EFFICIENT IMPLEMENTATION OF PARTIAL INTEGRATED NETWORK BARRIERS

(First Draft Version)

BY

BRYAN BARNEY REAGAN
B.S., University of Illinois at Chicago, May 1991
M.S., University of Illinois at Chicago, June 1995

THESIS

Submitted as partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Chicago 2004

Chicago, Illinois
Chapter 1: Introduction and Overview

About this document.

This document is intended to serve two purposes. Firstly, it is a proposal document for the second preliminary defense of Bryan Barney Reagan’s doctoral thesis. Secondly, it is also a rough draft of the thesis document, as requested by the Director of Graduate Studies and the Graduate College.

Introduction and overview of the purpose of this thesis.

The purpose of this thesis is to develop techniques to efficiently implement partial integrated network barriers on packet switched interconnection networks using the previously developed techniques for integrated network barriers.

An integrated network barrier is a specific type of barrier synchronization, which is required to ensure that pre-barrier operations and communications complete before post-barrier operations and communications (Solworth and Stamatopolous, 1993). These new techniques support barriers which are partial barriers in that they do not involve all processors in the network (Birk, Gibbons, Sanz, and Soroker, 1989). Normal partial barrier techniques allow the processors (or processes) to be partitioned into two groups: those participating in the barrier, and those not participating in barrier.

In this thesis, processors are partitioned using the routes over which packets they send and receive traverse, and these partitioning are called regions. A closed region can be created which implement a traditional partial barrier which operates over a set of routes in isolation from the rest of the network. In addition, techniques developed here can also create an open region, which partitions the processors into three groups: those actively participating in the barrier and which are injecting packets; those which receive packets in barrier order but which are not injecting packets nor actively participate in the barrier; and lastly those which are not participating in the barrier and are oblivious to the activity within the region. These regions are created using pickets, which are markings of queues which determine how they are treated during the traversal of a barrier wave. The pickets are of two types, entry pickets and exit pickets, which are defined, developed, and described here. Furthermore, they are being experimentally tested in simulations.
While the specific network being focused upon is a two dimensional mesh topology, the techniques developed are independent of network topology and are instead derived from the queue dependency graph and the partitioning of routes into disjoint subsets called plies. They are adaptable to any interconnection network with an acyclic queue dependency graph.

These techniques are being tested using a simulator which models the behavior of idealized packet switching interconnection networks with: a two dimensional mesh topology, dimension order routing (metropolis routing), virtual cut through switching, and separate turn and rifle input queues. It can be run for different network sizes, traffic patterns, different numbers of virtual interconnection networks (VINs), and can support integrated network barriers using wave patterns from either Solworth and Stamatopolous, or McShane. The simulator, named NetQSim for “Network of Queues Simulator”, is currently 470 kilobytes of C++ written using standard template libraries. NetQSim has been developed since 2001, and replaces the InetSim simulator which the author developed for this work from 1996 to 2001, the code for which had become too large and labor intensive to modify.
Chapter 2: Background

Part 1 Early Barriers and Flush Primatives

Early Synchronization Techniques

Even before the term barrier synchronization was coined, techniques were developed to synchronize concurrently executing processes or processors. Most of these early techniques rely on specific hardware or operating system features. Many of these techniques used counters of one form or another, which were either incremented or decremented when a process (or processor) reached the synchronization point. These techniques all required each individual process to gain access to the counter, either using a lock or semaphore to ensure exclusive access, or using an atomic fetch and add type operation. No only did this mean that the time to execute grew linearly as the number of processes to synchronize grew, but also created contention as they competed for access. In addition, while some used a notification scheme to inform the waiting processes that the all had reached the synchronization point, many of these implementation used “busy waiting”, which would further aggravate the “hot spot” at the counter. Excellent surveys of these early techniques can be found in (Birk, Gibbons, Sanz, and Soroker, 1989) or (Glew and Hwu, 1991).

Origin of the Term: Jordan’s Barrier Synchronization.

The term barrier synchronization, was coined by Henry F. Jordan in 1978 (Jordan, 1978). On page 265, he described a barrier as a division between two stages of a computation, in which “no processor should start the latter until all complete the former (Jordan, 1978).

Jordan’s implementation of a barrier required special dedicated hardware. The machine used message passing with “first in first out” buffering, but did not have direct hardware support for shared memory (Jordan, 1978). Furthermore, it had two pre-existing, independent communications networks: a set of dedicated processor to processor links between select pairs of processors in a fixed topology; and a time multiplexed, common bus linking all processors (Jordan, 1978).

Jordan’s scheme required that each processor have two single bit flag registers, called the barrier flag and report flag respectively (Jordan, 1978). These flags were wired together in daisy chained “and”
and “or” lines (Jordan, 1978). Upon reaching the barrier, a processor set its report flag true and then waited (Jordan, 1978). One processor was designated the controller, and was responsible for coordinating the end the barrier when all report flags were true by clearing the barrier flag (Jordan, 1978). At that time, all processors would then proceed (Jordan, 1978). Jordan also purposed the use of multiple barrier flags so that more complicated protocols could be implemented (Jordan, 1978).

Brook’s Butterfly Barrier.

In 1986, Eugene D. Brooks published a barrier synchronization technique, which was based in part upon Lamport’s Logical Clocks, and which used the pre-existing shared memory (Brooks, 1986). On page 295, Brooks describes a barrier synchronization as “all of the participating processors are required to meet at the barrier before any are allowed to proceed” (Brooks, 1986).

Brook’s barrier uses a two processor barrier synchronization, essentially a handshake using shared memory, as a building block to create larger barrier synchronizations (Brooks, 1986). Each processor has its own barrier flag in shared memory, and each processor in a pair to synchronize executes an identical set of four operations using the flags (Brooks, 1986). For narrative purposes, a processor’s associated flag in shared memory shall be called its local flag, and the flag associated with the processor with which it is synchronizing shall be called the remote flag.

From the point of view of one processor, a two processor barrier synchronization requires four steps: Firstly, it busy waits until its local flag is zero; secondly, it sets its local flag to one; thirdly, it busy waits until it detects that other processor has set the remote flag to one; lastly, it clears the remote flag (Brooks, 1986). Brooks points out that if the flags are distributed on different memory banks, then this can be done without creating a performance threatening, frequently accessed and contested “hot spot” (Brooks, 1986).

Larger barrier synchronizations are performed by performing multiple sets of concurrent two processor barriers (Brooks, 1986). The technique is designed to work optimally for systems with numbers of processors equal to integral powers of two, but is adaptable to systems with other numbers of processors by using the pattern for the next largest power of two, and ignoring the comparisons with non-existent
processors (Brooks, 1986). If the system has N processors, where N = 2^x, and where x is some positive integer, then x = log_2 N iterations are required processors (Brooks, 1986). Let us assume that the processors are numbered from 0 to (N - 1), so that operations may be performed on the processor numbers in order to identify the processor with which it is to be synchronized on a given iteration (Brooks, 1986). Given processor number p in [0, (N-1)], during the jth iteration with j in [1, log_2 N], each processor p performs a two processor barrier with processor number p’, where p’ = p ⊕ 2^j-1 (Brooks, 1986). It is worth noting that if the processor numbers are written in base two, they could be interpreted as the addresses of nodes in a hypercube and the pairs which execute the two processor barrier synchronizations would be immediate neighbors in the hypercube topology.

For example, consider a butterfly barrier synchronization on multi-processor with eight processors, labeled 0 through 7. This would require 3 = log_2 (8) iterations. On the first iteration, the processor pairs which perform two processor barrier synchronizations are: (0, 1), (2, 3), (4, 5), and (6, 7). On the second iteration, the processor pairs are: (0, 2), (1, 3), (4, 6), and (5, 7). On the third and final iteration, the pairs are: (0, 4), (1, 5), (2, 6), and (3, 7). This is depicted in figure 1.

Figure 1. A Butterfly Barrier Synchronization for Eight
In 1988 Debra Hensgen, Raphael Finkel, and Udi Manber published a brief paper which proposed two new techniques: the dissemination algorithm and tournament algorithm (Hensgen, Finkel, and Manber 1988). In addition, they developed some new concepts related to barrier synchronizations and timing.

Firstly, they defined a barrier synchronization with the sentence “No process may pass the barrier until all other processes have arrived at it” (Hensgen, Finkel, and Manber 1988). It is worth noticing that these authors are speaking in terms of processes, not processors. Since a process is essentially an abstraction of a processor executing a control stream, we need not concern ourselves with the conceptual differences. Being aware that a program may need to synchronize an arbitrary number of times, a specific execution of a barrier synchronization was referred to as an episode, and the time for pre-barrier operations was referred to as an epoch (Hensgen, Finkel, and Manber 1988).

The dissemination algorithm was similar to Brooks algorithm, in that if the system has N processes, where N = 2^x, where x is some positive integer, then x = log_2 N iterations were required, during which each process set a flag, and had a flag set by another process (Hensgen, Finkel, and Manber 1988). Unlike Brook’s technique, processes did not work in pairs for an iteration, but rather each process interacted with two separate other processes, one which set its flag, and one of which it set the flag, during the same iteration (Hensgen, Finkel, and Manber 1988). During iteration j in [1, log_2 N], processor p sent a message to processor number p’ = (p + 2^{j-1}) Mod N, resulting in an asymmetric communication pattern (Hensgen, Finkel, and Manber, 1988).

By far the more influential of the two techniques they presented was the tournament algorithm. The tournament algorithm also used flags in shared memory and busy waiting, but reduced the total number of operations by using a tournament tree pattern (Hensgen, Finkel, and Manber, 1988). Just as before, a system which had N = 2^x processes, where x is some positive integer, then x = log_2 N iterations were required (Hensgen, Finkel, and Manber 1988). In this case however, if a binary tree is used only (N-1) shared memory operations are needed to ensure that all processes have reached the synchronization point (Hensgen, Finkel, and Manber 1988). During iteration J from 1 to log_2 N, if a process number has a one in
its $2^{(J-1)}$ place, then it notifies the process with the same number except with one in its $2^{(J-1)}$ place, and wait for barrier completion (Hensgen, Finkel, and Manber 1988). During the Jth iteration, if a process has a number with a zero in its $2^{(J-1)}$ place, but is not yet waiting for barrier completion, then it receives notification from another process (Hensgen, Finkel, and Manber 1988). Ultimately only process zero receives $\log_2 N$ notifications and the barrier synchronization is completed (Hensgen, Finkel, and Manber 1988). Figure 2 shows an example of the fan in tree, but not the notification of barrier synchronization completion.

**Mohan Ahuja’s Flush Primitives**

In 1990, Mohan Ahuja published a paper describing a family of flush operations for use on distributed systems, which guaranteed of message arrival relative to the flush operation, even when over a channel which did not guarantee first-in first-out arrival order of messages (Ahuja, 1990). A channel which can support flush operations was called an **F-Channel** (Ahuja, 1990). Channels are assumed to be one
directional links between two specific processes, and hence those being used by a process were divided into two types: incoming channel and outgoing channels (Ahuja, 1990). A flush operation involves the transmission of a special message from one process to another over a F-channel, and forces certain order requirements on messages sent over that channel (Ahuja, 1990). A flush operation must have one or more of the following properties: (a) a message sent before the flush message must arrive before the message associated with the flush operation; (b) a message sent after the flush message must arrive at its destination after the flush operation (Ahuja, 1990). A flush operation which enforces pre-flush arrival before the flush message but does not guarantee arrival of post-flush messages after the flush message, is called a forward-flush (Ahuja, 1990). Similarly, a flush operation which ensures that post-flush messages arrive after the flush message, but makes no guarantees about the arrival of pre-flush messages before the flush message is called a backward-flush (Ahuja, 1990). A flush operation which segregates pre-flush and post-flush messages is said to be a two-way-flush (Ahuja, 1990). While Ahuja’s paper was not concerned with barrier synchronization per se, clearly it has direct applications for the ordering of message arrival.
In order to describe Integrated Network Barriers, packet switched interconnection networks and their related topics must be clearly defined.

In general a multi-processor is a computer with more than one processor (Tanenbaum, 1999). Each processor has a unique identifier, called a processor address, which allows its location to be determined relative to other parts of the network. Let the uppercase $P$ denote the set of all processors.

An interconnection network facilitates communication between the processors in a multi-processor. A packet switched interconnection network routes packets between the processors. A packet is the smallest routable unit of information (data and/or control), which must contain an address field specifying its destination (Culler, Singh, and Gupta, 1999). A packet typically contains other information about itself in its header, such as its own length. These details may be ignored for our discussion.

While routing through the network, packets are stored in queues. These queues resemble the abstract data type of the same name, in that packets are removed from them (de-queued) in the same order in which they are inserted (en-queued). Unlike the abstract data structures, these queues have a finite maximum capacity, and can be full and hence block, being unable to receive more packets until some have been removed. The end of the queue from which packets are removed is called the head of the queue, and the end where they are inserted is called the tail of the queue. In the literature about routing, it is often the convention to draw a queue as two or more rectangles sharing adjacent long sides, while the rectangle at the tail end has the one long side open. One may think of this as if the queues are ladders, up which the packets climb from the open end at the bottom, to the closed end at the top. This is depicted in figure PSA1.
We shall make three further assumptions about the operations on queues. Firstly, a queue may not allow more than one packet to be inserted at the same time, nor may more than one packet be removed at the same time. Secondly, an individual packet must be completely inserted before it can begin to be removed. Thirdly, if a queue may simultaneously insert one packet while a distinct, separate packet, which has previously been inserted, is being removed.

Let the uppercase letter $Q$ denote the set of all queues in the network in question. There are two special subsets of $Q$ which deserve mentioning. The source queues are those queues by which packets enter the network, and to which no queue in the network may transfer packets. Let $Q_s$ denote the set of all source queues. Similarly, the destination queues are those queues by which packets leave the network, and from which no packet is transferred to another queue in the network. Let $Q_d$ denote the set of all destination queues. Each source or destination queue is associated with a unique processor, which can be viewed as a black box which acts as the ultimate producer and consumer of packets. When we speak of a packet being addressed to a processor, it is actually addressed to a destination queue associated with that processor. Also, when we speak of a packet being from a processor, its point of origin within the network is actually a source queue associated with that processor.
The connections between queues shall be called **links**. While at this time we would assume that this is some type of copper or aluminum connectors with the electronics to drive them, this will be deliberately left abstract. A **phit** is the amount of information a link may transfer simultaneously (Culler, Singh, and Gupta, 1999). A **flit**, or flow control unit, (identified by some sources as a flow control digit,) is the smallest unit of information the transfer of which may be accepted or rejected (Culler, Singh, and Gupta, 1999). The sizes of phits and flits are obviously dependent on the switching technology, and flits are typically more than 1 phit in length. The sizes of packets, as well as the capacities of queues, shall both be assumed to be a discrete number of flits. A **Cycle** shall be defined as the unit of time required to transfer (or refuse to transfer) a flit of information across a link. Hence a packet of \( N \) flits in size would require \( N \) cycles to transfer from one queue to another. For our purposes the size of a flit and the length of a cycle shall be deliberately left abstract, rather than given artificial, and eventually laughably dated, values in fractions of seconds or multiples of bytes.

A packet switched interconnection network contains the following elements. Firstly, **sources** and **drains** of packets. Typically these would be processors which send packets to each other. For our purposes, a processor may be thought of as both a source and drain of packets. Secondly, **switches** route packets between the sources and drains of packets. Thirdly, **links** act as the connections between the switches, sources, and drains of packets.

The logical, but logical, not necessarily physically spatial, arrangement of the switches and links is called to **topology** of the network (Zargham, 1996). The term topology is used since the logical arrangement may be in a shape not physically possible, such as a five dimensional hypercube. Some of the more common network topologies include: Hypercube, N-Dimensional Mesh, Iliac, Torus, and the Omega network (Zargham, 1996). Please see figure PSA2 to see a Two-Dimensional Mesh topology.
A **channel** is defined as a physical link, together with the input queue and output queue which it connects (Culler, Singh, and Gupta, 1999). The **link-level protocol** specifies when and how transfers are to be made across the channel (Culler, Singh, and Gupta, 1999). **Virtual channels** are created when using multiple sets of queues which multiplex the use of a common physical link, to implement independent logical channels (Culler, Singh, and Gupta, 1999). These are depicted in figure PSA3.
Virtual channels play a vital link in the study of routing and integrated network barriers. For example, since adaptive routing can deadlock, T. M. Pinkston at UC San Diego has proposed using one set of virtual channels for adaptive routing in order to achieve higher performance, and another set to use a deadlock free routing scheme which can be used to drain packets from deadlocked regions of the network. (Pinkston, 1999).

In Jerry Stamatopolous’s initial work with Integrated Network Barriers, he used complete sets of virtual channels, in which all of the internal queues of the switches were replicated into he called “plies” of network (Stamatopolous, 1996). Eric McShane’s work on perfectly pipelineable barrier waves, his network require a scheme in which the network used multiple virtual channels to propagate the waves, which he
called “virtualizing the buffers” (McShane, 1996A). The differences between these concepts and virtual channels are nuanced and not particularly clear at times.

A switch contains two types of queues: input queues which receive packets from outside sources, and output queues into which packets bound for external destinations are transferred. A very simple switch for a two dimension switch is depicted in figure PSA4. Note that from the point of view of the switch, the processor is simple another external source and destination for packets.

A common optimization in switch design is to split input queues into separate turn queues and rifle queues. The rifle turn only accepts that packets which are not changing direction and pass through “like a rifle shot.” In contrast, the turn queue accepts the packets which are changing direction at this switch. This can give greater concurrency and prevents packets which are not turning from being forced to wait behind

![Figure PSA4. Layout of a Simple Switch](image-url)
ones that are turning. This arrangement is similar an overpass on a highway.

The Route Step Function, Routes, and Queue Dependence

Now that the general concepts of queues and switches are understood, routing and queue dependence must be described. Given a packet at the head of an input queue in a switch, the control mechanism of the switch must evaluate into which output queue to transfer the packet in order for it to ultimately reach its final destination. This decision process is partially modeled by the route step function, which takes as its parameters the packet’s current location and ultimate destination, and returns the set of queues into which it may be transferred. This obviously depends upon the network topology, and the routing algorithm being used.

Definition. **Route Step Function.** $\sigma : Q \times Q \rightarrow P(Q)$. Specifically, for a given queue $x$, where $x \in Q$, and a given queue $y$, where $y \in Q$, the route step function, denoted $\sigma (x, y)$, where $\sigma(x, y) \subseteq Q$, specifies the set of queues to which a packet at the head of queue $x$ may be directly transferred in order to ultimately route to $y$.

Note that in a deterministic routing scheme, the route step function always produces sets of no more than one element, and furthermore if there is no legal route from $x$ to $y$, $\sigma (x, y)$ will be empty. A route then shall be considered an ordered chain of queues which a packet would be inserted into in order reach the last queue, based on legal values of the route step function.

Definition. **Route**: a route from queue $q_1$ to queue $q_n$, shall be defined as a string of queues, $[q_1, q_2, \ldots, q_{n-1}, q_n]$, where $q_1$, to $q_n$, are all in $Q$, and in which $\forall x \in [1, n - 1], q_x \in \sigma(q_{x-1}, q_n)$. A route is said to be from its initial queue $q_1$ and to its final queue $q_n$.

The term routing algorithm refers to the overall scheme which selects the strategy to generate routes through the network as a whole (Tanenbaum, 1999). The route step function is simply the detail level decision of selecting the next queue a packet should route to, essentially implementing the routing algorithm at the lowest possible level.
There are two useful sets derived from the route step function which are fundamental to integrated network barriers. These are the successor set and predecessor set. For a given queue, its successor set is the set of all queues which may route to that queue. Similarly, the predecessor set is the set of queues which may route to the queue in question. These are more formally defined as follows.

**Definition.** **Successor Set.** For a given queue \( x \) in \( Q \), the successor set of \( x \), denoted \( \text{Succ}(x) \), is the set of all queues to which a packet may be transferred from \( x \). Formally, this is denoted as: \( \text{Succ}(x) = \bigcup_{y \in Q} \sigma(x, y) \).

**Definition.** **Predecessor Set.** For a given queue \( x \) in \( Q \), the successor set of \( x \), denoted \( \text{Pred}(x) \), is the set of all queues which may directly transfer a packet to \( x \). Formally, this is denoted as:

\[
\text{Pred}(x) = \{ \forall y \in Q \mid x \in \text{Succ}(y) \}.
\]

![Figure PSA5. Sample Network of Queues](image)

**Example.** Consider the arrangement of queues in figure PSA5. Let us assume that the queues depicted are the only ones in the network, and that the arrows show all legal transfers between queues. Under those circumstances, the following conditions apply.

\[
\text{Pred}(d) = \{a, b\}, \quad \text{Succ}(a) = \{c, d\}, \quad \sigma(a, c) = \{c\}, \quad \sigma(a, e) = \{c\}, \\
\sigma(a, f) = \{d\}, \quad \sigma(d, f) = \{f\}, \quad \sigma(d, e) = \{\}.
\]
Using successor and predecessor sets allows us to formally define some traits of source and destination queues.

**Lemma:** \(q \in Q_s \implies \text{Pred}(q) = \{\}\)

**Proof:** By the definition of source queues, if \(q\) is in \(Q_s\), then there are no queues which may route into \(q\). Hence \(q\) may have no predecessors.

**Lemma:** \(q \in Q_d \implies \text{Succ}(q) = \{\}\)

**Proof:** By the definition of destination queues, if \(q\) is in \(Q_d\), then there are no queues to which it may packets. Hence \(q\) can have no successors.

**Queue Dependence and Queue Dependence Graphs**

Queue dependence is a key concept for understanding routing. Essentially it abstracts routing to graph theory, and provides formalized, conceptual handles for routing concepts.

**Definition.** **Queue Dependency** – A queue \(A\) is said to depend upon queue \(B\) if packets can move from queue \(B\) to queue \(A\) while routing on some valid route (Pifare, Gravano, Fleperin and Sanz, 1994).

Queue Dependence depends upon the network geometry and the routing algorithm being used.

**Definition.** **Queue Dependency Graph** (QDG) – a directed graph \(G=(V,E)\), where

- each vertex in \(V\) corresponds to a queue in the switching system, and
- each edge \((u, v)\) indicates that a packet may route from the queue corresponding to vertex \(u\) to the queue corresponding to vertex \(v\) (Pifare, Gravano, Fleperin and Sanz, 1994).

One of the reasons why queue dependency graphs are so significant is that they can be used to design deadlock-free routing schemes.
Lemma. An acyclic Queue Dependency Graph corresponds to a deadlock free routing scheme (Pifare, Gravano, Fleperin and Sanz, 1994).

Using the terminology of Graph Theory, an arbitrary route, which may include cycles, can be viewed as a “walk” on the queue dependency graph, and a deadlock free route, which is without cycles, may be viewed as “path” on the queue dependency graph. See (Chartand, 1985) for definitions of these terms.

Before we can specify the abstract model of a network we shall base out work upon, we need to clarify two more concepts: dimension routing and virtual cut through switching.

Definition. Dimension Routing – a deadlock free routing scheme for n-dimensional meshes, in which packets are routed minimally to the correct coordinate in each dimension in turn, in a fixed order (Tanenbaum, 1999). Dimension routing is also called metropolis routing, or Manhattan Routing.

Definition. Virtual Cut Through Switching - a packet switching technique in only the head of a packet may block (Tannenbaum, 1999).

In other words, virtual cut through switching only transfers a packet in its entirety. No partial transfers are allowed.

Our Network Model Shall Assume: Packet switching using virtual cut through switching; Two dimensional Mesh (x, y) geometry; Dimension routing, routing first in the Y dimension, and then on X; Separate rifle and turn input queues on external wires, and separate X-turn and Y-turn input queues from the processor; The potential use of multiple piles.
Having defined the concepts of a queue, queue dependency graphs, and routes, it is necessary to create a higher level of abstraction which encapsulates these concepts. Specifically routes will be used to build up the concept of a ply and a virtual interconnection network.

The term ply was first used by Jerry Stamatopolous in his dissertation (Stamatopolous, 1996). He was essentially turning every link in an interconnection network into some number of virtual channels, which he then duplicated all of the nodes internal switching, once for each virtual channel (Stamatopolous, 1996). He then treated each complete collection of virtual channel and the layer of switching which serviced the queues at the ends of the links as a “ply” of network (Stamatopolous, 1996). His plies were essentially parallel versions of the complete network, and supported all routes between all valid source-destination processor pairs.

Our concept of ply is going to be more general. We are not going to require that an individual ply contain a route between all possible source–destination pairs. However, we are going to require that if a route contains one queue within a given ply, then it only contains queues in that ply.

**Definition. Ply** – the union of a set of routes, which has the property that a route which traverses a queue within a ply, exclusively traverses queues within that ply.

The fact that a route must exclusively traverse queues in one ply shall be called the **Total Route Embedding Property**. This becomes important because it guarantees that wave traversing a ply cannot escape to another ply. It also means that a wave sweeping a ply cannot be circumvented by packets routing on another ply.

Since a single ply is not required to contain routes between all valid source-destination pairs, we need a conceptual equivalent of a sub-network which does. Furthermore, it is useful to reason about it
having distinct components which are in fact plies and keep the Total Route Embedding Property. We shall call such a collection of plies a Virtual Interconnection Network, and an actually interconnection network may contain an arbitrary number of virtual interconnection networks.

Definition. (VIN) **Virtual Interconnection Networks** - a minimal collection of plies which contains at least one route between all valid source destination pairs of processors.

The plies discussed in (Stamatopolous, 1996) are in fact virtual interconnection networks consisting of a single ply. Furthermore, the routes created using the “virtualized buffers” discussed in (McShane, 1996A) are closely related.
Informally, Integrated Network Barriers, sometimes referred to as simply barriers, are a refinement of barrier synchronization. They essentially combine the features of barrier synchronization with the operation completion aspects of Ahuja’s Flush primitives. This makes them then ideal for synchronization in packet switched interconnection networks, were the state of resources somewhere in the network may depend upon a packet or message which is currently in transit within the network.

**Definition. Integrated Network Barrier – (INB)** a barrier implementation in which the completion of pending memory operations is integrated with barrier synchronization (Solworth and Stamatopoulos, 1993), (Birk, Gibbons, Sanz, and Soroker, 1989).

The INBs of Birk, Gibbons, Sanz, and Soroker

While the first published use of the term Integrated Network Barrier would not be coined for another four years the earliest proposed barrier implementation using barrier markers and wave fronts of which the author is aware was developed by IBM and described in (Birk, Gibbons, Sanz, and Soroker, 1989). These barriers were originally designed for use on a multistage dancehall networks, were all of the processors were located at one side of the network of switches, and all of the memory modules were located on the opposite end of the network (Birk, Gibbons, Sanz, and Soroker, 1989). Please see figure INB1 to see an example of such a network. The flexibility of the techniques developed allowed them to be extended later in the paper for use on other network topologies, including a hypercube and mesh network (Birk, Gibbons, Sanz, and Soroker, 1989).

In this implementation special packets, normally called barrier markers, (but referred to here as a “barrier messages”), were inserted into each input queue into the network by each processor (Birk, Gibbons, Sanz, and Soroker, 1989). Barrier markers act as dividers within the network, segregating
packets sent before, called **pre-barrier** packets, from those sent after the barrier, called post-barrier packets. In this sense they act like the flush messages in Ahuja’s Flush primitives.

In order to perform a barrier using this protocol, all a processor (source of packets) needs to do is complete pre-barrier operations and inject pre-barrier packets, insert a barrier marker into its queue into the network, and then start its post barrier operations and injecting its post barrier packets (Birk, Gibbons, Sanz, and Soroker, 1989).

The monotonic routing nature of the dancehall switch made coordination of the barrier markers relatively simple to see. Within each switch, there are two input queues and two output queues. When barrier markers reached the heads of both of the input queues, then clearly no more pre-barrier packets can arrive (Birk, Gibbons, Sanz, and Soroker, 1989). Hence at that time, new barrier markers were inserted into each output queue, and then the markers at the heads of the input queues were removed (Birk, Gibbons, Sanz, and Soroker, 1989). This advancing wall of barrier markers hence progressed from queue to queue, and hence was referred to as a **wave** (Birk, Gibbons, Sanz, and Soroker, 1989).

At the memories, the arrival of a barrier marker indicates that all pre-barrier packets have arrived, and that only post barrier packets will arrive in the future (Birk, Gibbons, Sanz, and Soroker, 1989).

Consider the network in figure INB1, which depicts a partially completed barrier on an omega network. Processors are depicted as circles on the left, and memories as hexagons on the left. Due to the space considerations, queues within the switches are represented without internal lines. A processor, memory, or queue which has been swept by a barrier marker is drawn with a diagonal line through it, and one with a barrier marker at its head, being on the **wave front**, has two diagonals drawn through it making an “X”. This network has completed the barrier as far as possible until the third processor from the top inserts a barrier marker. At that time the third switch from the top will have markers at the heads of both of its input queues, and will be able to insert markers into its output queues, and then remove the markers from its input queues. The markers will be forwarded across the links and into the two switches at the top.
of the second column, which will each then repeat the process. After those last two switches in the second column have completed propagating the wave, the all four switches in the second column will follow suite by inserting markers into their output queues, deleting the markers at the heads of their input queues, and forwarding the markers to the memory modules in the rightmost column.

The authors then attempted to generalize this INB technique for other network geometries, which may require the use of multiple waves to sweep all possible routes through the networks switches (Birk, Gibbons, Sanz, and Soroker, 1989). While their solution is straightforward, clean, and optimal for an
Omega network, their solutions for other networks were generally not efficient, but were rather existential in nature, and without sufficient detail to implement. In particular, their solution for the most studied topology, the two dimensional mesh, was particularly inefficient. For their sample implementation on a two dimensional mesh, they suggested using two waves which each swept the network twice, once from a corner to the opposite corner and back to the initial corner, for a total of four traversals of the network (Birk, Gibbons, Sanz, and Soroker, 1989). Please see figure INB4 for an example of a corner wave.

In the event of a partial barrier, processors which were not participating in the barrier were still required to insert barrier messages, although these were special barrier messages could be allowed to be overtaken a data packet or to overtake a data packet, with the final wave being unable to be overtaken or overtaking packets (Birk, Gibbons, Sanz, and Soroker, 1989).

While the solutions for other networks and partial barriers left a great deal to be desired, Birk, Gibbons, Sanz, and Soroker did in fact give the three necessary and sufficient conditions for correct operation of an integrated network barrier, which they used to prove the correctness of their omega network solution (Birk, Gibbons, Sanz, and Soroker, 1989). These conditions are:

1. **Flushing Property** – A barrier message traverses a link only after all pre-barrier packets (Birk, Gibbons, Sanz, and Soroker, 1989).
2. **Restraining Property** – A barrier message traverses a link before all post-barrier packets (Birk, Gibbons, Sanz, and Soroker, 1989).
3. **Clear Route Property** – The path of a pre-barrier packet is always clear of post-barrier packets (Birk, Gibbons, Sanz, and Soroker, 1989).

**Stamatopolous and Solworth’s Edge Wave Integrated Network Barriers**

Over the period from 1993 to 1996, Jerry Stamatopolous and Jon A. Solworth developed an integrated network barrier technique which used the queue dependency graph to design waves which
traversed an n-dimensional mesh using two waves per spatial dimension (Solworth and Stamatopoulos, 1993, 1994, 1995, 1996, 2002), (Stamatopoulos, 1996). Ironically, Jorge Sanz was an author in both the first INB paper (Birk, Gibbons, Sanz, and Soroker, 1989) and the first paper of queue dependency graphs Pifare, Gravano, Fleperin and Sanz, 1994), but did not see the application himself.

Edge wave integrated network barriers operate on mesh network which use dimension order or metropolis routing, which has an acyclic queue dependency graph (Solworth and Stamatopoulos, 1993). Essentially, since dimension ordering resolves routing one dimension at a time in a fixed order, the queue dependency graph cannot contain a cycle. By using barrier waves which sweep the network from one edge to the opposite, in routing order, all queues are swept free of pre-barrier packets (Solworth and Stamatopoulos, 1993, 1994, 1995, 1996, 2002), (Stamatopoulos, 1996).

In addition to the edge wave pattern, these new barriers feature a new device for keeping track of the number of barrier markers which had been inserted into and removed from a queue, the **barrier marker counter** (Solworth and Stamatopolous, 1993). Associated with the head of each queue, a **head barrier marker** counter keeps count of the number of barrier markers which had been removed from the queues head (Solworth and Stamatopolous, 1993). In addition, each queue has a **tail barrier marker counter** at its tail, which counts the number of barrier counters which have been en-queued at the queues tail (Solworth and Stamatopolous, 1993). Packets are then only allowed to transfer between queues if the originating queues head barrier marker counter has the same value as the receiving queues tail marker counter (Solworth and Stamatopolous, 1993). This allows the markers to be immediately de-queued when they reached the head of the queue, freeing up the space, and also this can allow greater concurrency at splits (Solworth and Stamatopolous, 1993).

**Example.** Consider the arrangement of queues in figure INB2. If a barrier marker is at the head of queue c, but not at the head of queue a. Using the rules from (Birk, Gibbons, Sanz, and Soroker, 1989) the barrier wave cannot propagate into either b or d until there are markers at the head of both a and c. By using barrier marker counters, the barrier marker can be immediately propagated into queue d. In addition, if a packet
behind the marker in queue c should route to queue d, it may do so, although if a packet to route to queue b is behind the marker, then it and all packets behind it must still wait.

*Example.* Consider a simple two dimensional mesh topology, with a single ply VIN, with a single input and output queue from each neighbor in the grid and from the processor, and upon which dimension order routing is implemented routing on Y first, then X.

- Consider a switch on the lower edge (minimal y ordinate) of the network, where the input queue from the processor has just had a barrier marker propagate to its head. Since the output queue in the y positive direction has only the input queue from the processor as a predecessor, the wave may immediately propagate in the y positive direction. A node on the upper edge of the network with a barrier marker at the head of its input queue from the processor may similarly propagate the wave in the y negative direction.

- Consider a switch on the left hand side (minimal x ordinate) of the network, where the input queue from the processor has just had a barrier marker propagate to its head. It may not propagate the barrier in the x positive direction until all of the predecessors of the output queue in the x positive directions have been swept by a wave for this barrier. This is testing by determining if their head
barrier marker counters are all greater than the tail barrier marker counter of the output queue in the x positive direction. In this case, the predecessors are: the input queues from (1) the processor, (2) the y-positive direction, (3) the y negative direction. At such point in time when all these predecessors have been swept, then the barrier wave may be propagated. A node on the rightmost (maximal x ordinate) edge will behave symmetrically with respect to propagating the wave in the x negative direction.

- A switch in the center of the network will forward a wave into an output queue only after all of that queue’s predecessors have been swept, as described above. Output queues in the directions have at most two predecessors: the input queue from the processor and the input queue from the opposite y direction. Output queues in the x direction can have four predecessors, being the input queue from the processor, the input queues from both y-directions, and the input queue from the opposite x –direction.

- All switches forward the wave to the processor via the processors output queue only after all predecessors of the output queue to the processor have been swept. This requires all four input queues from the four cardinal directions.

The primary focus of Solworth and Stamatopolous’s work was to use INBs as a traffic congestion control device in the networks. The time to complete the barrier, called barrier latency, threatened to undermine performance and counteract the benefits (Stamatopolous, 1996). In order to hide barrier latency, Stamatopolous used four virtual interconnection networks, and cyclically alternated active VIN, upon which the processors sent data packets, followed by a barrier to flush the VIN before switching to another VIN (Stamatopolous, 1996). On the downside, if the barriers must be coordinated at the processor level, since pre-barrier packets on one VIN may arrive at the processor after post barrier packets on another, but this is not difficult to coordinate (Stamatopolous, 1996). (Actually, the term VIN was coined eight years after Stamatopolous did this work. He used the term ply instead of VIN, since his VINs consisted of a single ply each (Stamatopolous, 1996).) To see this graphically, consider the gant chart in figure INB3. Please note that the barrier frequency would have to be tuned based on the packet and barrier latencies and
so that barriers would complete on a given VIN before post barrier data packets were injected on that VIN. Otherwise a packet might become trapped behind the wave and suffer large latencies.

McShane’s Perfectly Pipelinable Barriers and Corner Waves

Eric McShane wrote two papers on barriers in 1996. In the first of these papers, he proposed a variation on integrated network barriers which he called perfectly pipelineable barrier waves. Firstly, he noted that a single barrier could be composed of waves propagating on multiple plies, which he referred to as “virtualizing the buffers” (McShane, 1996A). Secondly, he noted that by reducing the links in the queue dependence graph within a ply, he could prune the graph in such a way as to traverse all queues in the ply with a single wave of barrier markers (McShane, 1996A). In particular, he noted that by only allowing packets to move in one direction per dimension, a wave starting from one corner and traversing to the opposite corner of a two dimensional mesh could sweep all queues traversed by routes of packets moving in the same directions as the wave (McShane, 1996A). Furthermore, since these waves moved on separate plies, they did not have to wait for each other, but rather run independently in their own plies, unlike the previous two dimensional mesh barriers by Stamatopolous and Solworth.

**Figure INB3. Gant Chart of Staggered Data and Barriers on a Four VIN network**
In addition, pairs of waves from opposite corners used disjoint sets of queues and link, and could theoretically be embedded in the same ply. For example, the northeast (X positive Y positive) and southwest (X negative Y negative) waves could run on one ply, and northwest (X negative Y positive) and southeast (X positive Y negative) could run on another. Packets could then be routed the one of the two plies which had a wave which swept its route, and the whole network flushed by a barrier using all four waves (McShane, 1996A). Please see figure INB4 which show a sample northeast (X Postive Y Positive) wave on a four by four, two dimensional mesh network.

While McShane’s solution for speeding up the barrier by maximizing independence of the waves was elegant, his method for partial barriers was complex and is possibly impractical to implement. Essentially he would use markers from the waves pre-existing global barriers which would be marked as being also used in the partial barrier, by the addition of a “done” marker as they entered the region to be synchronized (McShane, 1996B). In addition, the waves used would be selected ahead of time from different pre-scheduled global barriers, based on how far the edge of the region was from the corner in which the wave originated, in order to ensure that the waves entered the region at the same time (McShane, 1996B).

McShane’s partial barriers have a number of issues. Firstly, it would be extremely difficult to pre-schedule the barriers ahead of time, due to dynamic changes in network latency. If the global barriers arrive too early, then the rest of the network is delayed while the partial barrier completes. If the global barriers arrive too late, then partial barrier is held up. It is desirable to have the partial barrier have minimal, preferable no, interaction with the rest of the network.
Figure INB4. North East (X Positive Y Positive) Corner Wave
An Extended Example: A Linear Network of Three Nodes

In order to tie together the concepts of VINs, plies, and barrier waves, we shall demonstrate the simplest possible network with different paths. A network with a single node has no need of a switch, and one with two nodes simple has two channels communicating in each direction, and hence lacks any real opportunity to investigate switching. Assume that we have a network of three nodes connected in a straight line. These nodes will have processors numbered 0 through two, and we shall assume that the spatial ordering of the scheme will be named after the X axis, hence nodes 1 and 2 are to the x positive direction of node 0, and node 1 is in the x negative direction of node 2. These two directions shall be abbreviated “X+” for x positive and “X-” for x negative in diagrams. Furthermore, queues which are associated with the processor shall be marked as “Proc”. In addition, a queue polarity are considered to be Input (abbreviated In) if they receive packets being routed into the switch, and output (abbreviated Out) if they are used to route packets out of the switch to another switch or to the processor.

Figure 1DX1. One Dimensional Array In a 1 Ply VIN
As is generally the case, the processor input queues are source queues, and the processor output queues are destination queues. Hence a queue may be uniquely specified by a triple listing its node number, direction, and polarity, followed by an optional dash and ply number if a queue position is duplicated between to plies, as may sometimes be required in a multi-ply VIN. For example (1, X-, Out) denotes the queue on node number 1, in the X- side, used to route packets out. In addition, (1, Proc, Out)-1 denotes an output queue to the processor on node 1 in ply number 1 in a multiple ply VIN. As per our standard assumptions, all nodes may generate packets addressed to any other node in the network, and may also receive packets from any other node in the network. Messages from a node to itself are not allowed at the network level, as they may introduce circular dependencies. The queue dependency diagram for a single ply VIN version of this network is show in figure 1DX1.

Now consider a simple barrier, using the techniques of (Solworth and Stamatopolous, 1993) on this 1 ply VIN. Please recall that barrier markers are placeholders within the queues which separate pre-barrier and post-barrier packets, and their insertion or removal from a queue indicates the presence of the wave front. As the processor on node 0 enters barrier, it inserts a barrier marker, into (0, Proc, In) after the last of its pre-barrier packets. Since the queues always route the packets in FIFO order, when the marker arrives at the head of (0, Proc, In), all pre-barrier packets have already been routed into (0, X+, Out). The packets and marker then incremental proceed in order through (0, X+, Out) into (1, X-, In). When the marker eventually reaches the head of (1, X-, In) and is de-queued, a new marker cannot be inserted into (1, X+, Out) until a marker has also reached the head of (1, Proc, In), since a marker cannot be inserted into (1, X+, Out) until all of its predecessor queues have been traversed by a marker. Since the arrival of the marker also indicates that all pre-barrier data packets from the processor on node 1 have already been routed, the propagation of the wave into (1, X+, Out) guarantees that all pre-barrier packets that could route into (1, X+, Out) have already been routed. After a marker has been inserted into (1, X+, Out), the packets and markers proceed in order through (2, X-, In) and (2, Proc, Out) to the processor on Node 2. Note again that the arrival of the barrier marker at the processor on node 2 guarantees that all pre-barrier packets which would route to node 2 have already been delivered.
A barrier wave starting at Node 2 proceeds in the same manner, just in the opposite direction. It is worth noting that the two terminal nodes, on the far ends of the network, can only receive packets from one direction, and only need to receive a marker (or wave front) from that direction from which they receive traffic in order to complete. In fact, a node may have received waves and completed the barrier before it has started its own wave.

The queue (1, Proc, Out) does not receive a barrier marker until both (1, X+, In) and (1, X-, In) have had barrier markers reach their heads. Since this queue can receive packets from both directions, it requires two wave fronts, one from each direction, to insure that it have received all of the pre-barrier packets which route to its processor.

Now, consider if the VIN is partitioned into two separate plies, based on the direction of the flow of packets. Ply 0 contains the queues required to route packets in the X negative direction, and ply 1 contains the queues required to route packets in the X positive direction. In the case of node 1, duplicates of the input and output queues for the processor are needed, as node 1 must send and receive packets on both plies. Hence (1, Proc, Out)-0 represents the processor destination queue for node 1 on ply 0, which would receive packets routed to node 1 from node 2, which would have been routed in the x negative direction. This restructuring of the network is depicted in figure 1DX2.

Upon inspection of figure 1DX2 it should be obvious that a barrier must be implemented by two separate waves, one per ply, following the conventions of (McShane, 1996). Furthermore, since the two plies are essentially independent, waves do not interact within the switches, but rather must complete separately at the processors where the waves arrivals are coordinated for barrier completion.

Now consider a simple barrier on this newly re-configured network. As the processor on node 0 enters barrier, it inserts a barrier marker, into (0, Proc, In) on ply 1 after the last of its pre-barrier packets. The wave in the form of a barrier marker progresses through (0, X+, Out) and (1, X-, In). From there the wave is inserted into (1, Proc, Out) -1, and the wave front has arrived at the processor 1. At this time, both
node 0 and node 1 have entered the barrier on ply 1, independent of the action on ply 0. However, the wave may not propagate to \((1, X+, Out)\) until a marker has also been received from \((1, Proc, In)-1\). When the processor on node 1 has entered the barrier and inserted a barrier marker into \((1, Proc, In)-1\), and it has reached the head of the queue, then the wave may progress through \((1, X+, Out), (2, X-, In), and (2, Proc, out) -1\).

A similar wave would traverse ply 0 in the opposite direction. No processor would complete the barrier until it has participated in both waves. Note that node 0 and 2 each initiate a wave on the direction which they send, and receive a wave from the direction from which they receive packets. Node 1, which sends and receives packets in both directions must also insert and receive a barrier marker on each ply.

Figure 1DX2. One Dimensional Array In a 2 Ply VIN
Chapter 3: Reagan’s Pickets, Closed and Open Regions

Now that we have the tools developed by others laid out before us, our purpose now is to develop a set of techniques for implementing partial integrated network barriers on a packet switched interconnection network. For our purposes we shall assume that the partial barrier takes place over routes contained in some number of plies, all of which are contained in a single virtual interconnection network. We shall not make any assumptions about the existence or properties of other virtual interconnections in the actual network. Furthermore, the techniques given here are general enough that they can be used with the barrier protocols in (Birk, Gibbons, Sanz, and Soroker, 1989), (Solworth and Stamatopoulos, 1993), or (McShane, 1996A).

Given that P is the set of all processors in the network, let S and D be two subsets of P. The subset S contains sources processors which send packets and will be actively participating in a partial barrier by sending barrier markers. The subset D contains the processors to which the processors in S send data packets and which will be observers to the barrier, and which will receive the packets sent to them in barrier order.

Definition. Region – within a given virtual interconnection network, the set of processors participating in a barrier (S), and the queues within the virtual interconnection network which are traversed by all routes between valid source-destination pairs of processors in the region.

There are two types of regions, classified by the relationship between sets of sources and destination processors. These types are:

- **Closed Region** – a region in which all packets generated within the region stay within the region. These have: $S = D$.

- **Open Region** – a region in which packets generated within the region may also route to processors outside the region, which do not participate in the barrier. These have $S \subset D$. 
One may think of a closed region as operating in isolation from the rest of the network. In contrast, while an open region only synchronizes the processors within S, it also emits packets into other part of the network, possibly the entire network.

Before proceeding, we need to define some notation about regions. For a given VIN, let R denote the set of all routes between all valid pairs of source-destination processors, and let Q denote the set of all queues traversed by all routes in R. Let $R_{SD}$ denote the routes in a region of the VIN from processors in S to those in D, and let $Q_{SD}$ denote the queues traversed by the routes in $R_{SD}$. The queues in $Q_{SD}$ are said to be in the region participating in a partial barrier.

If we simply data route packets over a subset of the routes in the network, then the routes take care of themselves. The essential characteristic of a region is how they allow a barrier to take place within them without interfering with the operations outside the network. This can be accomplished by isolating the queues inside the region from those outside, essentially temporarily disconnecting their queue dependency graphs from the queues outside the region. Such queues outside the region, which have the potential to interact with those inside the region, but are to be ignored are marked as being pickets. There are two types of pickets, based upon their specific functions: entry pickets and exit pickets.

Entry Pickets

In a closed region, an entry picket marks queues on the outside boundary of a region, which would normally be predecessors of one or more queues within the region. In this case, they are to be ignored for purposes of checking predecessors for barrier joins. In an open region, entry pickets are required on all unused potential predecessors of any queue traversed by a route of a packet originating from inside the open region. Ironically, open regions can have many more entry pickets than closed regions. A sample entry picket marked with its conventional symbol is shown in figure RP1.

Definition. Entry Pickets – a queue a is marked as an entry picket when:

$$a \in Q \setminus Q_{SD} \land \exists b \in Q_{SD} | b \in \sigma(a, b)$$
Exit Pickets

Exit pickets are used to mark queues which could be predecessors of queues in a closed region, but which are not in the region and into which barrier waves are not to propagate. Exit pickets are not currently used in open regions, but might prove useful if we wanted to prevent a wave from propagating outside the queues in $Q_{SD}$. Please see figure RP2 to see an example of a queue marker as an exit picket.

**Definition. Exit Pickets** – a queue is $b$ is marked as an exit picket when:

$$b \in Q - Q_{SD} \land \exists a \in Q_{SD} | b \in \sigma(a, b)$$

Consider a closed region. All input queues on all switches not in the region but adjacent to queues inside the region are marked with exit pickets to prevent barrier waves inside the region from escaping into the outside network. Furthermore, all output queue on switches adjacent to the region which might route packets into queues inside the region are marked with entry pickets, so that they are ignored when predecessors are inspected to determine if its time to propagate a wave. In effect, the processors inside the
closed region are completely isolated from the rest of the network on the VIN in which the region has been constructed.

Figure RP2. An Exit Picket

Open regions are much more complicated. In order for barriers to work inside a closed region, it requires the same entry pickets on the perimeter as a closed region. In addition, all routes from the input queues associated with the processors in S to all output queues associated with the input queues in D must be isolated with entry pickets on all potential predecessor queues outside $Q_{SD}$ which might interfere with the barrier wave propagating along those routes. For example, every input queue from every processor not in S, but which is a potential predecessor to a queue in $Q_{SD}$ is marked as an entry picket. A description of the needed entry pickets to embed an open region in the center of a two dimensional mesh will be given later in this document.

Please consider the omega network depicted in figure RP3. The processors, queues, and memory modules in a closed region, which have been traversed by a barrier wave in the style of (Birk, Gibbons, Sanz, and Soroker, 1989), have been marked with a diagonal slash. Note that the queues marked with the circle bisected with a horizontal line are entry pickets and were ignored for barrier propagation. In addition, please note that the queue marked with a circle with a cross superimposed are exit pickets, and the wave was not propagated to these queues. If these exit pickets were removed, then the wave would have propagated to all of the memory modules and the region would have been an open region.
It is also worth noting that it the figure the processors and memory modules were contiguous. This is not required and the mechanisms would have worked regardless.

Why Are Open Regions Interesting?

Open regions are interesting because the performance cost of executing a barrier is limited by the size of the region, but the packet emanate from the open region, into a potentially much larger portion of
the network, and arrive sorted in by barrier order. Consider the potential in a very large network to have transmitted packets which arrive in guaranteed order relative to fixed events.

**An Extended Example: Open Region in the Center of a Two Dimensional Mesh**

Consider a VIN on a two dimensional mesh, using dimension order routing which resolves the Y dimension first, and then the X dimension. Furthermore, we shall assume the integrated network barrier techniques of (Solworth and Stamatopolous, 1993) to be in use. Consider a rectangular open region located in the center of this two dimensional mesh. Counting the region itself, the mesh is partitioned into nine different areas, which shall be enumerated similarly to the keys on a telephone keypad, and which is depicted in figure RP4. We shall assume that S corresponds to the processors in area 5, and D corresponds to all processors in the entire network. Now area 5, shown with crosshatching, is the open region itself, and areas 1-4, and 6-9 correspond to portions of the network in different spatial locations relative to the network. We shall assume that the open region (area 5) runs from LowerX to UpperX in the X-direction, and LowerY to UpperY in the Y direction. Assume that routing follows the conventions of metropolis routing, and is resolved in the Y-direction first, and then the X- direction. The following areas have the following relationships to the open region (area 5) : areas 1-3 are above (Y+) the open region; areas 7-9 are below (Y-) the open region; areas 1, 4, and 7 are to the left of (X-) the open region; and areas 3, 6, and 9 are to the right of (X+) the open region.

The following rules describe the placement of entry pickets to allow correct operation of the open region.

1. Since data packets and barrier waves emanate from the open region, and none can enter the open region, the open region has entry pickets on all input queues around its perimeter.
2. All processors outside the open region may not insert packets or barrier markers into this VIN. Hence all input queues from processors outside the open region are marked as being on an entry picket.
3. Barrier waves cannot arrive from the direction opposite the open region. Hence:
   - A. Switches above the region have entry pickets on the input queues from the Y+ direction.
B. Switches below the region have entry pickets on the input queues from the Y- direction.
C. Switches to the right of the open region have entry pickets on the input queues from the X+ direction.
D. Switches to the left of the open region have entry pickets on the input queues from the X- direction.

(4) All packets and barrier waves leaving the open region route first in the Y direction. This has two implications.

A. Hence once a packet or wave has propagated so that it is outside of the X range of the region, it may no longer move in either of the Y directions. Hence in nodes to the right or left of the region, which have x coordinates outside those of the open region, the input queues in both Y directions are on entry pickets.
B. The x side boundaries of the open region must be extended to the top and bottom of the network. In the regions above and below the open region, which have x coordinates the same as the region but Y coordinates outside the regions range, barrier waves will not be arriving from outside the region, hence entry pickets are required.
Chapter 4: Proposed Current and Future Work

The conceptual work for using pickets to implement partial barrier has been essentially completed. In addition, a simulator capable of creating these regions in networks which behave according either Stamatopolous’s or McShane’s integrated network barriers has been created as is producing experimental results.

The work remaining to be completed in the short term includes: (1) Creating larger data sets confirming the correctness and performance of pickets and regions, and (2) creating and defending a final thesis document.

The work to be completed in the long term includes: (1) Attempt to publish these results in a journal or conference; (2) create and prove stronger theorems about the bounds on barrier latency; (3) Attempt to determine if the field of Topological Graph Theory has any tools useful in the study of route embedding and decomposing VINs into plies.
Working Bibliography


