Self-stabilizing wormhole routing in hypercubes

Isai Arasu Marichamy
University of Nevada, Las Vegas
SELF-STABILIZING WORMHOLE ROUTING IN HYPERCUBES

by

Isai Arasu Marichamy
Bachelor of Science
University of Madras, India
2002

A thesis submitted in partial fulfillment of the requirements for the

Master of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2004

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
INFORMATION TO USERS

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 bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send 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.
Thesis Approval
The Graduate College
University of Nevada, Las Vegas

April 26, 2004

The Thesis prepared by

Isai Arasu Marichamy

Entitled

SELF-STABILIZING WORMHOLE ROUTING IN HYPERCUBES

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

Master of Science in Computer Science

Sebastien Tixeuil
Examination Committee Co-Chair

Ajoy Kumar Datta
Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Enunciator

Examination Committee Member

M. Vann

Graduate College Faculty Representative
ABSTRACT

Self-stabilizing Wormhole Routing in Hypercubes

by

Isai Arasu Marichamy

Dr. Ajoy K. Datta, Examination Committee Chair
School of Computer Science
University of Nevada, Las Vegas

Dr. Sébastien Tixeuil, Examination Committee Co-chair
Université de Paris-Sud, France

Many computer applications today require computational capability far beyond what can be obtained from the fastest single processor computers available. Multicomputers are widely used for these applications. One of the most common multiprocessor architectures is the hypercube. Wormhole routing is an efficient technique used to communicate message packets between processors when they are not completely connected. Since, the number of communication channels in a hypercube increases logarithmically with size, deadlock can be easily avoided during routing of packets. In wormhole routing, the packets are further divided into smaller fragments, called flits, before being routed.

The concept of self-stabilization (introduced by Dijkstra in 1974) is a unified framework to achieve fault-tolerance in any system. As per Dijkstra's definition, a system is called self-stabilizing if starting from an arbitrary (including an undesirable) state, it is guaranteed to converge to a desired state (or a behavior) in finite number of steps. Transient faults such as link failures, process failures, memory corruption, etc. can take the system to an
illegitimate state. A self-stabilizing system recovers to a legitimate state or behavior in finite time without human intervention, and preserves legitimacy until another fault occurs.

To the best of our knowledge, this is the first attempt at designing a self-stabilizing wormhole routing algorithm for hypercubes. Our first algorithm handles all types of faults except for node/link failures. This algorithm achieves optimality in terms of routing path length by following only the preferred dimensions. In an $n$-dimensional hypercube, those dimensions in which source and destination address bits differ are called preferred dimensions. Our second algorithm handles topological changes. We propose an efficient scheme of rerouting flits in case of node/link failures. Similar to the first algorithm, this algorithm also tries to follow preferred dimensions if they are nonfaulty at the time of transmitting the flits. However, due to topological faults it is necessary to take non-preferred dimensions resulting in suboptimality of path selection. Formal proof of correctness for both solutions is given.
ACKNOWLEDGMENTS

First, I would like to thank Dr. Ajoy K. Datta for advising this thesis research. My special thanks go to Dr. Sébastien Tixeuil of Université de Paris-Sud, France for his directions throughout this work. I am grateful for the time and effort which he put in when I needed it. I would also like to thank Dr. Thomas Nartker, Dr. Wolfgang Bein, and Dr. Venkatesan Muthukumar for serving on my thesis committee. I am thankful to UNLV faculty and staff for their education and financial support throughout my graduate study.

This thesis is dedicated to my parents for their love and support. I would also like to thank all my friends for their affection and encouragement.
CHAPTER 1

INTRODUCTION

The need for large computational power is increasing everyday. This has resulted in a very high demand for parallel-processing systems. These multiprocessors compute by exchanging messages via communication links. Therefore, implementing the message passing mechanism should target maximizing the throughput and minimizing latency. Many types of interconnection networks have been proposed in the literature, e.g., ring, mesh, hypercube, torus, etc. In this thesis, we will deal with hypercubes. The routing refers to the procedure followed by the nodes of a network to choose one (or, sometimes, more) of their neighbors to forward messages towards their final destination. The wormhole routing is commonly used in parallel architectures in which each message is sent as small fragments, called flits. The first flit, called header flit, contains all the information for routing the entire message. As the header flit moves from the source towards the destination, the remaining flits follow in a pipelined fashion resembling a worm, hence the name wormhole routing.

Various types of faults can occur in multiprocessor systems. Memory of the nodes can be corrupted. Messages can get corrupted or lost. Nodes and links can fail or recover. Thus, designing fault-tolerant algorithms for multiprocessor networks like hypercubes is desirable. The paradigm of self-stabilization, introduced by Dijkstra in 1974 [27], is considered to be the most unified strategy to design fault-tolerant systems. Although it was intended to
handle transient faults, research has shown that all types of faults can be dealt with in a stabilizing manner.

1.1 Our Contributions

In this thesis, we will study the application of self-stabilization in designing a wormhole routing scheme for hypercubes. Similar studies have been conducted earlier, but only on much simpler topologies, such as rings and meshes. Our goal is to design the first self-stabilizing wormhole routing protocol in hypercubes. We propose an adaptive routing technique in the following sense. Each $n$-cube ($n$-dimensional hypercube) has $n$ dimensions. Every node (source or intermediate) examines the preferred dimensions in a particular order to decide which dimension to use in forwarding the current message, and uses the first free dimension to route the message. This process is followed by all the nodes on the routing path until the message arrives at the destination.

Our solution handles many types of faults, including variable corruptions and message losses. We will present two routing algorithms. The first algorithm computes a minimal path, but does not handle node/link failures. The second algorithm computes an alternate path in case of node/link failures, but the path may not be minimal.

1.2 Thesis Outline

In Chapter 2, we give an overview of some areas which are involved in this research, such as multicomputers, direct networks, wormhole routing, and self-stabilization. Chapters 3 and 4 include the main contribution of this thesis. In Chapter 3, we present an algorithm for self-stabilizing wormhole routing in hypercubes, followed by its proof of correctness. In Chapter 4, we have extended the algorithm given in Chapter 3 to accommodate node/link
failures and recoveries. In Chapter 5, we give concluding remarks and future directions.
CHAPTER 2

BACKGROUND CONCEPTS

In this chapter, we define many terminologies in the areas of parallel computing, interconnection networks, routing protocols, fault-tolerant computing, and self-stabilization. In Section 2.1, we define multicomputers, and in Section 2.2, we discuss direct networks. A brief discussion of routing protocols is included in Section 2.3. Section 2.4 defines wormhole routing, and lists its advantages and disadvantages. In Section 2.5, we give a short introduction to self-stabilization.

2.1 Multicomputers

Multicomputers are computing systems in which thousands of processors are connected in a parallel manner to achieve teraflops of computational power [18, 26]. Such large-scale multiprocessors are usually organized as ensembles of nodes, where each node can be viewed as a combination of a router and a processor with some RAM, bus, and I/O circuitry [10]. The processors can be graphics processors, vector processors, or symbolic processors. This type of architecture is called physically distributed memory parallel computer, and hence, multicomputers are called distributed-memory multiprocessors [56]. These processors need to be interconnected, and Direct Networks (refer Section 2.2) serve this purpose with many advantages [67].
2.2 Direct Networks

A *Direct Network* is a type of interconnection network model in which nodes send messages to one another through directed channels. Examples of direct network topologies include mesh, tree, and hypercube. Direct networks have emerged as a popular architecture for massively parallel computers because of their scalability. The total communication bandwidth, memory bandwidth, and processing capability of the system increase with the number of nodes. Neighboring nodes exchange messages directly while nodes that are not connected directly communicate by passing messages through intermediate nodes. Hence routing becomes inevitable. Different routing techniques can be employed depending upon the system specifications, resource requirements, network traffic, etc. Some of the routing algorithms commonly used in direct networks are discussed in Section 2.3. Examples of experimental and commercial systems based on direct networks include Intel’s iPSC, Touchstone Delta [26], Paragon [58], Ncube-2/3 [18], Cray T3D [47, 60], MIT J-Machine [57], and Stanford DASH [52].

Factors that characterize direct networks are:

**Topology**: The topology defines the model in which nodes are interconnected in a network. If every node is connected to every other node, then the network is said to be *complete*. Complete networks obviate forwarding of packets by intermediate nodes, but are practical only for very small networks. Hence *multihop* topologies [56] are widely used for direct networks rather than complete topologies. Next, we define some of the properties of a topology. *Bisection width* is the minimum number of channels required to divide the network into two equal halves. *Channel width* is the number of bits that a physical channel can transmit simultaneously between two adjacent nodes. Chan-
**nel rate** is the peak rate in bits/sec at which bits can be transferred over a channel. Topologies with low bisection width, high channel width, and high channel rate are preferred for better performance.

**Routing:** *Routing* is the process of forwarding packets from source to destination when they are not directly connected. The sequence of channels through which this forwarding is done is called a *path*, and the number of channels involved is called *path length*. Path length is critical in routing because it decides the time a packet stays in the network, which in turn affects the network traffic. Different types of routing techniques are briefly discussed in Section 2.3.

**Flow Control:** A network consists of many channels and buffers. *Flow control* deals with the allocation of channels and buffers to a packet as it travels along a path through the network. A good flow control policy should avoid channel congestion while reducing network latency. A routing algorithm determines which output channel is selected for a packet arriving on a given input channel. Therefore, routing may be referred to as the *output selection policy*. The *input selection policy* determines which input channel may use an available outgoing channel.

**Switching:** *Switching* is the actual mechanism that removes data from an input channel and places it on an output channel. We briefly define four different switching techniques used in multicomputer networks. In *store-and-forward* switching, when a packet reaches an intermediate node, the entire packet is stored in a buffer before being forwarded to a neighboring node. This technique is also called *packet switching*. In *virtual cut-through* [45], a packet is stored at an intermediate node only if the next required channel is busy. In *circuit switching*, a physical circuit is constructed.
between the source and destination before the actual packet is transmitted. All channels in this circuit are exclusively reserved until the transmission ends, and hence buffers can be eliminated. *Wormhole routing* is a type of switching that uses both virtual cut-through and circuit switching techniques and is discussed in more detail in Section 2.4.

2.3 Routing Protocols

Since nodes do not physically share memory, they must communicate by passing messages through the network. And routing of packets becomes unavoidable when the source and destination are not neighbors. The scheme used by a router to decide one channel, among many available channels, to forward a packet is called as a *routing protocol*. A routing protocol needs to be simple and robust [36] with low latency and high throughput. Different types of routing protocols are used depending upon the system specifications, router capabilities, processor memory, etc. We will briefly discuss the different types of routing protocols in this section. Since many of these routing protocols can be combined with one another it becomes hard to distinguish clearly the differences between some of these routing protocols. For literature on different routing protocols, see [63] and [38].

A routing algorithm is said to be *deterministic* or *oblivious* if its behavior is independent of current network conditions. For example, deadlock-free routing (refer Section 2.4.2) can be achieved by assigning each channel a unique number and allocating channels to packets in strictly ascending (or descending) order. This type of routing, called *dimension ordered routing*, is deterministic. Deterministic routing can be further classified into *source* (e.g., street-sign routing [56]) and *distributed* (e.g., e-cube routing [56]) routing. In the previous case the routing path is decided at the source node and in the later case it is decided
at every node along the path. The main disadvantage of deterministic routing is that it cannot respond to dynamic network conditions, such as congestion [22]. In adaptive routing, the router behaves according to the current network conditions. At each node the router allocates an outgoing channel to the incoming packet depending on the resources available at that instant, hence adaptive. This type is further classified into fully adaptive and partially adaptive routing, depending on the level of adaptiveness. Based on the length of the path taken from the source to destination, routing can be minimal or non-minimal. For example, in e-cube routing the path taken from the source to the destination is minimum and it can be non-minimal in case of west-first routing protocol.

2.4 Wormhole Routing

Wormhole routing is a cut-through approach to circuit switching. Although both virtual cut-through and circuit switching (refer Section 2.2) offer low network latencies that are relatively independent of path length, virtual cut-through requires that blocked packets be buffered, and circuit switching makes it difficult to implement virtual channels. Dally and Seitz [23] proposed wormhole routing to overcome these difficulties while offering similar network latency. In wormhole routing, message packets are broken into smaller fragments called flow control digits (or flits). It is a lightweight and efficient method of routing messages in a multiprocessor framework. 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 source to the destination like a worm.

2.4.1 Advantages of Wormhole Routing

1. Message flits need to be buffered only if no free channel is available for forwarding when they reach a node. This decreases the amount of time spent in transmitting data.

2. Buffer needs to be only few flits in size as opposed to whole packet and this reduces channel setup time. Message flits are only few bits in size and depends on system parameters, in particular the channel width.

3. Low latency and high throughput. Latencies for store-and-forward and virtual cut through are functions of product of packet length and distance to travel, while that for wormhole routing and circuit switching is a function of sum of packet length and distance to travel.

4. The concept of virtual channels can be incorporated to achieve adaptive routing. It is a technique in which a physical channel is divided into many channels to employ parallel transmission of flits between two routers.

5. Packet replication becomes easier, which is used in broadcast and multicast communications. Since the flit size is small, it can be duplicated faster and sent through different channels from a single router.

6. Ease of implementation in VLSI. Since all routing protocols need to be deployed in real-time systems using hardware chips, it is necessary for these systems to be practical.
7. Reduced dependency between latency and inter-node distance. Therefore, wormhole routing can be implemented even in a network with humongous number of nodes.

2.4.2 Overheads in Wormhole Routing

Since wormhole routing extensively uses packets (i.e., flits), it becomes an important issue to assure safe delivery of packets at the destination. Switching strategy and routing algorithm used are among several factors that affect communication latency of the network. Below are some of the situations that can postpone packet delivery indefinitely.

**Deadlock.** Deadlock can occur if packets are allowed to hold some resources (channels) while requesting others; a set of packets may become blocked forever. One way to solve the deadlock problem is to allow the preemption of packets involved in a deadlock situation, but this increases the latency, and hence not used. Another method that is more widely used to avoid deadlock is by ordering the resources and allocating them in a strictly monotonic order (e.g., dimension ordered routing). A channel dependence graph [24] can be used to develop a deadlock-free routing algorithm. The nodes of a channel dependency graph for a network is formed by its links and its vertices are formed by the pairs of channels connected by the routing function. Virtual channels can also be used to avoid deadlock.

**Livelock.** This problem arises whenever a packet can be denied delivery to its destination forever. Livelock may occur if nonminimal routing is allowed or if the input selection policy (refer Section 2.2) is unfair.

**Starvation.** A network is said to starve if some resources which a node is waiting for is never granted. This problem arises when a node wants to send a packet but is never allowed
to do so. For example, starvation can occur if a circuit setup attempt repeatedly encounters a blocked channel forcing the circuit to be torn down.

2.4.3 Wormhole Routing for Collective Communication

Communication in a wormhole-routed network environment may be point-to-point which involves a single source and a single destination, or collective, in which more than two nodes participate [54]. Nodes use collective operations to distribute, gather, and exchange data to perform global computations such as max, min, sum, etc. These computations are used in sorting, graph and search algorithms [49]. For the systems which do not support collective communications, they can be simulated by sending multiple point-to-point messages. These implementations are called unicast-based [55]. This strategy is often expensive, and hence most of the systems are designed to support collective communication of messages. The growing interest in collective operations is evidenced by their inclusion in Message Passing Interface (MPI) [1], an emerging standard for communication routines used by message-passing programs.

Collective communication can be broadcast, scatter, gather, all-to-all broadcast, or all-to-all scatter-gather. They are achieved by either hardware routing algorithms or software routing algorithms. Hardware routing algorithms include using multiple ports, virtual channels, and Intermediate Reception (IR) capability [54]. Software routing algorithms employ arranging of constituent messages, e.g., in a tree, so as to make better use of the underlying hardware features. Wormhole routing can be used for collective communication to increase the overall efficiency.
2.5 Self-stabilization

The concept of self-stabilization has been known for about thirty years as a paradigm of designing fault-tolerant systems. This concept was introduced to computer science by Dijkstra [28, 27] and later strongly endorsed by Lamport [50]. The idea of self-stabilization was used in other areas (such as numerical analysis, control theory, systems science, etc.) long before Dijkstra coined the word “self-stabilization”. Many definitions of self-stabilization exist in the literature, and unfortunately, the stabilizing research community did not converge on a single definition. One widely accepted definition is as follows: A self-stabilizing system, regardless of its initial state, converges in finite time to a set of states that satisfy its specification. It can also be defined with respect to behavior instead of state as follows: A self-stabilizing system, starting from an arbitrary state, reaches a state in finite time such that it starts behaving according to its specification. In this section, we will give an informal overview of some aspects of stabilization. Our goal is not to give a comprehensive summary of the whole area. Readers can refer to [41] for an almost current on-line bibliography of stabilizing literature, [29] for the only book on this topic, [33, 37, 59] for surveys of the area, and [6, 9] for an introduction to the concept of self-stabilization.

Although Dijkstra's work [27] did not mention any application of self-stabilization to fault-tolerance, there has been a lot of research on this topic. The dissertations [5, 48] are excellent sources on this topic. The self-stabilization was defined in terms of closure and convergence in [5, 7]. Closure refers to the property which requires that during all system executions, the system stays within some set of legal or desirable set of states unless a fault occurs. Convergence requires the system to reach a legal state from any arbitrary (possibly illegal) state in finite steps. A system is self-stabilizing if it satisfies both closure and
convergence properties. In [5, 7], a comprehensive study of different types of faults (such as crash, stuck-at, fail-stop, omission, timing, performance and Byzantine) and how they are accommodated in their definition of stabilization (in terms of closure and convergence) was included. The first formal definition of fault-tolerance was given in [7]. The results like [5, 7] and some others (refer [41]) in subsequent years establish the fact that self-stabilization is the most unified strategy of achieving fault-tolerance in distributed systems. Previous attempts were specific to some technologies, architectures, and applications. In [48], it was shown that a fault-tolerant program is a composition of a fault-intolerant program and a set of fault-tolerance components. A method for designing multi-tolerant programs (ones that tolerate multiple types of faults) was also presented in [48]. It was shown in [43] that a sequence of crashes can drive a protocol into arbitrary global states.

Self-stabilization has been extensively used in the area of network protocols. Numerous papers have been written on protocols like routing (including cut-through, wormhole), alternating bit, sliding window, session control, congestion control, connection management, high-speed networks, sensor networks, and max-flow computation. Refer to [29, 41] for the pointers. Many of these protocols also consider message losses and duplications, and node/link failures.

There exist many self-stabilizing distributed solutions for graph theory problems. Examples are different types of spanning trees, finding center and median, maximal matching, search structures, and graph coloring. Stabilization has been applied to solving many classical distributed algorithms. Examples include mutual exclusion, token circulation, leader election, synchronization and clocks, distributed reset, distributed diffusing computation, termination detection, and propagation of information with feedback. Problems have been
considered in different topologies (e.g., ring and tree) as well.

Numerous models have been considered in the literature. There exist several dimensions of the model, such as execution model (shared registers and message passing), fairness (weakly fair, strongly fair, and unfair), granularity of the atomic step (composite versus read/write atomicity), and types of daemons (central and distributed). Stabilization time complexity and space complexity have been two factors of the stabilizing solutions. Several optimal solutions have been proposed.

Proving correctness of a stabilization program is quite challenging. Two techniques have been commonly used in the literature: convergence stair [39] and variant function [46] methods. Proof techniques for randomized algorithms are discussed in [15, 32, 40, 42]. Many general methods of designing self-stabilizing programs have been proposed. We mention some of them here without any description: diffusing computation [8], silent stabilization [30], local stabilizer [4], local checking and local correction [11, 64], distributed program checking [12], counter flushing [65], window washing [19], self-containment [35], snap-stabilization [21], super-stabilization [31], power supply [2], and transient fault detector [14]. A brief survey of self-* systems is given in [68].
CHAPTER 3

SELF-STABILIZING WORMHOLE ROUTING IN HYPERCUBES

In this chapter, we present an adaptive, minimal, and self-stabilizing wormhole routing algorithm for hypercubes. The roadmap of this chapter is as follows: In the next section, we state the motivation of this research. In Section 3.2, we discuss some of the fault-tolerant routing techniques used in wormhole routing, and present self-stabilizing wormhole routing algorithms for rings and meshes. In Section 3.3, we first state the model used in writing the algorithm. The program (including its notations) used is reported next. We also give a formal definition of self-stabilization in this section. Finally, in this section, we give the specification of the problem solved in this chapter. We list the various types of problems encountered and their solutions in Sections 3.4 and 3.5, respectively. The actual routing scheme used to solve the problem is informally explained in Section 3.7. In Section 3.9, the wormhole routing algorithm (Algorithm \textit{WRH}) is given. This section includes a detail informal description and the formal algorithm. Finally, the proof of the algorithm is given in Section 3.10.

3.1 Motivation

Many of today’s applications such as weather prediction, aerodynamics, and artificial intelligence are very computationally intensive and require vast amounts of processing power.
To cater to the needs of such applications, we need multicomputers which use large number of processors that execute in parallel. These parallel processors are interconnected, and need to communicate with each other to synchronize in processing of instructions. Since the time taken for communication between processors is very critical, it is important to choose the appropriate topology used to connect these processors. Hypercubes offer maximum efficiency with low latency. The number of links in a hypercube network increases logarithmically with its size. Hence, the chances of deadlock is much less in hypercubes. Since hypercubes are not completely connected graphs, the next issue is to route the message packets from a source to destination. It is easy to handle message packets of smaller size. Wormhole routing is the technique in which message packets are broken into much smaller fragments (called flits) at the source and are combined at the destination. The entire path from source to destination is decided by the header flit, and the remaining flits follow as a worm. The main advantage of wormhole routing is, there is no need to execute the routing algorithm for every message fragment. The process of routing can be interrupted by different types of faults in the system. These faults may result in message losses and false delivery of messages. This demands a robust routing protocol. We chose to use the concept of self-stabilization to implement the fault-tolerance in the routing protocol. As mentioned in Chapter 2, self-stabilization offers a unified framework to achieve fault-tolerance.

3.2 Related Work

Considerable research has been done in making wormhole routing robust (fault-tolerant). Virtual channels were added to the network to handle faults [22]. Virtual channels divide a single physical channel into many, sharing the bandwidth between them. An adaptive turn-based model was used to avoid faults in [34]. If a faulty processor is encountered
in 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 [53], meaning that one or more processors will cease to function entirely, while the remainder will faithfully execute their programs. Self-stabilizing network algorithms in virtual cut through setting were given in [13, 20] but not in a wormhole routing environment. Self-stabilizing algorithms for simpler topologies such as rings and 2D meshes can be found in [25] and [44] respectively.

Efficient algorithms were proposed for routing flits in the event of node and/or link failures. Algorithms for adaptive fault-tolerant routing in hypercube multicomputers are given in [16]. An improved version of this algorithm is given in [51]. Broadcasting in wormhole routed hypercubes is given in [66]. This algorithm uses the concept of broadcast subcube, which only uses local safety information. Deadlock is the most challenged problem when dealing with routing algorithms. To avoid deadlock, we use the concept of channel dependency graph [24] to investigate the proposed algorithms. Virtual channels are used to overcome deadlocks in message circuits. A fault tolerant routing algorithm for n dimensional hypercubes that can tolerate n/2 node faults with two virtual channels is given in [62]. A similar algorithm that can tolerate n − 1 node faults with five virtual channels is presented in [17]. The cost and complexity of hardware increase dramatically with the use of five virtual channels. A better algorithm that uses only three virtual channels is proposed in [61], which tolerates the same n − 1 faulty nodes.

### 3.3 Model and Preliminaries

#### 3.3.1 Model

**Network Topology.** The hypercube is the most efficient network used for communication between parallel processors. It can efficiently simulate any other network of the same
size. For example, the algorithms devised for mesh can be implemented in a hypercube by simulating mesh in a hypercube. One drawback of the hypercube structure is that the number of links to each node increases logarithmically with the size of the network.

The $n$-dimensional hypercube, called $n$-cube, has $2^n$ nodes and $n2^{n-1}$ links. All nodes have $n$ links, and each such bi-directional link is modeled as a pair of unidirectional links. Processors have unique identifiers. Channels are assigned identifiers based on the routing algorithm employed.

The network is modeled as a graph $G = \{V, E\}$, where $V$ is a set of processors in $0..2^n - 1$, $E$ is the set of bidirectional links, and $|E| = n2^{n-1}$. Each processor has a unique $n$-bit binary label.

**Program.** The *state* of a process is defined by the value of its variables. The processes represent nodes or routers. The *state* of a system is a vector of $n + 1$ components where the first $n$ represent the state of $n$ processes, and the last component refers to the set of messages in transit in the links. In the following, we refer to the state of a process and system as a *(local)* state and configuration, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\rightarrow$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a maximal sequence of configurations $e = \gamma_0, \gamma_1, \ldots, \gamma_t, \gamma_{t+1}, \ldots$, such that for $t \geq 0, \gamma_t \rightarrow \gamma_{t+1}$ (a single computation step), if $\gamma_{t+1}$ exists, or $\gamma_t$ is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this thesis are assumed to be maximal.

During a computation step, one of the following actions *(local steps)* occurs on at least one process $p$: (1) $p$ receives a message; (2) $p$ executes some internal actions; (3) $p$ sends
at least one message. The set of computations of a protocol $P$ in system $S$ starting with a particular configuration $\alpha \in C$ is denoted by $\mathcal{E}_\alpha$. The set of all possible computations of $P$ in system $S$ is denoted as $\mathcal{E}$.

Each action of a process is of the form:

\[
<\text{label}> <\text{guard}> \rightarrow <\text{statement}>
\]

\[
\ldots
\]

\[
<\text{statement}>
\]

The guard of an action in the program of a process $p$ is one of the following: a local guard of $p$ or a receiving guard of $p$. A local guard of $p$ is a boolean expression involving the variables of $p$. A receiving guard of $p$ is of the form:

\[
\text{rcv} <\text{message\_type}> \text{ from } <\text{sending\_channel\_name}>
\]

The statements of a process are of four types: assignment, sending, selection, and iteration. An assignment statement of $p$ is of the form: $x_p := E_p$ where $x_p$ is a variable of $p$ and $E_p$ is a constant or expression of the same type as $x_p$. A sending statement of $p$ is of the form: send $<\text{message\_type}>$ to $<\text{receiving\_channel\_name}>$.

A selection statement of $p$ is of the form: if \ldots fi. An iteration statement of $p$ is of the form: for \ldots endfor or do while \ldots od.

The statement of an action of $p$ updates one or more variables of $p$. When $p$ executes a statement, we say that “$p$ moves” or “$p$ executes an action”. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.
Self-stabilizing Program. Let \( \mathcal{L} \) be a predicate (called, *legitimacy predicate*) defined with respect to a specification (predicate) \( R \). An algorithm \( A \) is self-stabilizing for the specification \( R \) if (i) any computation of \( A \) starting from a configuration satisfying \( \mathcal{L} \) satisfies \( R \) (correctness) and (ii) starting from any configuration \( \in \mathcal{C} \), any computation of \( A \) reaches a configuration which satisfies \( \mathcal{L} \) (convergence) in finite steps.

3.3.2 Problem Specification

Our algorithm for self-stabilizing wormhole routing in hypercubes ensures that every message is successfully delivered from source to destination as a sequence of flits.

**Specification 3.1 (Wormhole Routing in Hypercubes)** Given a well-constructed message \( M \) from a source node \( S \) in a hypercube network, an execution of the system satisfies the Wormhole Routing in Hypercubes problem (we will call it WRH) if the following property holds:

**Reliable Delivery:** The message \( M \) from \( S \) will be safely delivered at its destination.

This property ensures correct behavior of the algorithm. We also require our algorithm to be self-stabilizing as per the definition given in the previous section.

3.4 Problems Encountered

In the self-stabilizing environment, network faults can corrupt the local variables of any network processor. Thus message flits and their wormhole routing paths can be spontaneously introduced, lost, or corrupted.

3.4.1 Faulty Messages

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 header and/or tail flits 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.

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.

3.4.2 Faulty Paths

The hypercube topology offers many paths for a single flit to follow to its destination. Thus all hypercube 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.

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.

3.5 Solutions and Ideas

Our algorithm implements the following solutions to these problems:

3.5.1 Faulty Messages

**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 timestamp the channel locked as soon a header flit is received. When a new header is received the channel is freed and the timer variable takes the current timestamp.

Headerless Flooding. Since the network can start in any arbitrary state, it is possible to have every processor filled by a non-header flit. These flits will not have channels locked for them so they will be discarded.

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.

3.5.2 Faulty Paths

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 from the timestamp on the message and the message will eventually be dropped.

3.6 Assumptions and Conventions

Data Corruption: Only variable corruptions at the processors are handled. Data corruption at processors and during transit (in links) are not handled. It is the responsibility of the application layer to discard these messages. Instead completely lost data are handled.

Unicast: Message from a source is sent only to one destination. Multicast and broadcast are not handled.

Multiple senders: At a given instant, any number of processors can send messages.

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.

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 [38]. 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 [3]. 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.

**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.

**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 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.

3.7 Model and Hypothesis

A wormhole routing processor in a $n$-cube network has $n$ incoming channels and $n$ outgoing channels, requiring more hardware and software sophistication. Complex decision making must be done in the routing code as to how and where to send a packet.

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 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.

3.7.1 E-cube Routing for Hypercubes

In an $n$-cube, the maximum distance between any two processors is $n$. The relative address of source and destination is arrived using XOR operation on their binary addresses. If the distance between source and destination is $d$ then their relative address will have bit 1 at $d$ bit positions corresponding to $d$ distinct dimensions. These dimensions are called preferred dimensions while the remaining $n - d$ dimensions are called spare dimensions. An optimal (minimal) path of length $d$ contains $d$ preferred dimensions only. Any such path must use links at each of these preferred dimensions later or sooner.

When a node receives a header flit it checks if it is destination. If it is then the flit is...
delivered and the channel is reserved for data and tail flits. If this is not the case then the lowest preferred dimension is reserved if it is available, if not the next preferred dimension and so forth. This is done for all the preferred dimensions by repeated checking since all these dimensions must be used to get the optimal path.

3.8 Data Structures

**Network:** A $n$-cube network has $2^n$ nodes and $n2^{n-1}$ links. Nodes are aware of the dimension ($n$) of the hypercube. Each node has an unique identifier represented by a sequence of $n$-bit binary digits $(b_{n-1}, b_{n-2}, \ldots, b_0)$, where $b_i \in \{0, 1\}$ for $0 \leq i \leq n-1$. The bit $b_i$ is called the $i$-dimensional bit. Two nodes are connected by a link if and only if the binary representation of their nodes differ in exactly one bit. A link is called an $i$-dimensional link if it connects two nodes that differ in their $i$-dimensional bits. Each node represents a processor and its memory. Each link represents a communication channel between a pair of processors.

**Channels:** Each processor $P$ has $n$ incoming channels and $n$ outgoing channels. The incoming and outgoing channels are named as $P^{i0}, P^{i1}, \ldots, P^{i(n-1)}$ and $P^{o0}, P^{o1}, \ldots, P^{o(n-1)}$, respectively. The Hamming distance between two processors $X$ and $Y$, denoted by $H(X,Y)$, is the number of bits in which their binary representation differs.

**Virtual Channel:** 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 $P^{iv}$.
**Messages:** Messages are exchanged by the processors in the form of *flits*. There are three types of flits: header, data, and tail flits. Header and tail flits are control flits that contribute to the establishment and destruction of the circuits, respectively. Header flits carry the destination address where the message has to be delivered. Tail flit marks the end of the message. Data flits contain the actual content of the message. All flits contain local message identifiers to prevent mismatch of flits at the destination.

**Message Identifier:** The protocol must be able to distinguish messages from one another. However, since there are many simultaneous senders in the network, the message identifiers may not be globally unique. We implement this uniqueness by using a two-tuple made of source processor (*S*) and processor message number (*mid*). These pairs guarantee uniqueness since the processor identifiers are assumed to be unique.

### 3.9 Algorithm

This section formally presents the self-stabilizing wormhole routing algorithm in hypercubes. Finally, in Section 3.9.2 each action of the algorithm is briefly described.

#### 3.9.1 Main Program

The main program (presented as Algorithm 3.1) consists of three sets of actions:

1. **RECV** actions (presented as Algorithm 3.2) are activated when a flit is received on an incoming channel,

2. **SEND** actions (presented as Algorithm 3.3) are activated when a processor is able to transmit a flit buffer,

3. **ERROR** actions (presented as Algorithm 3.4) are activated when a local error condition is detected.
3.9.2 Algorithm Description

In this section we briefly explain the behavior of our algorithm in finding a path for the message flits from source to destination. The main algorithm is presented as three sets of actions: receive actions, send actions, and error correction actions. All of these actions are evaluated in parallel. If more than one actions are enabled at an instant, the daemon selects exactly one action and it is executed. Receive actions and send actions are enabled when flits move from one processor to another. Receive and send actions are enabled at the receiver and sender, respectively. Error correction actions are special class of actions that are executed when transient faults occur in the system. These actions contribute to the self-stabilizing task of the algorithm.

First, we explain the process through which the message flits are routed from a source to a destination with the corresponding receive/send actions. Second, we explain the actions involved in recovering the system from faults.

3.9.2.1 Normal Behavior

Choosing Preferred Dimensions: Each $n$-cube ($n$-dimensional hypercube) has $n$ dimensions. The message flits from source to destination are transmitted only using specific dimensions, called preferred dimensions. These dimensions are calculated in Line 2.08 in Algorithm 3.2. One of these dimensions is then chosen in Lines 2.09-2.11, and the header flit is routed in that dimension. All of the routing calculations are done only for the header flit since data and tail flits follow the path taken by the header flit.

Circuit Establishment by Header Flits: The header flit establishes a circuit for data and tail flits to follow. When a processor receives a header flit, Action $R_1$ becomes
Algorithm 3.1 Self-stabilizing Wormhole Routing in Hypercubes (Main program) (Algorithm \textsc{WRH}) for Processor $x$.

Constants:
\begin{enumerate}
\item $n$ :: Total number of processors in the hypercube;
\item $\text{max} - \text{timer}$ :: Maximum time before which a channel gets unlocked;
\item $C$ :: Current processor label;
\item $S$ :: Source processor label;
\item $D$ :: Destination processor label;
\end{enumerate}

Variables:
\begin{enumerate}
\item $R$ :: Relative address of Source and Destination;
\item $k$ :: $0..n-1$;
\item $\text{sys - timer}$ :: Current system time;
\item $\text{dat}$ :: Data;
\item $\text{mid}$ :: Message ID;
\end{enumerate}

Macros:
\begin{enumerate}
\item $\text{Locked}(P^i_j) \equiv$ Returns the number of outgoing channels locked for the input channel $P^i_j$;
\end{enumerate}

Predicates:
\begin{enumerate}
\item $\text{ValidFlit}(P^i_j) \equiv P^i_j.\text{Timer} \leq \text{max} - \text{timer} \land P^i_j.\text{Source} = S \land P^i_j.\text{MID} = \text{mid}$;
\item $\text{CanSend} \equiv P^o_k.\text{CTS} = \text{LOW} \land P^o_k.\text{Lock} = P^i_k$;
\end{enumerate}

Flits:
\begin{enumerate}
\item $\text{hf}(S,\text{mid},D)$ :: Header flit;
\item $\text{df}(S,\text{mid},\text{dat})$ :: Data flit;
\item $\text{tf}(S,\text{mid},\text{dat})$ :: Tail flit;
\end{enumerate}

Channels:
\begin{enumerate}
\item $P^i_k$ :: Incoming channel;
\item $P^o_k$ :: Outgoing channel;
\item $P^o_k.\text{Lock}$ :: \{NULL, $P^i_k$\};
\item $P^i_k.\text{Buffer}$ :: \{<empty>, hf, df, tf\};
\item $P^o_k.\text{Buffer}$ :: \{<empty>, hf, df, tf\};
\item $P^i_k.\text{CTS}$ :: \{HIGH, LOW\};
\item $P^o_k.\text{CTS}$ :: \{HIGH, LOW\};
\item $P^i_k.\text{Timer}$ :: Timestamp
\end{enumerate}

Actions:
\begin{enumerate}
\item \textbf{begin}
\item Receive Actions
\item \[\] Send Actions
\item \[\] Error Correction Actions
\item \textbf{end}
\end{enumerate}
Algorithm 3.2 Algorithm WRH Receive Actions.

2.01 (R1) rcv hf(S, mid, D) from P^i \rightarrow \\
2.02 \quad \text{do while exists}(P^o\text{k}.\text{Lock} = P^i) \\
2.03 \quad P^o\text{k}.\text{Lock} = \text{NULL} \\
2.04 \quad \text{od} \\
2.05 \quad \text{if } D \oplus C = 0 \rightarrow \\
2.06 \quad \quad \text{deliver } hf(D) \\
2.07 \quad \quad \text{if } D \oplus C \neq 0 \rightarrow \\
2.08 \quad \quad \quad R, r := C \oplus D, 0; \\
2.09 \quad \quad \quad \text{do while } \neg(R[r] = 1 \land P^o\text{r}.\text{Lock} = \text{NULL}) \\
2.10 \quad \quad \quad \quad r = r + 1 \\
2.11 \quad \quad \quad \text{od} \\
2.12 \quad \quad \quad P^o\text{r}.\text{Lock}, P^o\text{r}.\text{Buffer}, P^o\text{r}.\text{CTS} := P^i, hf(S, mid, D), \text{HIGH}; \\
2.13 \quad \quad \quad P^o\text{r}.\text{Timer}, P^o\text{r}.\text{Source}, P^o\text{r}.\text{MID} := \text{sys\_timer}, S, mid; \\
2.14 \quad \text{fi} \\
2.15 (R2) rcv df(S, mid, dat) from P^i \rightarrow \\
2.16 \quad \text{if } \text{Locked}(P^i) = 0 \land \text{ValidFlit}(P^i) \rightarrow \\
2.17 \quad \quad \text{deliver } df(S, mid, dat); \\
2.18 \quad \quad \text{if } \text{Locked}(P^i) = 1 \land \text{ValidFlit}(P^i) \rightarrow \\
2.19 \quad \quad \quad P^i.\text{Buffer}, P^i.\text{CTS} := \text{df}(S, mid, dat), \text{HIGH}; \\
2.20 \quad \quad \quad \text{do while exists}(P^o\text{k}.\text{Lock} = P^i) \\
2.21 \quad \quad \quad \quad P^o\text{k}.\text{Lock} = \text{NULL} \\
2.22 \quad \quad \quad \text{od} \\
2.23 \quad \quad \text{discard } df \\
2.24 \text{fi} \\
2.25 (R3) rcv tf(S, mid, dat) from P^i \rightarrow \\
2.26 \quad \text{if } \text{Locked}(P^i) = 0 \land \text{ValidFlit}(P^i) \rightarrow \\
2.27 \quad \quad \text{deliver } tf(S, mid, dat) \\
2.28 \quad \quad \text{if } \text{Locked}(P^i) = 1 \land \text{ValidFlit}(P^i) \rightarrow \\
2.29 \quad \quad \quad P^i.\text{Buffer}, P^i.\text{CTS} := \text{tf}(S, mid, dat), \text{HIGH}; \\
2.30 \quad \quad \quad \text{do while exists}(P^o\text{k}.\text{Lock} = P^i) \\
2.31 \quad \quad \quad \quad P^o\text{k}.\text{Lock} = \text{NULL} \\
2.32 \quad \quad \quad \text{od} \\
2.33 \quad \quad \text{discard } tf \\
2.34 \text{fi} \\

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 3.3 Algorithm \textit{WRH} Send Actions.

3.01 (S1) \textit{CanSend} \rightarrow
3.02 \textit{if } \textit{P}^{ik}.\textit{Buffer} = hf(S, mid, D) \rightarrow
3.03 \textit{send } hf(S, mid, D) \textit{to } P^{ok};
3.04 \textit{[P}^{ik}.\textit{Buffer} = df(S, mid, dat) \rightarrow
3.05 \textit{send } df(S, mid, dat) \textit{to } P^{ok};
3.06 \textit{[P}^{ik}.\textit{Buffer} = tf(S, mid, dat) \rightarrow
3.07 \textit{send } tf(S, mid, dat) \textit{to } P^{ok};
3.08 \textit{P}^{ok}.\textit{Lock} = NULL;
3.09 \textit{fi}
3.10 \textit{P}^{ik}.\textit{CTS}, \textit{P}^{ik}.\textit{Buffer} := \textit{LOW}, < \textit{empty} >;

Algorithm 3.4 Algorithm \textit{WRH} Error Correction Actions.

4.01 (E1) \textit{P}^{ik}.\textit{CTS} = \textit{LOW} \land \textit{P}^{ik}.\textit{Buffer} \neq < \textit{empty} > \rightarrow
4.02 \textit{P}^{ik}.\textit{CTS} := \textit{HIGH};
4.03 (E2) \textit{[P}^{ik}.\textit{CTS} = \textit{HIGH} \land \textit{P}^{ik}.\textit{Buffer} = < \textit{empty} > \rightarrow
4.04 \textit{P}^{ik}.\textit{CTS} := \textit{LOW};
4.05 (E3) \textit{[P}^{ik}.\textit{Buffer} = df(S, mid, dat) \lor \textit{P}^{ik}.\textit{Buffer} = tf(S, mid, dat))
\land \textit{Locked}(P^{ik}) \neq 1 \rightarrow
4.06 \textit{P}^{ik}.\textit{CTS}, \textit{P}^{ik}.\textit{Buffer} := \textit{LOW}, < \textit{empty} >;
4.07 \textit{do while exists}(\textit{P}^{ok}.\textit{Lock} = \textit{P}^{ik})
4.08 \textit{P}^{ok}.\textit{Lock} := \textit{NULL};
4.09 \textit{od}
4.10 (E4) \textit{[TIMEOUT P}^{ok}.\textit{Lock} = \textit{P}^{ik} \land \textit{P}^{ik}.\textit{CTS} = \textit{LOW} \rightarrow
4.11 \textit{P}^{ok}.\textit{Lock}, \textit{P}^{ik}.\textit{CTS}, \textit{P}^{ik}.\textit{Buffer} := \textit{NULL}, \textit{LOW}, < \textit{empty} >;
enabled. As mentioned earlier, all routing decisions are made for the header flit by this action. The first task performed here is to clear out all channel locks that already exist for the incoming channel of the header flit. Next, the flit's destination field is matched with the current processor, and if it matches, the flit is delivered. Otherwise the flit is transmitted through one of the preferred dimensions. The flit is stored in the node buffer (Buffer), and all the variables, viz., Lock, CTS, Timer, Source, and MID are assigned appropriate values. When the Buffer is filled with the header flit and CanSend predicate returns true, Action S1 is enabled. This predicate evaluates if the necessary variables have appropriate values in order to send the flit. The header flit is sent by executing Line 3.03.

Transmission of Data Flits: The data flits follow the path established by the header flit. When a data flit is received at a processor from one of its input channels, say $P_{ik}$, Action R2 becomes enabled. The number of output channels locked for $P_{ik}$ is then computed using the macro Locked($P_{ik}$). If there are no output channels locked for $P_{ik}$ and the flit is a valid flit (ValidFlit($P_{ik}$) macro is used), then the flit is delivered at the node. If exactly one channel is locked, the flit is stored in its buffer. If more than one channels are locked or the flit is not valid, then it is discarded and channels are cleared. Now, the flit is checked for its validity by Action S1, and is sent to the locked output channel (Line 3.05).

Circuit Destruction by Tail Flits: The tail flits use the path established by the header flit, and releases the resources reserved on its way to the destination. Action R3 is enabled when a processor receives a tail flit from one of its input channels. Similar to the data flits, the number of output channels locked for the flit is evaluated. If no
output channels are locked, the flit is delivered and the circuit is destroyed. If only one channel is reserved, the flit is transmitted through that channel. Action $S_1$ is enabled and Line 3.07 is executed to send a tail flit. The circuit is destroyed after the transmission of the tail flit.

3.9.2.2 Error Correction

When the Buffer of any channel is filled with a flit, the $CTS$ variable must be set to $HIGH$ and only one output channel should be locked. Output channels locked must be one. The former condition is ensured by the error correction actions $E_1$ and $E_2$. The latter condition is ensured by $E_3$. When a flit stays in a processor for a prolonged period of time, it is considered to be unroutable and is discarded using Action $E_4$.

3.10 Proof of Correctness

In this section, we will prove the correctness of the algorithm presented in Section 3.9 and show that it satisfies the specifications as defined in Section 3.3.2. First, the system legitimacy predicates are described in Section 3.10.1. Second, we proceed to prove reliable delivery in Section 3.10.2 and convergence in Section 3.10.3.

3.10.1 Legitimacy Predicates

In this section, we will define three sets of predicates: processor ($LP$), message ($LM$), and circuit ($LC$) predicates. Thus, the legitimacy predicate is defined as follows:

$$L_{WRH} \equiv LP \land LM \land LC.$$  

The detailed definitions of the predicates $LP$, $LM$, and $LC$ will be given in Sections 3.10.1.1, 3.10.1.2, and 3.10.1.3 respectively.
3.10.1.1 Processor Predicates

\[ \mathcal{LP} \equiv \mathcal{LP}_1 \land \mathcal{LP}_2. \]

\[ \mathcal{LP}_1: \text{For every processor, all incoming channels with a } CTS \text{ value equal to } HIGH \text{ must have a flit in their buffer.} \]

\[ \mathcal{LP}_1 \equiv \forall P \in V : \ P^{ik}.CTS = \text{HIGH} \iff P^{ik}.\text{Buffer} \neq <\text{empty}>. \]

\[ \mathcal{LP}_2: \text{For every processor, all incoming channels must have a unique output channel locked for it, or no locked output channel at all.} \]

\[ \mathcal{LP}_2 \equiv \forall P \in V : \ \text{Locked}(P^{ik}) = 1 \lor \text{Locked}(P^{ik}) = 0. \]

3.10.1.2 Message Predicates

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

\[ \mathcal{LM} \equiv (M = (\text{flits delivered}) \cup (\text{flits in transit}) \cup (\text{flits not yet transmitted})). \]

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: \( H, D_1, D_2, T \)
3.10.1.3 Circuit Predicates

Message passing in hypercubes involves heavy usage of circuits. Each processor in an \( n \)-cube can participate in \( n \) different circuits. All processors have \( n \) incoming and \( n \) outgoing channels. The following are the legitimacy predicates for network circuits.

\( \mathcal{LC} \): Formally, a structurally legitimate circuit \( C \) is a finite set of incoming and outgoing channels well-ordered by the relation \( R \), such that \( N R M \) iff there is a path from \( N \) to \( M \), where \( N \) and \( M \) are processor channels.

Formally, \( R \) is defined as follows:

(i) We define \( P^{ix} R P^{ox} \) on a Processor \( P \) iff \( P^{ox}.Lock = P^{ix} \).

(ii) We define \( P^{ox} R Q^{ix} \) iff \( P^{ox} \) and \( Q^{ix} \) are the same physical unidirectional link \( < P, Q > \) in \( E \).

(iii) The relation \( R \) is reflexive, \( P^{ch} R P^{ch} \).

(iv) The relation \( R \) is transitive, \( P^{ix} R P^{ox} \wedge P^{ox} R Q^{ix} \Rightarrow P^{ix} R Q^{ix} \).

(v) The relation \( R \) is antisymmetric, \( P^{ch} R Q^{ch} \wedge Q^{ch} R P^{ch} \Rightarrow P^{ch} = Q^{ch} \).

An example of a legitimate circuit \( C_1 = \{ P^{iv}_1, P^{o2}_1, P^{i1}_2, P^{o7}_2, P^{i9}_3 \} \).

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

\[
\{ < P^{iv}_1, P^{o2}_1 >, /* a channel lock */
< P^{o2}_1, P^{i1}_2 >, /* a physical link */
< P^{iv}_2, P^{i1}_2 >, /* transitive */
< P^{i1}_2, P^{o7}_2 >, /* a channel lock */
< P^{o2}_2, P^{o7}_2 >, /* transitive */
< P^{iv}_2, P^{o7}_2 >, /* transitive */
\]
3.10.2 Reliable Delivery

We will prove that when the system is in a legitimate configuration the algorithm behaves correctly. We have assumed that no new faults occur, hence the error actions will not be invoked. Only a transient fault can cause the network to enter an illegitimate state.

In order to prove the reliable delivery property, we need to establish the following liveness related properties. Liveness is the combination of these three properties. Our first step will be to prove that these three situations do not occur in our algorithm. 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.

1. **Deadlock:** A network is in a deadlock state when one or more processors are waiting for a resource that will never be released. A circular chain of processors exists such that each processor holds a resource which some other processor is waiting for.

2. **Livelock:** Livelock happens when all processors are executing as normal, but the
algorithm fails to progress. This occurs when a flit is repeatedly denied for delivery at the destination.

3. **Starvation**: Starvation occurs when a processor is prevented from performing a critical function forever. This can occur when a particular resource a processor waiting for is not granted forever.

Liveness is the combination of the above three properties. The first step is to prove that these three situations do not occur in our algorithm.

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

**Proof.** Deadlock can occur only when there is a cycle in the network channel dependency graph. In a legitimate state our Algorithm (WRH) assigns channels in strict descending order of dimension. Hence, no cycles can be formed in the channel dependency graph. □

**Lemma 3.2 (Livelock Freedom)** Starting from a configuration that satisfies the legitimacy predicates, the network will not livelock.

**Proof.** 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. In Algorithm WRH, the path taken by these flits comply to the predicate $LC_1$. So, a flit is not delivered only if it gets lost in the channels. However, that cannot happen in a legitimate state. □
Lemma 3.3 (Starvation Freedom) Starting from a configuration that satisfies the legitimacy predicates, the network will not starve.

Proof. Starvation can be caused due to any of the following reasons:

1. A processor waits forever to send a flit. This cannot happen because the input selection policy is implemented in our solution using a fair (the round-robin scheduling) scheme.

2. 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. The processor cannot receive any flit in that channel and the sender will starve. In a legitimate state this cannot happen as per predicate $LP_1$.

3. There are more than one outgoing channels locked for a particular incoming channel in a node. Due to this one-to-many mapping, a node may not be able to allocate an output channel for a new flit causing starvation of the node. But, the predicate $LP_2$ implies a one-to-one mapping.

4. A header flit is waiting for an output channel infinitely because a previously established circuit is not destroyed. Since the message predicate $LM$ and circuit predicate $LC$ hold in a legitimate state, after a tail flit is sent, the circuit will be destroyed (refer Line 3.08 in Algorithm 3.3). Hence the channel is freed for future use.

Now, we prove the reliable delivery assuming that the liveness properties hold. We will establish the result in four steps.

Lemma 3.4 (Receive Flits) Starting from a configuration that satisfies the legitimacy predicate, every flit sent to a processor is eventually received.
Proof. In a legitimate state, a Processor $P$ with a true RECV action on a particular incoming channel $P^{ch}$ will be activated when the input channel buffer is filled with a flit. Starvation will not occur since 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 $P^{ch}$ 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 (refer Lemma 3.3), then the RECV action on the incoming channel $P^{ch}$ will be activated. □

Lemma 3.5 (Store/Deliver Flits) 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 $R_1$, $R_2$, and $R_3$, there are three possible outcomes for a received flit: delivery, write to a flit buffer, or discard. Every discard statement (Lines 2.24 and 2.35) in these three actions is protected by a guard that checks for faulty allocation of channels, 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 (Send Flits) 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. The network is in a legitimate state and there is a $P^{iz}$.Buffer variable,
which contains a flit that has to be transmitted.

A SEND action requires three conditions to fulfill: the channel lock variable of an outgoing channel $P_{ox} = P_{ix}$, 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 liveness properties it will eventually be granted the outgoing channel resource $P_{ox}$. 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 $P_{ox}$ 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 (Establishing Circuit)** Starting from a configuration that satisfies the legitimacy predicate, a circuit is created using which message flits are delivered at the destination.

**Proof.** For a flit to be routed it has to be allocated an outgoing channel. This channel is allocated only a preferred dimension that does not have the Lock variable set, using the scheme *dimension ordered routing*. Since we have assumed that the network is in a legitimate state this allocation is continued till the destination where the message flits are delivered. □

**Theorem 3.1 (Correctness)** Starting from a configuration which satisfies the legitimacy predicate, every well-constructed message sent by a node will be delivered safely at its destination.

**Proof.** The proof follows from Lemmas 3.4, 3.5, 3.6, and 3.7. □
3.10.3 Convergence

Lastly, we prove that this algorithm will converge to a legitimate state from any arbitrary initialization in finite time. Here, we prove that the conjunction of all predicates will eventually hold in the system, and thus the system converges to a legitimate state.

3.10.3.1 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 3.8** ($\mathcal{LP}_1$) Starting from an arbitrary configuration, $\mathcal{LP}_1$ eventually holds.

**Proof.** The proof directly follows from the Actions $\mathcal{E}_1$ and $\mathcal{E}_2$. □

**Lemma 3.9** ($\mathcal{LP}_2$) Starting from an arbitrary configuration, $\mathcal{LP}_2$ eventually holds.

**Proof.** We need to show that no input channel of a processor can be connected to more than one output channels. Assume that in the current configuration, a processor $P$ has an input channel $P^{ix}$ such that two channels $P^{or1}$ and $P^{or2}$ have their Lock variables set to $P^{ix}$. The correction of this fault would depend on the type of flit $P$ receives.

- $P$ receives a header flit. The recv action $\mathcal{R}_1$ is activated. When $\mathcal{R}_1$ is executed, all outgoing channels are unlocked. Hence, $\mathcal{LP}_2$ is satisfied.

- $P$ receives a data or tail flit. One of the recv actions $\mathcal{R}_2$ or $\mathcal{R}_3$ is enabled at $P$. Execution of $\mathcal{R}_2$ or $\mathcal{R}_3$ frees up all the outgoing channels. Hence, $\mathcal{LP}_2$ is satisfied. □
3.10.3.2 Message Predicates

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

**Lemma 3.10** \((\mathcal{LM})\) *Starting from an arbitrary configuration, \(\mathcal{LM}\) 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 3.11

- Headerless fragments cannot traverse the network forever due to TIMEOUT action. Assuming that there is no misrouting, a data or a tail flit will be delivered to the processor of the last incoming channel in the circuit.

\[\Box\]

3.10.3.3 Circuit Predicates

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

**Lemma 3.11** \((\mathcal{LC})\) *Starting from an arbitrary configuration, \(\mathcal{LC}\) 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 \(C = \{P_1^\text{IF}, P_1^\text{OF}\},\)
$P_2^{ix}, P_3^{ix}, ..., P_N^{ix}$. 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 $P_1^{ix}$). The action $E_4$ will be eventually activated on each outgoing channel in $C$, and the circuit will be destroyed.

\begin{proof}

\end{proof}

**Theorem 3.2 (Convergence)** Starting from any configuration, any computation of Algorithm \textsc{WZH} reaches a configuration satisfying $L_{\textsc{WZH}}$.

\begin{proof}
The proof follows from Lemmas 3.8, 3.9, 3.10, and 3.11.
\end{proof}

**Theorem 3.3 (Self-stabilizing)** Algorithm \textsc{WZH} is self-stabilizing.

\begin{proof}
Follows directly from Theorems 3.1 and 3.2.
\end{proof}
CHAPTER 4

SELF-STABILIZING AND FAULT-TOLERANT WORMHOLE ROUTING IN HYPERCUBES

In this chapter we present a self-stabilizing algorithm similar to that of previous chapter with some additional features. First, we list the new assumptions made for the algorithm. Informal description of the algorithm is given in Section 4.2. Finally, we end this chapter by giving the proof of correctness in Section 4.3.

4.1 New Assumptions

- A fault can be a processor fault or a link fault.
- The source and the destination are fault free.
- If a processor fails then all its links are considered to be failed.
- All links are bidirectional. If a link is faulty, both directions are faulty.
- The total number of faulty components (faulty links and faulty processors together) is less than the dimension of the hypercube.
- Each node maintains the local link failure information in a n-bit status-vector \( X = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle \)
Algorithm 4.1 Self-stabilizing Wormhole Routing in Hypercubes (Data Structures) (Algorithm WWRH_{FT}) for Processor $x$ with node and link failures.

Constants:
5.01 $n$ :: Total number of processors in the hypercube;
5.02 $\text{max - timer}$ :: Maximum time before which a channel gets unlocked;
5.03 $C$ :: Current processor label;
5.04 $S$ :: Source processor label;
5.05 $D$ :: Destination processor label;

Variables:
5.06 $R$ :: Relative address of Source and Destination;
5.07 $X$ :: $n$-bit binary label that has information about faulty neighbors;
5.08 $O$ :: $n$-bit binary label that stores the dimensions traversed;
5.09 $k$ :: $0..n-1$;
5.10 $\text{sys - timer}$ :: Current system time;
5.11 $\text{dat}$ :: Data;
5.12 $\text{mid}$ :: Message ID;

Macros:
5.13 $\text{Locked}(P^i) =$ Returns the number of outgoing channels locked for the input channel $P^i$;
5.14 $\text{FirstOne}(B)$ = Returns the lowest bit position that has value 1 in the binary number $B$ if $B \neq 0$ or returns $-1$ if $B = 0$;
5.15 $\text{FreeChannels}(P^k) =$ Unlocks all output channels that are locked to the input channel $P^k$;
5.16 $\text{IsCycle}(O,l)$ = Returns true if the dimension $l$ forms a cycle in the binary string $O$;

Predicates:
5.17 $\text{ValidFlit}(P^i) =$ $P^i\.\text{Timer} \leq \text{max - timer} \land P^i\.\text{Source} = S \land P^i\.\text{M ID} = \text{mid}$;
5.18 $\text{CanSend} =$ $P^k\.\text{CTS} = \text{LOW} \land P^k\.\text{Lock} = P^k$;

Flits:
5.19 $\text{hf}(S,\text{mid},D,T,O) ::$ Header flit;
5.20 $\text{df}(S,\text{mid},\text{dat}) ::$ Data flit;
5.21 $\text{tf}(S,\text{mid},\text{dat}) ::$ Tail flit;
5.22 $\text{mf}(S,\text{mid}) ::$ Marked flit;

Channels:
5.23 $P^k$ :: Incoming channel;
5.24 $P^k$ :: Outgoing channel;
5.25 $P^k\.\text{Lock} :: \{\text{NULL}, P^k\}$
5.26 $P^k\.\text{Buffer} :: \{<\text{empty}>, \text{hf}, \text{df}, \text{tf}\}$
5.27 $P^w\.\text{Buffer} :: \{<\text{empty}>, \text{hf}, \text{df}, \text{tf}\}$
5.28 $P^k\.\text{Status} :: \{\text{marked}, \text{NULL}\}$
5.29 $P^k\.\text{CTS} :: \{\text{HIGH}, \text{LOW}\}$
5.30 $P^w\.\text{CTS} :: \{\text{HIGH}, \text{LOW}\}$
5.31 $P^k\.\text{Timer} ::$ Timestamp

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Algorithm 4.2 Algorithm WRMFT Receive Actions.

6.01 \( (R_1) \) recv \( h_f(S, \text{mid}, D, T, O) \) from \( P^{ik} \) →
6.02 \[ \text{FreeChannels}(P^{ik}); \]
6.03 \[ \text{do while exists}(P^{ik} . \text{Source} = S \land P^{ik} . \text{MID} = \text{mid}) \]
6.04 \[ P^{ik} . \text{Status} = \text{marked}; \]
6.05 \[ \text{od} \]
6.06 \[ R = C \oplus D; \]
6.07 \[ \text{if } R = 0 \rightarrow \text{deliver } h_f(S, \text{mid}, D, T); \]
6.08 \[ \text{end} \]
6.09 \[ \text{MASK} := (2^n - 1) - 2^l; l := \text{FirstOne}(R \land X \land \text{MASK}); \]
6.10 \[ \text{if } ((l = -1) \land (R[c] = 1)) \rightarrow l := c; \]
6.11 \[ \text{if } ((l = -1) \land (R[c] = 0)) \rightarrow \]
6.12 \[ T := T \vert \text{MASK}; l := \text{FirstOne}(\neg T \land X); \]
6.13 \[ \text{if } (l = -1) \]
6.14 \[ T := T \vert R; l := \text{FirstOne}(\neg T \land X); \]
6.15 \[ \text{fi} \]
6.16 \[ \text{fi} \]
6.17 \[ \text{fi} \]
6.18 \[ \text{fi} \]
6.19 \[ P^{il} . \text{Lock}, P^{il} . \text{Buffer}, P^{il} . \text{CTS} := P^{il} . hf(S, \text{mid}, D, T), \text{HIGH}; \]
6.20 \[ P^{il} . \text{Timer}, P^{il} . \text{Source}, P^{il} . \text{MID} := \text{sys} - \text{timer}, S, \text{mid}; \]
6.21 \[ \text{fi} \]
6.22 \[ \text{fi} \]
6.23 \[ \text{fi} \]
6.24 \[ (R_2) \] recv \( df(S, \text{mid}, dat) \) from \( P^{ik} \) →
6.25 \[ \text{if } \text{Locked}(P^{ik}) = 0 \land \text{ValidFlit}(P^{ik}) \rightarrow \text{deliver } df(S, \text{mid}, dat); \]
6.26 \[ \text{if } \text{Locked}(P^{ik}) = 1 \land \text{ValidFlit}(P^{ik}) \rightarrow \]
6.27 \[ P^{ik} . \text{Buffer}, P^{ik} . \text{CTS} := df(S, \text{mid}, dat), \text{HIGH}; \]
6.28 \[ \text{FreeChannels}(P^{ik}); \text{discard } df; \]
6.29 \[ \text{fi} \]
6.30 \[ \text{fi} \]
6.31 \[ \text{fi} \]
6.32 \[ (R_3) \] recv \( tf(S, \text{mid}, dat) \) from \( P^{ik} \) →
6.33 \[ \text{if } \text{Locked}(P^{ik}) = 0 \land \text{ValidFlit}(P^{ik}) \rightarrow \text{deliver } tf(S, \text{mid}, dat); \]
6.34 \[ \text{if } \text{Locked}(P^{ik}) = 1 \land \text{ValidFlit}(P^{ik}) \rightarrow \]
6.35 \[ P^{ik} . \text{Buffer}, P^{ik} . \text{CTS} := tf(S, \text{mid}, dat), \text{HIGH}; \]
6.36 \[ \text{FreeChannels}(P^{ik}); \text{discard } tf; \]
6.37 \[ \text{fi} \]
6.38 \[ \text{fi} \]
6.39 \[ \text{fi} \]
6.40 \[ (R_4) \] recv \( mf(S, \text{mid}) \) from \( P^{ik} \) →
6.41 \[ \text{if } \text{Locked}(P^{ik}) = 1 \land \text{ValidFlit}(P^{ik}) \rightarrow \]
6.42 \[ P^{ik} . \text{Buffer}, P^{ik} . \text{CTS} := mf, \text{HIGH}; \]
6.43 \[ \text{if } \text{Locked}(P^{ik}) \neq 1 \land \neg \text{ValidFlit}(P^{ik}) \rightarrow \]
6.44 \[ \text{FreeChannels}(P^{ik}); \text{discard } mf; \]
6.45 \[ \text{fi} \]
Algorithm 4.3 Algorithm $WRH_{FT}$ Send Actions.

7.01 $(S_1)$ CanSend $\rightarrow$
7.02 if $P^k.Buffer = h(S, mid, D) \rightarrow$
7.03 $\quad$ send $h(S, mid, D)$ to $P^k$;
7.04 $\quad$ if $P^k.Buffer = df(S, mid, dat) \land P^k.Status = NULL \rightarrow$
7.05 $\quad$ send $df(S, mid, dat)$ to $P^k$;
7.06 $\quad$ if $P^k.Buffer = tf(S, mid, dat) \land P^k.Status = NULL \rightarrow$
7.07 $\quad$ send $tf(S, mid, dat)$ to $P^k$;
7.08 $\quad$ $P^k.Lock = NULL$;
7.09 $\quad$ if $P^k.Buffer = df(S, mid, dat) \land P^k.Status = marked \rightarrow$
7.10 $\quad$ send $mf(S, mid)$ to $P^k$;
7.11 $\quad$ $P^k.Buffer = tf(S, mid) \rightarrow$
7.12 $\quad$ send $mf(S, mid)$ to $P^k$;
7.13 fi
7.14 $P^k.CTS, P^k.Buffer := LOW, < empty >$;

Algorithm 4.4 Algorithm $WRH_{FT}$ Error Correction Actions.

8.01 $(E_1)$ $P^k.CTS = LOW \land P^k.Buffer \neq < empty >$ $\rightarrow$
8.02 $P^k.CTS := HIGH$;
8.03 $(E_2)$ $P^k.CTS = HIGH \land P^k.Buffer = < empty >$ $\rightarrow$
8.04 $P^k.CTS := LOW$;
8.05 $(E_3)$ $(P^k.Buffer = df(S, mid, dat) \lor P^k.Buffer = tf(S, mid, dat))$
8.06 $\land Locked(P^k) \neq 1$ $\rightarrow$
8.07 $P^k.CTS, P^k.Buffer := LOW, < empty >$;
8.08 do while exists($P^k.Lock = P^k$)
8.09 $P^k.Lock := NULL$;
8.10 od
8.11 $(E_4)$ $TIMEOUT P^k.Lock = P^k \land P^k.CTS = LOW$ $\rightarrow$
8.11 $P^k.Lock, P^k.CTS, P^k.Buffer := NULL, LOW, < empty >$;
4.2 Algorithm Description

There are two major modifications made to the algorithm presented in the previous chapter. First, we need an extra computation to reroute flits if faulty nodes or links are encountered. Second, we introduce a new type of flits, called marked flits to help establish a new path for the header flit, if necessary. The marked flits are control flits, hence not delivered at the destination. However, they increase the efficiency of the algorithm.

The additional calculations for establishing a fault-free route is performed in Action $\mathcal{R}_1$. The following order is used by the header flit to find a dimension to route the flits in case of node/link failures. First, the lowest nonfaulty preferred dimension that is not the incoming dimension, is checked for availability. If such a dimension is found, the received flit is sent through that dimension. Second, if all nonincoming preferred dimensions are faulty and the incoming dimension is also a preferred dimension, then the incoming dimension is selected and the flit is returned to the sender. Third, excluding incoming channel, the lowest nonfaulty spare dimension is chosen and the flit is routed through that dimension. Finally, if none of the above conditions holds then the incoming spare dimension is chosen and the flit is returned back to the sender.

As discussed in the above paragraph, there are two situations where a header flit may be sent back (also, called backtracked) to the sender. If the network has several faulty nodes/links, a header flit may backtrack several times before finding its path to the destination. Obviously, this backtracking can cause inefficient message delivery, especially for large messages. Our solution makes a serious attempt to improve this situation by using the marked flits. Once a backtracking occurs, the sender places these special flits in the outgoing channel before sending additional data flits. The purpose is to prevent the future data
flits to follow the wrong and longer path taken by the header flit due to node/link failure. The data flits which were already sent before sending the marker, will take the longer path. However, receipt of the marker back at the same sender will end this inefficient routing of flits. From this point onwards, all data flits and the tail flit will follow the new and better path. Thus, the efficiency of the algorithm is increased. A marked flit is generated only if the sender has more data flits to send after receiving the header flit back. Extra send actions are added to Action $S_1$ for sending these marked flits.

4.3 Proof of Correctness

4.3.1 Legitimacy Predicates

$LM$: 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 in the network, while data and the tail flits are still waiting to be transmitted. In addition, we have to carefully eliminate the marked flit to get a valid message. To get all of the flits for a single message $M$, for a path $C$ take:

$$LM \equiv (M = (flits\ delivered) \cup (flits\ in\ transit) \cup (flits\ not\ yet\ transmitted) \setminus (marked\ flits\ in\ the\ network)).$$

4.3.2 Reliable Delivery

In the previous chapter, we proved three liveness properties Algorithm $WRH$ needs to satisfy. The solution needs to satisfy one additional fault-tolerant property as defined below:
The nodes and links can fail or recover.

Lemma 4.1 (Deadlock Freedom) Starting from a configuration that satisfies the legitimacy predicates, the network will not deadlock.

Proof. Deadlock can occur only when there is a cycle in the network channel dependency graph. In a legitimate state our Algorithm \( W\mathcal{R}H_{FT} \) assigns channels either from preferred dimensions or from spare dimensions. In the former case, the assignment is made in strict descending order of dimension. In the latter case, the macro \( \text{IsCycle} \) is used to check for possible cycle formation, and a spare dimension is assigned accordingly. Hence, no cycles can be formed in the channel dependency graph. □

Lemma 4.2 (Livelock Freedom) Starting from a configuration that satisfies the legitimacy predicates, the network will not livelock.

Proof. The additional calculations for routing do not affect the conditions for livelock freedom. So, the proof of Lemma 3.2 remains valid. □

Lemma 4.3 (Starvation Freedom) Starting from a configuration that satisfies the legitimacy predicates, the network will not starve.

Proof. The additional calculations for routing do not affect the conditions for starvation freedom. So, the proof of Lemma 3.3 remains valid. □

Lemma 4.4 (Fault-tolerance) Starting from a configuration that satisfies the legitimate predicates, the algorithm can cope with nodes and links failures/recoveries.
Proof. Due to a failure/repair, our algorithm chooses an alternate, possibly a suboptimal path for the header flit. Hence the proof follows directly from the algorithm.
CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we have studied specific network topologies and routing protocols that can be used for communication between different processors in multiprocessor networks. We proposed two self-stabilizing wormhole routing algorithms for hypercube networks, one without node/link failures and the other with node/link failures and recoveries. In addition to the topology changes, our solutions can deal with many other types of faults, such as, memory corruptions, message losses, message corruptions, and faulty circuits. Detection and recovery of all these faults were implemented in our solutions using the paradigm of self-stabilization. This reaffirms the well-known fact that the technique of self-stabilization subsumes all other fault handling mechanisms. This research demonstrated the application of self-stabilization in designing protocols for massively parallel multiprocessor systems.

Message losses are unpredictable. So, our solutions cannot guarantee reliable delivery if a message loss occurs while transmitting a message. The proposed algorithms satisfy all liveness properties, hence assuring safe delivery of messages in the absence of any faults.

The research initiated in this thesis can be explored further in various directions. Different cost metrics (such as time and bandwidth) can be evaluated both theoretically and experimentally (either by actual implementation on multiprocessor machines or simulation). Other topologies such as cube-connected cycles and torus rings can be studied. We

53
considered only unicast algorithms, meaning there is only one sender and receiver. Fur­
ther research needs to be done in self-stabilizing multicast wormhole routing algorithms for hypercubes.

In the second and final algorithm that handles node/link failures, if the channel rate is higher, then the efficiency will be less. This is because more message fragments will be routed before finding the correct path, and they will be rerouted when the actual fault-free path has been established. More research needs to be done in getting better efficiency while using faster channels. In this algorithm, a special type of flits, called marked flits, is used to avoid taking a false path established by the header flit.
BIBLIOGRAPHY


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


VITA
Graduate College
University of Nevada, Las Vegas

Isai Arasu Marichamy

Home Address:
4223 Cottage Circle, Apt#1
Las Vegas, NV 89119

Degrees:
Bachelor of Science, Computer Science, 2002
University of Madras, India

Thesis Title: Self-Stabilizing Wormhole Routing in Hypercubes

Thesis Examination Committee:
Chairperson, Dr. Ajoy K. Datta, Ph.D.
Co-chairperson, Dr. Sébastien Tixeuil, Ph.D.
Committee Member, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. Thomas Nartker, Ph.D.
Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.