple data flits may have been legitimately delivered to the destination while a tail flit remains on the network. A header flit may be on the network, while data flits and the tail flit wait to be transmitted. To get all of the flits for a single message \( M \), for a path \( C \) take:

\[
(\text{flits delivered}) \cup (\text{flits in transit, reverse-ordered in } C) \cup (\text{flits not yet transmitted}).
\]

*For example:* Let \( H \) be a header flit, \( D_1 \) be a data flit, and \( T \) be a tail flit:

**RECEIVER PROCESSOR** Delivered: \( H \)
**INTERMEDIATE PROCESSOR 2 Flit Buffer:** \( D_1 \)
**INTERMEDIATE PROCESSOR 1 Flit Buffer:** \( D_2 \)
**SENDER PROCESSOR Application Buffer:** \( T \)

The correct message is: \( H, D_1, D_2, T \)

### 3.8.1.3 Circuit Predicates

Circuit predicates are a new concept in this chapter. In a ring algorithm, there is no real need to strongly enforce circuits in this manner, since every processor has a single incoming and
a single outgoing channel. While a processor in a ring may only participate in a single circuit, a mesh processor can participate in up to four, one for each outgoing channel. The following are the legitimacy predicates for network circuits.

**C1** Formally, a structurally legitimate circuit $C$ is a finite set of incoming and outgoing channels well-ordered by the relation $R$, such that $X R M$ iff there is a path from $X$ to $M$.

Formally, $R$ is defined as follows:

- We define $P^{iz} R P^{oz}$ on a Processor $P$ iff $P_{lock}$ = $P^{iz}$.
- We define $P^{oz} R Q^{iz}$ iff $P^{oz}$ and $Q^{iz}$ are the same physical unidirectional link $< P, Q >$
- The relation $R$ is reflexive. $P^{ch} R P^{ch}$.
- The relation $R$ is transitive. $P^{iz} R P^{oz} \land P^{oz} R Q^{iz} \Rightarrow P^{iz} R Q^{iz}$.
- The relation $R$ is antisymmetric. $P^{ch} R Q^{ch} \land Q^{ch} R P^{ch} \Rightarrow P^{ch} = Q^{ch}$.

An example of a legitimate circuit $C_1 = \{ P^{iw}_1, P^{ow}_1, P^{ie}_1, P^{os}_1, P^{in}_1 \}$

Thus the relation $R$ (a set) on the example circuit $C_1$ with five channels will look like:

\[
\begin{align*}
\langle P^{iw}_1, P^{ow}_1 \rangle & \text{ - a channel lock} \\
\langle P^{ow}_1, P^{ie}_1 \rangle & \text{ - a physical link} \\
\langle P^{ie}_1, P^{os}_1 \rangle & \text{ - transitive} \\
\langle P^{os}_1, P^{in}_1 \rangle & \text{ - a channel lock} \\
\langle P^{ow}_1, P^{os}_1 \rangle & \text{ - transitive} \\
\langle P^{ie}_1, P^{os}_1 \rangle & \text{ - transitive} \\
\langle P^{os}_1, P^{in}_1 \rangle & \text{ - a physical link} \\
\langle P^{ow}_1, P^{in}_1 \rangle & \text{ - transitive} \\
\langle P^{ie}_1, P^{in}_1 \rangle & \text{ - transitive} \\
\langle P^{iw}_1, P^{iw}_1 \rangle & \text{ - reflexive} \\
\langle P^{ow}_1, P^{ow}_1 \rangle & \text{ - reflexive} \\
\langle P^{ie}_1, P^{ie}_1 \rangle & \text{ - reflexive} \\
\langle P^{os}_1, P^{os}_1 \rangle & \text{ - reflexive} \\
\langle P^{in}_1, P^{in}_1 \rangle & \text{ - reflexive}
\end{align*}
\]

**C2** A logically correct circuit $C$ is constructed from the sending channel of a source Processor $P \in V$, to a destination processor. An outgoing channel $P^{oz}$ and its corresponding incoming
channel $Q^{ch}$ are to C by the leading message header flit using the functions CHECK-PATH, FORCE-PATH, and FREE-CHANNEL on Processor P. Informally, this predicate defines what is and what is not a well-constructed circuit. If a channel lock corruption causes two message circuits in a processor to switch paths, but the circuits still remain structurally correct, or if a set of processor faults creates a structurally correct circuit for a message, the network is in an illegitimate state until those logically faulty circuits are removed.

**C3** Once a channel $P^{ch}$ has been added to a circuit, it cannot be removed by any means other than action (S3). If a channel lock is lost at the head of a circuit C after the header flit has been delivered, the circuit is still structurally correct, but not logically correct. thus the network is in an illegitimate state until C is removed from the network.

**C4** No circuit can remain on the network permanently. If a circuit C remains on the network permanently, then the network may deadlock. Processors will be forced to wait forever for the resources held by the circuit to be released.

### 3.8.2 Correct Behavior

First we prove that when the network is in a legitimate configuration, then the algorithm behavior is correct. Only a transient fault can cause the network to enter an illegitimate state.

#### 3.8.2.1 Progress

The first step is to prove three important features of distributed routing algorithms:

**Lemma 3.1 (Deadlock)** Starting from a configuration that satisfies the legitimacy predicate, the network will not deadlock.

**Proof.** Deadlocks are caused when processors are waiting on resources that are never freed. The following situations cause resources to be unavailable forever:
• There is a cycle in the network channel dependency graph. The West-First routing algorithm is proven to be deadlock free since all channel assignments in a circuit are made in strict descending order. We assumed that the network is in a legitimate state, so the (C1) predicate holds for every circuit, and the (P1) predicate holds for every processor. If a circuit contains a cycle in it, then it violates the antisymmetric property of the well-ordering $R$ in (C1). The predicate (P1) is immediately violated by an invalid channel assignment.

• A CTS variable corruption causes the CTS variable for an incoming channel to be set to HIGH when there is no data in the flit buffer. There is no data to send, so the incoming channel is set to not ready to receive forever. We assumed that the network is in a legitimate state, so the predicate (P2) holds. Therefore, this fault cannot occur.

• There is an unroutable header flit in an incoming channel Buffer variable. The unroutable header flit can occupy a critical incoming channel forever. We assumed that the network is in a legitimate state, so the predicate (P4) holds. Therefore, this fault cannot occur.

• There is an unroutable data or tail flit in an incoming channel Buffer variable. The unroutable body flit can occupy a critical incoming channel forever. We assumed that the network is in a legitimate state, so the predicate (P5) holds. Therefore, this fault cannot occur.

• There is an outgoing channel Lock variable that will never be released. We assumed that the network is in a legitimate state, so the predicate (C4) holds. Eventually the circuit containing the channel resource will be removed from the network.

In any case, when the network is in a legitimate state, a deadlock cannot occur.

Lemma 3.2 (Starvation) Starting from a configuration that satisfies the legitimacy predicate, the network will not starve.

Proof. A processor is starved if it needs to send a message, but it is unable to do so due to the scheduler or because it is too busy routing other messages. We assumed a fair scheduler, so any
processor that wishes to send a message may do so whenever needed using its internal virtual channel with channel id = maxchannelid. This channel is included in the input selection policy round-robin scheme, so all of the flits inputs from the processor will be accepted and introduced onto the network.

Lemma 3.3 (Livelock) Starting from a configuration that satisfies the legitimacy predicate, the network will not livelock.

Proof. The network is livelocked when processors are executing as normal, but no progress is made. No progress is made in the network when flits are routed but none are ever delivered, or if processors do not allocate resources fairly. We assumed a fair input selection policy as well as a FIFO output selection policy, so each processor will fairly receive flits and fairly allocate resources. If all message flits are routed and none are ever delivered, then there is a cycle in the network. West-first routing is deadlock free in a legitimate state, and in Lemma 3.1 we proved that deadlocks cannot occur in a legitimate state.

Theorem 3.1 (Progress) Starting from a configuration that satisfies the legitimacy predicate, the network will not deadlock, livelock, or starve.

3.8.2.2 Reliable Delivery

Next we prove that every flit transmitted will be received, and that every flit received will be transmitted until it is delivered, and finally we show that every message circuit will be destroyed.

Lemma 3.4 (Reliability 1) Starting from a configuration that satisfies the legitimacy predicate, every flit sent to a processor is eventually received.

Proof. Assume for a contradiction that the network is in a legitimate state, and a Processor P with an true RECV action on a particular incoming channel \( \text{P}^{ch} \) will never be activated. The scheduler is fair, so given an infinite execution sequence, the Processor P will be selected by the scheduler an infinite number of times. Since the RECV action on \( \text{P}^{ch} \) can never be activated, there must
be an infinite sequence of other actions that are true in Processor $P$. There cannot be an infinite execution of SEND actions without an infinite set of corresponding RECV action activations to fill the flit buffer. Since we must have an infinite set of RECV action activations, and we assumed a round robin input selection policy, then the RECV action on the incoming channel $P^{ch}$ will be activated.

Lemma 3.5 (Reliability 2) Starting from a configuration that satisfies the legitimacy predicate, every flit received at a processor is eventually delivered or written to the local Buffer variable.

Proof. In the actions (R1), (R2), and (R3), there are three possible outcomes for a received flit: delivery, discard, or write to a flit buffer. Assume for a contradiction that the network is in a legitimate state, and a received flit is discarded. The RECV guarded statements call the procedures CHECK-PATH, PROCESS-BODY, and FORCE-PATH, which contain the only discard actions. Every discard statement is protected by a guard that checks for wrong turns, branching circuits, and cycles. Since we assumed that the network is in a valid state, those guards will never be activated so the flit cannot be discarded.

Lemma 3.6 (Reliability 3) Starting from a configuration that satisfies the legitimacy predicate, every processor having an incoming channel Buffer variable containing a flit eventually transmits this flit.

Proof. Assume for a contradiction that the network is in a legitimate state and there is a $P^{iz}.Buffer$ variable that contains a flit that will never be transmitted.

A SEND action requires three things to fulfill: the channel lock variable of an outgoing channel $P^{oz} = P^{iz}$, the CTS variable of the adjacent processor (read from the outgoing channel) must be set to LOW, and the flit buffer must contain a flit.

If the flit is a header flit, then by progress it will eventually be granted the outgoing channel resource $P^{oz}$ by a FREECHANNEL call. If the flit is a data or a tail flit, and the processor is in a legitimate state, we can assume that an outgoing channel $P^{oz}$ is already reserved for the flit.
Assuming that there are no cycles in the network, eventually $P^{ox}.CTS$ will be set to LOW, and the SEND action will be activated.

**Lemma 3.7 (Reliability 4)** Starting from a configuration that satisfies the legitimacy predicate, a transmitted flit is never misrouted.

**Proof.** The function CHECK-PATH will never route a header flit along an invalid turn. The channel id of the incoming channel and outgoing channel will always be compared for legitimacy. The function FORCE-PATH will do the same, as well as guarantee that a header flit cannot be written to the buffer if it requires an invalid turn.

The RECV action (R1) performs all of the adaptive routing. The FREE-CHANNEL function will only allow static moves.

- $(P^{ox}.Buffer.dest.x < P.i) \land (P^{ox}.ID \geq P^{iz}.ID)$. A header flit that requires a faulty west route cannot happen, since the action for an east route action will not activate on a flit that has reached its destination x-coordinate.

- $(P^{iz}.Buffer.dest.x = P.i) \land (P^{iz}.Buffer.dest.y < P.j) \land (P^{ox}.ID \geq P^{iz}.ID) \land (P^{ow}.ID \geq P^{iz}.ID)$. A header flit that has reached its destination x-coordinate, but cannot move west or south will never exist on the network. This can only happen if a header flit is routed too far north after the destination x coordinate is reached. (R1) would have delivered the header flit at the appropriate destination.

- $(P^{iz}.Buffer.dest.x = P.i) \land (P^{iz}.Buffer.dest.y > P.j) \land (P^{ox}.ID \geq P^{iz}.ID) \land (P^{ow}.ID \geq P^{iz}.ID)$. A header flit that has reached its destination x-coordinate, but needs to go north is unroutable if it cannot move west or north can never exist on the network. (R1) would have delivered the header flit at the appropriate destination.

**Lemma 3.8 (Reliability 5)** Starting from a configuration that satisfies the legitimacy predicate, a header flit will always reach its destination.
Proof. We proved in Lemma 3.7 that a header flit is never misrouted, so a flit can never return to the same processor in a legitimate configuration. There are only finitely many routing actions that can be performed in a mesh without a cycle. Thus there are only finitely many legitimate processors that can be traversed by a header flit. One of those legitimate processors is the destination, where the header flit is delivered.

Lemma 3.9 (Reliability 6) Starting from a configuration that satisfies the legitimacy predicate, every structurally correct circuit C will be destroyed.

Proof. This follows from (S3), since no tail flits are lost and since $R$ well-orders C, the tail flit will eventually reach the destination. The last action in (S3) tears down each channel lock passed through in C.

Theorem 3.2 (Reliability) Starting from a configuration that satisfies the legitimacy predicate, every flit sent to a processor is eventually received. Every flit received at a processor is eventually delivered or written to the local Buffer variable. Every processor having an incoming channel Buffer variable containing a flit eventually transmits this flit. No header flits are misrouted. Every header flit will reach its destination. Finally, every structurally correct circuit C is eventually destroyed.

Theorem 3.3 (Closure) Once the network is in a legitimate state, all new and structurally correct messages introduced on the network will eventually reach their destination.

Proof. This follows from both Theorem 3.1 and Theorem 3.2. In Theorem 3.1 we proved that the algorithm cannot be deadlocked or livelocked, and that new messages are always introduced onto the network. Then in Theorem 3.2 we proved that every one of those introduced messages is eventually delivered.

3.8.3 Convergence

Lastly, we prove that this algorithm will converge to a legitimate state from any arbitrary initialization in finite time. This is done in the same manner as in chapter two using the convergent stair
(or *attractor*) method. First we show that TRUE is an attractor of all processor predicates. Then we show that the processor predicates are attractors for the message and circuit predicates. Thus we prove that the conjunction of all predicates will eventually hold in the system, and the system is in a legitimate state.

### 3.8.3.1 Processor Predicates

First we prove that starting from an arbitrary configuration, all of the processor legitimacy state predicates will be satisfied in finite time.

**Lemma 3.10** (*P1*) *Starting from an arbitrary configuration, (P1) eventually holds.*

**Proof.** Assume that there is at least one invalid channel assignment in the network at Processor P. Let $P_{oz}.Lock = P_{oz}.ID \land P_{oz}.ID \geq P_{oz}.ID$. The action (E1) is enabled on P, and given the fair scheduler it will be activated. The actions of (E1) will set $P_{oz}.Buffer$ to <empty>. $P_{oz}.CTS$ to LOW, and then FREE-CHANNEL will be called on $P_{oz}$. FREE-CHANNEL will result in a valid channel assignment for $P_{oz}.Lock$.

**Lemma 3.11** (*P2*) *Starting from an arbitrary configuration, (P2) eventually holds.*

**Proof.** This is guaranteed by the (E2) action. Given the fair processor scheduler, (E2) will eventually be activated on P. The statements of the action (E2) will satisfy network predicate (P2).

**Lemma 3.12** (*P3*) *Starting from an arbitrary configuration, (P3) eventually holds.*

**Proof.** Assume that a Processor P has an input channel $P_{iz}$ such that two channels $P_{oz_1}$ and $P_{oz_2}$ have their Lock variable set to $P_{iz}$. Assume the worst possible case in that neither of these Lock variables violate (P1), so they will not be automatically corrected by other means. We have three possibilities that may happen at P to correct the circuit:

- The RECV action (R1) is activated. The first action in (R1) performed on channel $P_{iz}$ is to free all outgoing channel Lock variables equal to $P_{iz}$. Both $P_{oz_1}$ and $P_{oz_2}$ will be freed, and (P3) is satisfied.
- One of the RECV actions (R2) or (R3) is activated on \( P^{1z} \), meaning that a data or tail flit has been received. The SEND-BODY procedure will check for a branching circuit, and \( P \) destroys both \( P^{o21} \) and \( P^{o22} \) (there is no way for \( P \) to know which way the header went). Thus (P3) is satisfied.

- No RECV actions are performed at all on \( P^{1z} \), thus the TIMEOUT action (E5) will activate on each output channel in the branch, thus satisfying (P3).

Lemma 3.13 (P4) Starting from an arbitrary configuration. (P4) eventually holds.

Proof. The action (E3) deals specifically with header flits that cannot ever reach their destination with valid turns. More specifically \( P^{1z}.Buffer = hflit(time, dest) \) and:

- \( P^{1z}.Buffer.dest.x < P.i \land P^{ow}.ID \geq P^{1z}.ID \). A header flit needs to go west, but it cannot. In the west-first routing scheme, a flit is statically routed west first. After a move in any other direction, a flit can never be routed west again. This error can occur if a header flit is misrouted too far east.

- \( P^{1z}.Buffer.dest.x = P.i \land P^{1z}.Buffer.dest.y < P.j \land P^{ow}.ID \geq P^{1z}.ID \land P^{ow}.ID \geq P^{1z}.ID \). A header flit that has reached its destination x-coordinate, but needs to go south is unroutable if it cannot move west or south. This error can occur if a header flit is misrouted too far north.

- \( P^{1z}.Buffer.dest.x = P.i \land P^{1z}.Buffer.dest.y > P.j \land P^{on}.ID \geq P^{1z}.ID \land P^{ow}.ID \geq P^{1z}.ID \). A header flit that has reached its destination x-coordinate, but needs to go north is unroutable if it cannot move west or north. This error can occur if a header flit is misrouted too far south.

Therefore, the actions of (E3) set \( P^{1z}.Buffer = \text{empty} \), thus satisfying processor predicate (P4). \( \square \)

Lemma 3.14 (P5) Starting from an arbitrary configuration. (P5) eventually holds.
Proof. A data or tail flit is unroutable at a Processor $P$ if it is contained in an incoming channel Buffer variable, and $P$ does not have an outgoing channel lock for it (none of the SEND actions will ever evaluate true). Since $P$ has no concept of where to send the flit, it must be discarded. Action (E4) will be activated on Processor $P$ to facilitate this.

3.8.3.2 Message Predicates

Next we prove that starting from an arbitrary configuration, the message legitimacy predicate will be satisfied in finite time.

Lemma 3.15 (M1) Starting from an arbitrary configuration, $(M1)$ eventually holds.

Proof. The algorithm will deal with structurally incorrect messages in the following manner:

- Header fragments contain a header flit and zero or more data flits. This message fragment will route to their destination, leaving a stale circuit behind them. Both stale and broken circuits are handled in Lemma 3.16

- Headerless fragments cannot traverse the network forever as in the ring algorithm. Assuming that the network has no illegitimate turns, a data or a tail flit will be delivered to the processor of the last incoming channel in the circuit.

3.8.3.3 Circuit Predicates

Finally, we prove that starting from an arbitrary configuration, the conjunction of all circuit legitimacy predicates is eventually satisfied.

Lemma 3.16 (C1) Starting from an arbitrary configuration, $(C1)$ eventually holds.

Proof. Structurally invalid circuits are all circuits $C$, such that $C$ cannot be well ordered by $R$. If there is a hole or a branch in a circuit, $C$ cannot well order $R$ since there are at least two incomparable
processors in the circuit. Two processors that are incomparable in \( R \) have no path between them. Assume that there is a circuit fragment \( C = \{P_1^{\pi_x}, P_2^{\pi_x}, P_3^{\pi_x}, \ldots, P_{n}^{\pi_x}\} \). We can safely assume that a tail flit will not traverse this entire circuit (or else we are done). Since the circuit is a fragment, eventually no more flits will move across the outgoing channels (none can be introduced, since there is no path into \( P_{n}^{\pi_x} \)). The action \((E5)\) will be eventually activated on each outgoing channel in \( C \) and the circuit will be destroyed. Examples of branching, broken, and logically invalid circuits are provided in Figures 3.4 and Figure 3.5. Each circuit is eventually removed from the network by the algorithm.

Lemma 3.17 (C2) Starting from an arbitrary configuration. \((C2)\) eventually holds.

Proof. A logically invalid circuit is the result of multiple channel lock corruptions that result in a crossed or a dead end circuit. If a circuit satisfies \((C1)\), a tail flit can move from one end of the well-order to the other, and \((S3)\) will clear out the channel locks. If no tail flit exists on the circuit, then the circuit is stale, and handled the same as a circuit fragment in Lemma 3.16.

Lemma 3.18 (C3) Starting from an arbitrary configuration. \((C3)\) eventually holds.

Proof. This follows from 3.17, where we proved that logically invalid circuits are removed from the network.

Lemma 3.19 (C4) Starting from an arbitrary configuration. \((C4)\) eventually holds.

Proof. We proved in Lemma 3.16 that structurally invalid circuits are removed from the network. Lemma 3.17 proves that logically invalid circuits are removed from the network. Thus any circuit will be removed from the network in finite time.

Theorem 3.4 (Convergence) Once the network is in an illegitimate state, it will return to a valid state in finite time. We have proven that each of the predicates will hold after finite time. so the conjunction of the predicates will hold after a finite time.
Figure 3.4: Network Stabilization Part 1

1. M1 makes an invalid West First turn.
2. M2 makes a wrong turn, stranding a fit.
3. M3 branches.
4. The invalid turn is detected and destroyed.
5. The unruetable header fit is removed.
6. The branch is detected and destroyed.

Figure 3.5: Network Stabilization Part 2

1. M1 tail destroys the circuit.
2. M2 tail destroys the circuit.
3. M3 tail destroys the circuit.
4. Fragments remain on the network.
5. The fragments are eventually removed.
CHAPTER 4

CONCLUSIONS

We have proposed two self-stabilizing wormhole routing algorithms, one for a ring and one for a mesh topology. From any arbitrary network state, the algorithm will converge to a legitimate one in finite time, and the algorithm will remain in a legitimate state until a fault occurs. In a ring topology, our wormhole routing algorithm tolerates any type of faulty message. In a mesh topology, our algorithm not only tolerates faulty messages, but it also tolerates faulty circuits. In both algorithms messages may be lost while the network is in error, but messages will always be delivered in a legitimate state. Self-stabilization will allow distributed computations in a massively parallel multiprocessor system to continue in spite of faults rather than having to stop and start over.

Further research needs to be done in terms of the additional time complexity and bandwidth overhead associated with the self-stabilizing code for each algorithm. Other topologies such as the hypercube and torus ring should be studied. Since all algorithms presented in this thesis are unicast algorithms (meaning there is only one sender and receiver), further research needs to be done in self-stabilizing multicast wormhole routing algorithms.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada. Las Vegas

Anthony B. Kenitzki

Local Address:
4501 S. Maryland Parkway
Las Vegas, NV 89154

Home Address:
10705 Sapphire Visa
Las Vegas, NV 89144

Degrees:
Bachelor of Science. Computer Science. 1997
College of Engineering/University of Nevada. Las Vegas

Thesis Title: Self Stabilizing Wormhole Routing

Thesis Examination Committee:
Co-Chairperson. Ajoy K. Datta. Ph.D.
Co-Chairperson. Sébastien Tixeuil. Ph.D.
Committee Member. John T. Minor. Ph.D.
Committee Member. Kazem Taghva. Ph.D.
Graduate Faculty Representative. Douglas Burke. Ph.D.