Routing Algorithms on Static Interconnection Networks: A Classification Scheme

Erik Demaine and **Sampalli Srinivas**

Division of Computing Science
Department of Mathematics, Statistics, and Computing Science
Dalhousie University
Halifax, Nova Scotia
Canada B3H 3J5

edemaine@ug.cs.dal.ca **srini@cs.dal.ca**

Abstract

One of the important factors that governs the performance of a parallel computer system is the algorithm that determines the routing of messages in the interconnection network. There have been a number of routing algorithms proposed in the literature for various interconnection networks. The objective of this paper is to present a scheme for classification of routing algorithms on direct networks. The classification scheme is based on five main categories: hardware specification, destination type, worst-case characteristics, fault-tolerance, and performance. To illustrate its use, we have applied it to several previously proposed routing algorithms.

It is shown that the scheme serves as a powerful tool in comparing the performance features of routing algorithms for direct networks. The taxonomy summarizes routing techniques, showing what features it lacks, and where extension would be worthwhile. For example, we observe (by classifying it) that Algorithm A1 is an effective hypercube routing algorithm; however, its large header size causes it to perform poorly. This leads to an improvement that drastically reduces the header size. Furthermore, we see how one can combine routing algorithms to create general hybrid strategies. We thus hope that our classification scheme will facilitate research in the area of routing algorithm development.

1 Introduction

The implementation of many parallel and distributed algorithms on multiprocessor systems requires intensive communications between processors [1,2,3,4]. Consequently, one of the dominating factors that governs the performance of a parallel computer system is the underlying interconnection network and the associated algorithms that determine the routing of messages from one processor to another. *Static interconnection networks*, also known as *direct networks* [1], are a popular class of interconnection networks which are characterized by static links (or *channels*) conforming to a specific topology. Direct networks have been employed in many commercial parallel machines such as the Cray T3E, CM-5, Tera supercomputer, nCUBE, and Intel Paragon, iPSC and iWarp [1,5].

The design of routing algorithms for static interconnection networks has been the focus of much research attention in recent years. In order to fully realize the performance benefits of a parallel application, the routing algorithm must not only provide high throughput and low latency (so as to provide high message-passing performance), but must also be fault tolerant and deadlock, livelock and starvation free (so that in all cases messages are not infinitely delayed). There have been a plethora of routing algorithms that have been proposed for various topologies of direct networks, ranging from simple non-adaptive ones to highly-adaptive routing schemes that provide excellent fault tolerance [6,7]. Unfortunately, no serious attempt has been made to classify routing techniques. This makes it difficult to compare the various algorithms in existence.

The objective of this paper is to present a comprehensive scheme for the classification of routing algorithms on static interconnection networks. The classification scheme is based on five main categories: hardware specification, destination type, worst-case characteristics, fault-tolerance, and performance.

The *hardware specification* describes the parallel machine that the routing algorithm was designed for. This includes network topology and switching technique. The *network topology* refers to the geometric pattern in which the processors are interconnected. The *switching technique* refers to the mechanism by which messages travel along intermediate nodes and links. *Packet switching*, *virtual cut-through*, *circuit switching* and *wormhole routing* are the popular forms of switching methods.

The *destination type* specifies how many processors are involved in a single communication. We will only consider *unicast* routing algorithms, where exactly two nodes communicate at a time. Other possibilities include *multicast* (one-to-many) and *broadcast* (one-to-all) operations. Although we do not cover such algorithms, the classification scheme is general enough to allow them.

The *worst-case characteristics* specify how high congestion in the network can affect the routing algorithm. It includes deadlock, livelock, and starvation freedom. Deadlock, livelock, and starvation are situations which keep messages from reaching their destination, and hence must be avoided. *Deadlock* occurs when messages wait indefinitely due to interdependencies. A message *livelocks* if it is continually misrouted due to faults or busyness in the network. *Starvation* is when a node cannot inject new messages into the network because of the overload of messages being received.

The issue of *fault tolerance* is important, since we do not want parallel programs to grind to a halt whenever there is a hardware error. It is measured using three parameters: fault knowledge, fault tolerance guarantee, and fault tolerance probability. *Fault knowledge* specifies whether only adjacent processors need to know about faults, or if the fault status must be known globally. The *fault tolerance guarantee* is the number of faults that the routing algorithm can guarantee that it can handle. Once this value has been exceeded, the *fault tolerance probability* specifies how likely it is that a message can still be routed.

The final category is *performance*, which gives an indication of the efficiency of the routings. It includes three parameters: minimality of paths, header size, and number of virtual channels. The *minimality of paths* specifies how long the routes are compared to an optimal path. The *header size* details the extra number of bits that must be added to each message. *Virtual channels* are a method of avoiding deadlock that offer significant overhead.

We classify several algorithms that have been proposed in the literature. We see how this classification can be used to extend individual algorithms by identifying bottlenecks. Further, we see how two routing algorithms can be combined to make a general, powerful hybrid-scheme.

The rest of the paper is organized as follows. Section 2 describes the proposed taxonomy for routing algorithms. In Section 3, we apply the classification scheme to several routing algorithms, and show how the scheme helps to find weak points in routing algorithms. Furthermore, we discuss how one can combine routing algorithms. We draw concluding remarks in Section 4.

2 Proposed Classification Scheme

In this section we discuss issues involved with routing algorithm design. Each issue constitutes a portion of the proposed classification scheme.

2.1 Routing Algorithms

As mentioned earlier, this paper deals with static interconnection networks, such as those in Figure 2. An obvious restriction is that nodes in the network are not directly connected by a channel; otherwise, $O(n^2)$ links are needed, and each processor must have $n-1$ ports. This yields an unscalable network and is completely impractical.

Instead, messages must travel over several nodes and links to reach their destination. A *routing algorithm* decides the way that messages traverse the interconnection network. Routing algorithms cannot be implemented in such a global manner, with a "command central" examining the entire network and controlling every message, since this yields too much overhead. Instead, the decisions must be distributed throughout the parallel computer. Typically an intermediate processor is interrupted when a message needs further instruction.

One characterizing feature of routing algorithms is the type of destination node that they allow. In *unicast* methods, a message is sent from one processor to another. *Multicast*, the most general destination type, allows the same message to be sent a group of processors. *Broadcasting* is a special case wherein the group consists of every processor in the network (except the sender). This usually gives the opportunity for more efficient algorithms. We will focus on unicast routing algorithms in this paper; however, we leave **destination type** as a category of the classification scheme so that this is not a limitation.

An obvious issue in routing is the time to send messages. Formally, *latency* is the length of the interval from when the message was injected into the network to when the message reaches its destination [2,8]. Take for example a 2×2 mesh as illustrated in Figure 1(a). Suppose that in a single clock cycle node $(1,1)$ wants to send a message to $(2,2)$ and vice-versa.

If both chose the same node to go to, say $(2,1)$, then in the next clock cycle node $(2,1)$ could only receive one of the messages (suppose $(2,2)$'s), causing $(1,1)$ to wait (Figure 1(b–c)). In fact, we will see that if node $(1,1)$ does not have enough buffers, this situation will cause deadlock. On the other hand, if one message is sent to node $(2,1)$ and the other is sent to $(1,2)$, then each would reach their destination in two (instead of three) clock cycles (Figure 1(d–e)).

It is thus important to consider traffic congestion and other latency topics when designing a routing algorithm, to reduce communication time and further congestion. However, it is often

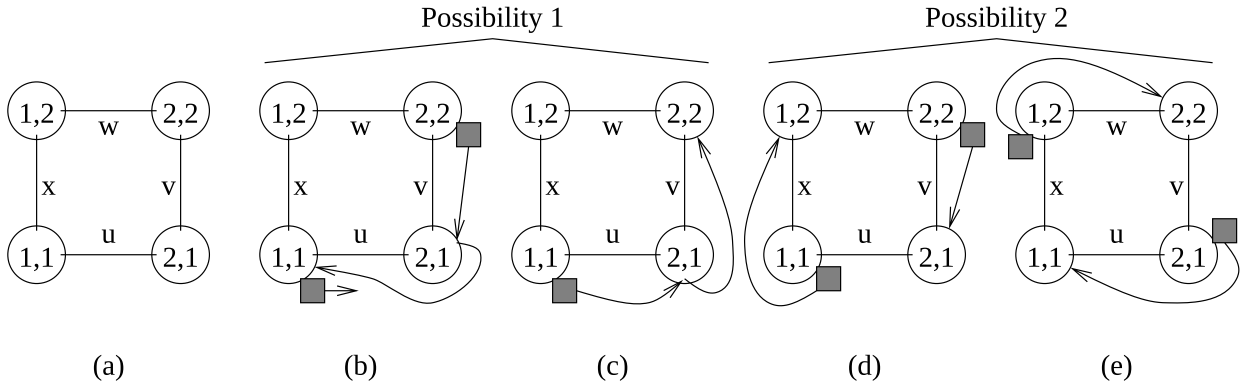

Figure 1: *An illustration of how routing decisions can affect latency. (a) A 2×2 mesh. (b–c) Case where both messages attempt to use node (2,1). (d–e) Case where messages choose different nodes.*

difficult to measure latency directly, since it depends greatly on the network size and topology, as well as the pattern in which processors communicate. For this reason, we do not include it in our classification scheme; instead, we will use other information to determine performance (see Section 2.5).

2.2 Hardware Specification

Routing algorithms are usually designed for a particular type of parallel computer. Thus, routing techniques often specify certain characteristics of the underlying hardware, which we call the *hardware specification*, and exploit these assumptions to provide efficient routing.

2.2.1 Network Topology

The most common specification is the **network topology**. For instance, popular topologies used for static interconnection networks include the k-ary n-cubes such as hypercubes, tori and rings, as well as meshes and trees (Figure 2). There are many network topologies which have been proposed in the literature. Most routing algorithms are only designed for a particular class of topologies, such as hypercubes or meshes.

In some cases, the topology of the parallel computer's network need not match that of the routing algorithm. For example, hypercubes can *embed* a mesh and therefore mesh-based algorithms can be employed on hypercube networks by ignoring some links.

Another example is the CCC (cube-connected-cycle) network, which replaces each node in a k-dimensional hypercube with a ring of k nodes; by enforcing a constant node degree it yields an extremely scalable network. One may modify a hypercube routing algorithm for a CCC network by considering each ring as a "pseudo-node," and providing a means to get from one end of a pseudo-node to another (by traversing the ring). Such a method would likely involve virtual channels, discussed in Section 2.5.3.

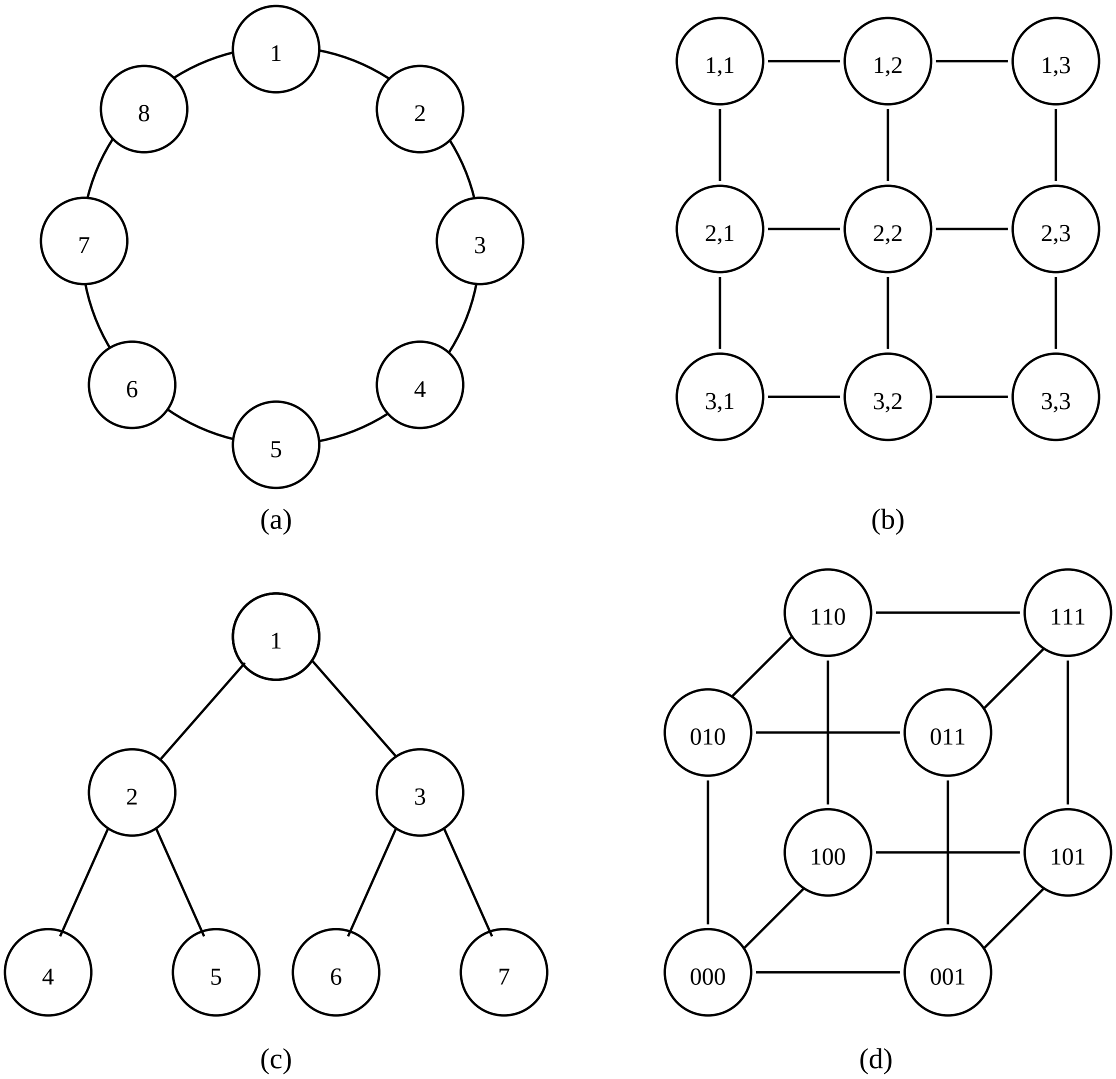

Figure 2: *Examples of static interconnection networks. (a) Ring with 8 nodes. (b) 3 × 3 mesh. (c) Tree of depth 3. (d) 3-D hypercube.*

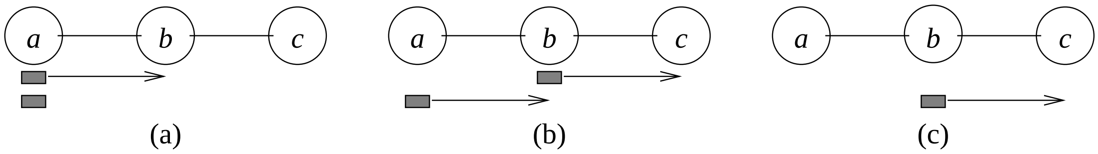

Figure 3: *Illustration of routing a two-flit message through two links.*

2.2.2 Switching Technique

The **switching technique** [2,3,9] refers to the mechanism by which messages travel through intermediate nodes and links. There are four main switching methods addressed in this section: packet switching, virtual cut-through, circuit switching, and wormhole routing.

Packet switching (or *store-and-forward routing*) is the simplest switching method. To send a message from one processor to another under packet switching, the current processor sends the whole message to the next processor. All messages (or *packets*) are of fixed size. The main disadvantage of packet switching is that the entirety of the message received before it can be sent to the next node.

Kermani and Kleinrock presented *virtual cut-through* [3,9] as an improvement over packet switching. In this type of switching, the *switcher* transfers the message to an intermediate processor only when the next link is busy or the destination is reached, in which case it is buffered. Otherwise, the switcher will send the message directly to the next intermediate node. Note that the switcher is intelligent enough to employ the routing algorithm itself. This allows the processor to concentrate on computation and be interrupted less.

Another improvement over packet switching is *circuit switching.* Routing algorithms of this class reserve all needed links (waiting if they are not available), set up a physical circuit using extra hardware, and then send the message directly to the destination. No intermediate buffers are needed, thereby reducing latency induced by the distance of the source and destination nodes. After transmission, the circuit is "torn down" and unreserved. Computers such as the Intel iPSC-2 and iPSC/860 use circuit switching [3].

In wormhole routing, a message is divided into a series of fixed-size blocks called *flits* (flow-control units). The first flit contains necessary header information, and the remaining ones contain the message's content. The flits travel along the same route in a pipeline manner; once a startup time is paid, a flit will arrive every clock cycle. Hence, it effectively removes the latency's dependence on distance.

Packet switching and wormhole routing are the more popular switching techniques. To illustrate the efficiency of the latter, let us consider an example where the network is a three-node linear array (Figure 3). Suppose that a two-flit message from node a wishes to be delivered to c. If packet switching is used, two clock cycles are required for the message to be forwarded to b, followed by another two cycles for the message to reach the destination. On the other hand, if wormhole routing is used, less time is taken. The first flit will require one clock to reach b. While it then traverses to c, the second flit will simultaneously travel to b. The final clock is used by the second flit as it traverses to the destination. Thus, three clock ticks are required by wormhole routing, instead of the four used by packet switching. Immense performance benefits are achieved when the

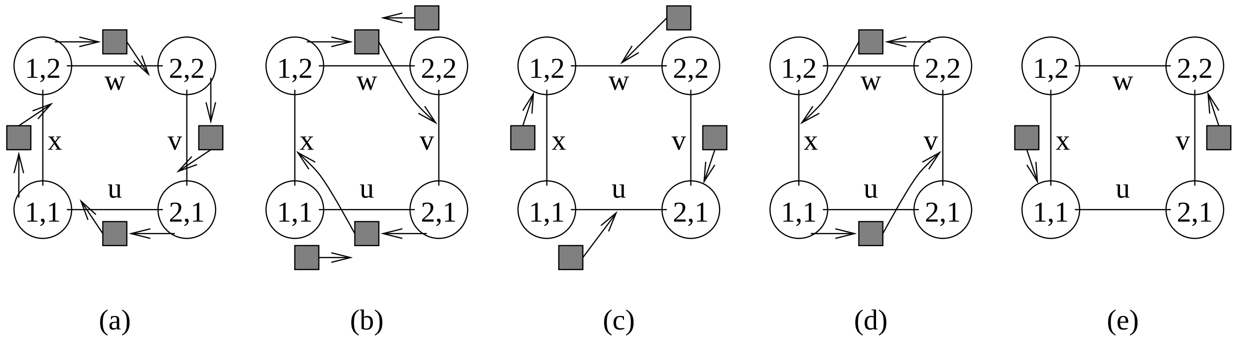

Figure 4: *Illustration of deadlock. (a) Clockwise routing yields deadlock. (b–e) e-cube routing avoids deadlock.*

source and destination are further apart and more flits are used. It is common for a flit to be as small as one computer word, and hence routing algorithms should be designed with little header information.

One should keep in mind that the switching technique is enforced by the hardware. Routing algorithms are often designed for a particular switching method. Although it is often possible to use an algorithm on a parallel computer that uses a different switching technique, this usually does not come without problems. For example, wormhole routing is very susceptible to deadlock because of the trails (or "wormholes") that a single message leaves behind, whereas it is much easier to avoid deadlock in packet-switched algorithms. We will discuss this further in the next section.

2.3 Worst-Case Characteristics: Deadlock

An important issue in routing is *deadlock*. Deadlock occurs when the routing algorithm can never deliver a message due to interdependencies in the network [2,3,4,10,11]. This can be caused by faulty nodes/links and activity in the network. It is critical that it be avoided, since it causes parts of the network to be indefinitely in use, making it more difficult for other messages to reach their destinations.

Let us again consider the 2×2 mesh. Suppose that in one clock cycle all processors generate a message sending to the opposite corner. Assume that messages wait in links. If the routing algorithm decides to route all messages in a clock-wise direction, the system comes to a halt (Figure 4(a)). In contrast, if the algorithm attempts to correct the X direction (if needed), followed by an optional Y direction fix, then no deadlock occurs (Figure 4(b–e)). This method, called e-cube routing, can in fact be generalized to arbitrary hypercubes; it is discussed in Section 3.2.1.

There are a variety of ways to overcome deadlock. One possibility is to increase the number of buffers at the nodes [2]. Proving "deadlock-freedom" then becomes determining the maximum number of buffers required. Unfortunately, this technique is not always applicable. For example, wormhole-routing based algorithms use the channels as a critical resource instead of the buffers at the nodes. In this case a more detailed analysis is required; a common approach is to use *channel-dependency graphs* [8,12]. Thus, the term **deadlock freedom** has different meanings depending on the switching technique of routing algorithm; in the case of packet-switched algorithms, it must be qualified to specify the number of queues and buffers.

2.4 Worst-Case Characteristics: Livelock and Starvation

Even in deadlock-prone routing algorithms, deadlock is uncommon. This is because several messages must be simultaneously injected, and happen to involve a communication pattern that forms cycles. If parallel programs did not pass messages often, or the network was extremely fast, the network would only ever contain a single message, thereby avoiding deadlock.

Unfortunately this is never the case. The *worst case*, as one may call it, is not difficult to generate with basic parallel applications. This is why the *worst-case characteristics* category is important. Congestion should never cause the network to come to a halt (although it may increase latency), since it would make parallel computers unpredictable.

Deadlock is not the only consideration for the worst case. Two other problems may occur when the network is very busy: livelock and starvation.

Livelock [2,4] occurs when a message is misrouted indefinitely, and the message never reaches the destination. Here "misrouted" means that the message is sent to a node that is no closer to the destination, likely because the routing algorithm could not find a better path (due to significant busyness or faults). It is clear that algorithms that employ misrouting must be careful not to do so to the point that the message will never arrive; at worst, it must report that it cannot find a route to the destination.

A node is said to **starve** [2] if it can never inject a message into the network because the buffers are continually in use by other traveling messages. This obviously cannot occur if the *injection model* is *static* [2], that is, all processors inject their messages at the same time, with no other messages present in the network. Common only to SIMD computers, this model is extremely restricting and so will not be considered. When injection is dynamic, injection must (when possible) take higher priority over traveling messages. This can usually be achieved by proper queue handling or protocols, and so is often not of concern to the routing algorithm.

2.5 Performance

As mentioned in Section 2.1, we cannot measure the performance of a routing algorithm by simply looking at its latency. Instead, we will examine three common forms of overhead induced by the routing algorithm: minimality of paths, header size, and virtual channels.

2.5.1 Minimality of Paths

An obvious parameter that affects latency is the length of the path chosen by the routing algorithm. Again, this cannot be measured exactly; instead, we rate the **minimality of paths** generated by a routing algorithm as one of three types. An algorithm is *fully minimal* when only most direct routes are taken to send messages, that is, ones with minimum length. A routing technique is *partially minimal* if it allows a longer path, but with limited or no backtracking. In a *non-minimal* method, backtracking is routinely employed to make a full search of the network.

Partially minimal routing algorithms are most common, since they have more flexibility and so make it easier to provide other important features such as fault tolerance. When a switching technique such as wormhole routing is used, the extra distance involved usually has negligible affect on latency. On the other hand, one must be careful with partially minimal paths; since misrouting is involved, there is a potential for livelock (this is also the case for non-minimal paths).

One problem with non-minimal algorithms is that they often induce a large header size (see below) to ensure that no link is traversed more than once.

2.5.2 Header Size

In the front of every message, a *header* must be included, providing information about the message such as the destination node. In wormhole routing, this is stored in the first flit; since we want flits to be as small as possible, the header should also be of minimal size. With packet-switching, every packet (of which there may be many for a single (large) message) has a header. In any case, a large header can greatly increase the latency of message passing. Hence, we would like to minimize header information whenever possible.

The length of this header is called the **header size**. Header size (in bits) is normally written in terms of n, the number of nodes, and specifies the number of bits. Usually we are interested in the asymptotic value, although when implementing routing algorithms it is useful to know any leading coefficients and lower-order terms.

2.5.3 Virtual Channels

Suppose that two messages want to share a channel. What normally happens is that one waits while the first is sent, and then the second is sent. This can increase the maximum latency, and has the potential for deadlock. Thus, an alternative, called *virtual channels* [13], was proposed. In our example, we would send both messages, with each one using half of the channel bandwidth. In general, a message can use one n^{th} of the bandwidth when n messages are sent simultaneously. Typically, these virtual channels are set up ahead of time, so that a virtual channel specifies a "portion" of a physical channel.

Virtual channels are useful to prevent deadlock. When potential deadlock is detected, channels may be shared so that deadlock is avoided. Dynamically creating virtual channels, however, is often costly, and so is not used. One common approach is to combine different routing algorithms on the various channels, so that when one fails, a message can switch to another set of virtual channels, thereby using a different algorithm.

An unfortunate drawback is the overhead of keeping track of the virtual channels. This grows as the busyness of the network increases. Virtual channels also degrade overall performance. Hence, measuring the **number of virtual channels** is important. On the other hand, virtual channels are very useful as a solution to deadlock.

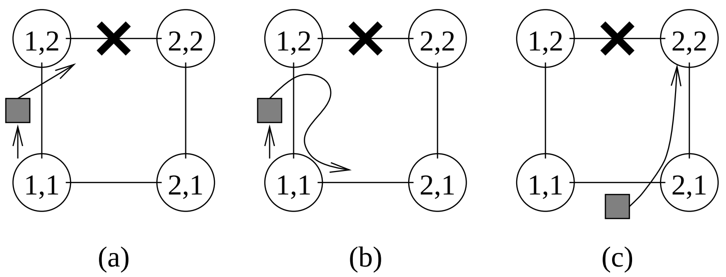

Figure 5: *Routing in a faulty* 2×2 *mesh. If the message decides to go to node* $(1,2)$ *in order to get to node* $(2,2)$*, it must turn back.*

2.6 Fault Tolerance

An important factor of routing algorithms is *fault tolerance* [14,15,16]. In a typical massively parallel supercomputer which is in constant use, links and/or nodes may malfunction and go faulty. There are several techniques to overcome faults in a static interconnection network. One method is to provide extra links and nodes which may become substitutes for faulty devices [14,15]. Another less popular technique is *degradation* of the topology. This significantly decreases performance, and requires significant user support.

A more common way to tolerate faults is to structure the routing algorithm so that it tolerates a certain number of faulty links and nodes [2,5,10]. This is done by making messages avoid faults, assuming a healthy path exists.

As an example, let us examine a 2×2 mesh with one link faulty (Figure 5). If a message wants to be sent from node $(1,1)$ to $(2,2)$, and the routing algorithm in place selects the first node to traverse to as $(1,2)$ (planning to then send to $(2,2)$), then a problem occurs. In order for the routing algorithm to be *fault tolerant of one link*, either this situation should not be allowed to occur, or the algorithm must be intelligent enough to route the message back to $(1,1)$ after it notices the bad link, and take a different route.

Let us note that viewing every link incident to a faulty node as faulty will allow us to disregard faulty nodes. There are two general methods for providing fault tolerance for links. The first (most common) method is to view a faulty link as permanently busy. Assuming the algorithm has good support for busy links, it will then also perform well in the case of faulty links. The second method is only applicable to routing algorithms that do not require a specific topology: when a faulty link is detected, remove it from the "working topology" so that future routing will not decide to use it. Since most routing algorithms operate on a single topology, this method is not very attractive.

Another issue is what faults are known by which nodes. Faults are known *locally* if a link fault is known only by the nodes incident to that link. This method is the most desirable, although it may be too inflexible, compared to *global* fault information which issues broadcasting overhead.

Other than **fault knowledge** described above, we measure fault tolerance of routing algorithms using two parameters. The **fault guarantee** is the number of faults that may exist in the network where the algorithm is guaranteed to find a path to the destination node (assuming one exists). Such a property is usually established via a mathematical proof. The **fault tolerance probability** specifies the chance on average that a message can be routed in a network with more faults than the guarantee. This value can only be measured approximately and is often evaluated using a simulation.

One obvious requirement for any fault tolerance is adaptivity. A routing algorithm is *adaptive* if the route that it creates depends on the current state of the network, including busyness and faults. Unfortunately, adaptivity makes it more difficult to provide deadlock and livelock freedom.

3 Example and Application

As an illustration of the applicability of the proposed scheme, we give a table classifying several routing algorithms that have been proposed in the literature (Table 1). The table serves as a fast and easy way to compare these algorithms.

We will now observe how the table can be used to extend and combine routing algorithms. Thus, we hope that our classification scheme will serve to facilitate new research in the routing algorithm area.

3.1 Improving an Individual Algorithm

First we will examine the entry of a single algorithm, Algorithm A1 [3]. Table 1 allows us to determine the weak points of this algorithm in a simple manner. Using this information, we will examine how these bottlenecks can be alleviated.

To begin, we must first describe Algorithm A1 in more detail.

3.1.1 Algorithm A1

Chen and Shin proposed an adaptive, partially-minimal routing algorithm for hypercubes, called *A1* [3]. As seen in Table 1, a significant advantage is its fault guarantee of $d-1$ links for an d-dimensional hypercube. Since a node in such a hypercube has degree d, this is optimal if we do not make assumptions about connectivity of the network. In addition, when this guarantee is exceeded, the fault tolerance probably reports that the algorithm has a high chance of successfully routing messages.

Unfortunately, A1 has a unusually large header size. The header contains the following information:

- $[c_1, \ldots, c_k]$, a list of dimensions that must be traversed to reach the destination. Note that k is the Hamming distance to the destination node. $d \log d$ bits at most.
- m_i $(1 \leq i \leq d)$, a flag determining if dimension i has been previously used for misrouting. d bits.

Algorithm	Topology	Switching Technique	Deadlock Freedom	Livelock Freedom	Fault Guarantee	Fault Probability	Virtual Channels	Minimality of Paths	Starvation Freedom	Header Size
Adaptive Minimal [2]	Hypercube	Packet	Yes	Yes	None	High	1	Yes	Yes	$\log n + 1$
Algorithm A1 [3]	Hypercube	Wormhole	No	No	$\log n - 1$	High	1	Partial	Yes	$n + n \log n$
Chaos Router [2]	Hypercube	Packet	Yes	No	$\log n - 1$	Low	1	Partial	Yes	$\log n$
Depth-First Search [5]	Hypercube	Wormhole	Yes	Yes	Full	—	1	No	Yes	$\log^2 n$
Direction-first e-cube [17]	k-ary n-cube	Wormhole	Yes	Yes	1	High	1	Partial	Yes	$O(\log n)$
Duato's Protocol [8]	Hypercube	Wormhole	Yes	Yes	None	High	$k + 2$	Yes	Yes	$\log n$
e-cube [4]	Hypercube	Wormhole	Yes	Yes	None	Low	1	Yes	Yes	$\log n$
e-cube (KNC variant) [12]	k-ary n-cube	Wormhole	Yes	Yes	None	Low	2	Yes	Yes	$\log n$
Exchange Model [2]	Any	Packet	Yes	Yes	None	Low	1	Partial	Yes	$\log n$
Exhaustive Misrouting Backtracking [3]	Hypercube	Wormhole	Yes	Yes	Full	—	1	No	Yes	$\log n$
Exhaustive Profitable Backtracking [3]	Hypercube	Wormhole	Yes	Yes	Full	—	1	No	Yes	$\log n$
Idle [3]	Hypercube	Wormhole	No	Yes	None	Medium	1	Yes	Yes	$\log n$
Interdimensional [3]	k-ary n-cube	Wormhole	No	Yes	None	High	1	Partial	Yes	$n + \log n + \texttt{ts}$
Intradimensional [3]	k-ary n-cube	Wormhole	No	No	None	High	1	Partial	Yes	$\log n + \texttt{ts}$
Intra/Interdimensional [3]	k-ary n-cube	Wormhole	No	Yes	None	High	1	Partial	Yes	$n + \log n + \texttt{ts}$
k-family [3]	Hypercube	Wormhole	Yes	Yes	$n - k + 1$	High	1	No	Yes	$n + \log n$
Kim and Shin [10]	Hypercube	Wormhole	Yes	Yes	Full	—	1	Partial	Yes	$\log n$
Negative-first [3,4]	Torus	Wormhole	Yes	Yes	1	High	1	Partial	Yes	$\log n + 1$
Random [2]	Any	Packet	Yes*	Yes	Varies	Varies	1	Yes	Varies	Varies
R_{CCC} [12]	CCC	Wormhole	Yes	Yes	None	Low	3	Yes	Yes	$\log n$
Stepwise e-cube [10]	Hypercube	Wormhole	Yes	Yes	2 non-adjacent subcubes	None	1	Yes	Yes	$\log n$
Two-Phase Misrouting Backtracking [3]	Hypercube	Wormhole	Yes	Yes	Full	—	1	No	Yes	$\log n$
West-first [3,4]	Torus	Wormhole	Yes	Yes	1	High	1	Partial	Yes	$\log n + 1$

Table 1: *Examples of classification method. In the table, log is in base 2. Note that where full fault tolerance is provided, the fault probability is irrelevant. Virtual channels are measued per physical channel. The destination type is unicast for all the algorithms. The fault knowledge is local for all routing algorithms except Kim and Shin's. * Only proved for hypercubes.*

The basic algorithm for A1 is simple. At a particular node, a message first attempts to find a non-faulty link equal to a c_i for some i. If such an i does not exist, a non-faulty link j will be taken, where $m_j = 0$. Note that limiting misrouting with m avoids livelock.

3.1.2 Improving A1's Header Size

m has a reasonable size of $d = \log_2 n$. On the other hand, c takes significantly more space. This makes the routing algorithm slower, and so it is worth-while to look for an improvement to shrink the header size.

Fortunately, there is an alternative storage mechanism for c which is similar to that of m. We can simply store $c_1, ..., c_d$, where $c_i = 1$ if and only if dimension i need be traversed to reach the destination. This reduces the header size to $O(\log n)$ total.

3.2 Combining Two Algorithms

Next we will look at a more important application of the proposed classification scheme: combining routing algorithms. It is common that such a hybrid scheme provides more generality, and achieves the advantages of both algorithms.

One may notice that in Table 1 there are few effective algorithms designed for a general k-ary n-cube, a popular static interconnection network which effectively combines the torus and hypercube networks. In this section we will see how hypercube- and torus-based algorithms may be combined to create a powerful routing algorithm for k-ary n-cubes.

3.2.1 e-cube Routing

The simplest deadlock-free wormhole-routing algorithm is *e-cube* [4]. It is a fully minimal, non-adaptive routing algorithm. The header of a message need only store the destination of the node.

The algorithm is as follows: at an intermediate node, the least significant (or equivalently, the most significant) dimension that must be corrected is traversed. The message will wait until the necessary link is available. e-cube is hence extremely simple. However, it does not provide any fault tolerance. When a link on the path is busy, messages are forced to wait. Assuming that no online repairs may be made, a fault constitutes a permanent busyness, causing messages to wait indefinitely.

3.2.2 West-First Routing

West-first routing is an adaptive, partially minimal, deadlock free routing algorithm for a two-dimensional torus and invented by Glass and Ni [3,4]. In this routing technique, all *west* (a consistent direction) routing is done first, followed by other direction routing. Once any direction other than west is traversed, west cannot be done again to avoid deadlock. When the west phase is completed,

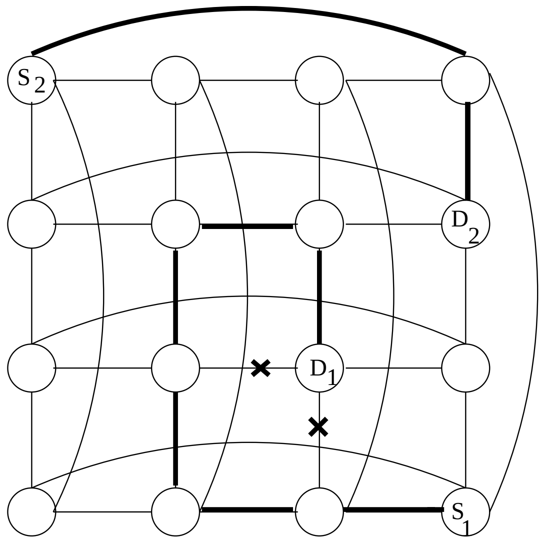

Figure 6: *A* 4×4 *torus with west-first routing.*

the message is adaptively sent, avoiding faults, toward the destination in a close-to-minimal path. The wraparounds in the torus are only used in the first hop of the routing. Sample routings are given in Figure 6.

If no faults exist in the network, all paths are minimal. Note that the direction "west" can be chosen in any way, so long as the decision does not change during execution, extending the algorithm to what we call a *direction-first routing.* The low latency of messages and fault tolerance are the main highlights, providing efficient routing for a torus.

Although west-first routing appears much better than e-cube, it is only designed for tori, which is not very practical in many applications. It is this motivation that led to the proposal of the direction-first e-cube routing algorithm.

3.2.3 Direction-First e-cube

The k-ary n-cube network (Figure 7) is a generalization of the hypercube wherein each 2×2 square is replaced by a $k \times k$ torus. Thus, an n-dimensional hypercube is a binary n-cube. The nodes are numbered in a similar way, but each bit is replaced with a digit d where $0 \leq d < k$.

Few routing algorithms work on a k-ary n-cube for all values of k and n, and those that have accomplished this are generally neither fault tolerant nor deadlock free, or employ costly virtual channels. To solve this problem, we proposed a new adaptive, partially-minimal algorithm for k-ary n-cubes, $k > 1$, which combines direction-first routing and e-cube [17]. The strategy adopted in this algorithm consists of first selecting the torus defined by the two dimensions corresponding to the least significant differing bits between the source and destination nodes. Then the direction-first routing technique is employed to correct these dimensions using the designated torus. This is done by traversing in any particular direction first, and then traversing other directions adaptively. The wraparound links are optionally used in the first hop. The above two steps are repeated until the destination is reached.

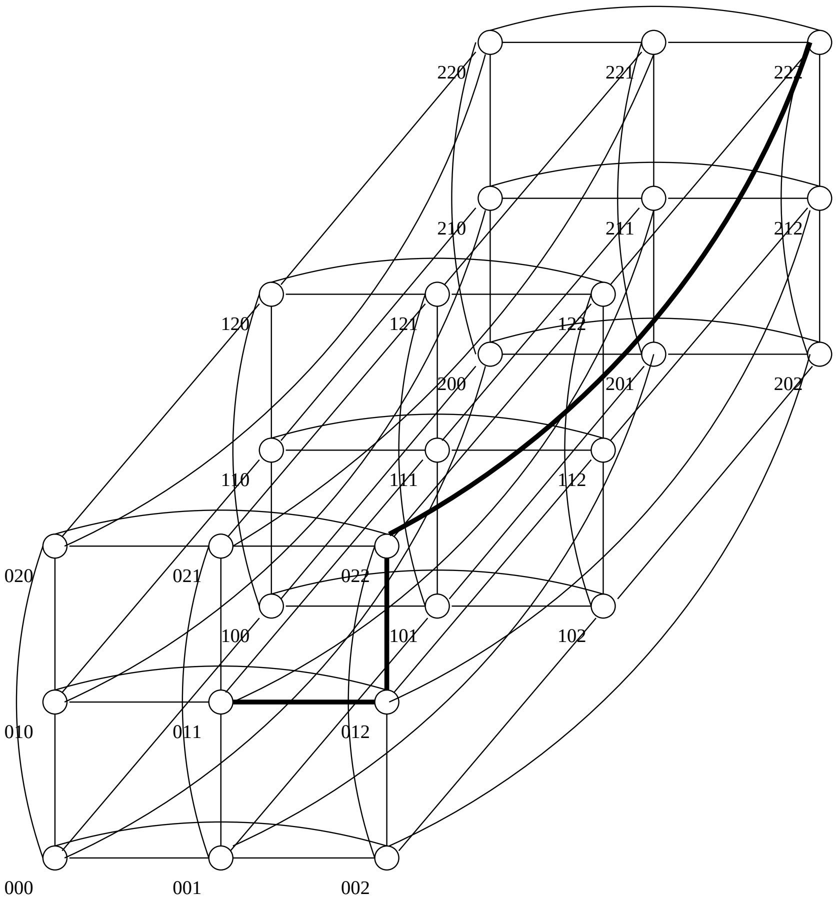

Figure 7: *A ternary 3-cube network with a direction-first e-cube routing from 011 to 222.*

The following is the pseudo-code description of the algorithm.

- *Input*: k-ary n-cube with local channel information; the source (current) node $s_{n-1}...s_0$; the destination node $d_{n-1}...d_0$; the direction; a flag f which equals 1 if and only if the direction has been traversed in the previous dimensions x, y.
- *Output*: The next node to traverse $a_{n-1}...a_0$; an updated f, x, and y. This algorithm should then be applied with $s = a$ on node a.
- *Initial values:* The first time direction-first e-cube is used for a particular message, use $f = x = y = 0$.
- *Algorithm*:
  1. ("double" e-cube phase) Let i, j be dimensions such that
     (a) $s_i \neq d_i$ and $s_j \neq d_j$. (i and j are incorrect dimensions.)
     (b) If $0 \leq l \leq n-1$, $l \neq i$, $l \neq j$, and $s_l \neq d_l$, then $i < j < l$. (i and j are minimal.)

     If no such j exists, choose it to be y if $y \neq i$ or else a random dimension. If no such i exists either, the destination has been reached (stop).
  2. If $i \neq x$ or $j \neq y$, then set $f \leftarrow 0$, $x \leftarrow i$, $y \leftarrow j$.
  3. Employ direction-first routing given $f, (s_i, s_j)$, and (d_i, d_j).
  4. If the direction was used, set $f \leftarrow 1$.

Figure 7 shows a ternary 3-cube with a routing from 011 to 222 using west-first e-cube. The first two differing dimensions are 0 and 1 (the two least significant ones). We take 0 and 1 to be the X and Y axes, respectively, defining a 2-D torus. The west direction is negative on the X, so no west routing need be done. Suppose the algorithm adaptively takes the route 011-012-022 which corrects dimensions 0 and 1 (any such path may be chosen adaptively). Then dimension 2 is the last one to fix. Assume that dimension 1 is chosen as the other (Y) dimension. The message can take a wraparound link in the west direction, since this is the first traversal made for this torus. The destination is then reached by the path 022-222.

Direction-first e-cube is seen to remove most of the disadvantages of west-first and e-cube routing. It is simple, general, has low latency, adaptive, and very fault tolerant. Combinations are often the best type of routing algorithms.

4 Conclusion

In this paper, we have proposed a taxonomy for routing algorithms on static interconnection networks. Many routing algorithms have been proposed in the literature, but no serious attempt has been made to classify them. Such a classification scheme has many advantages. The most obvious application is that it allows one to easily compare various routing algorithms for performance and other characteristics. This makes it simple for a parallel-computing vendor to choose a suitable routing technique depending on the situation (the hardware specification) and other decisions (should fault tolerance be present in the routing algorithm, etc.).

In the second part of this paper we classified existing routing algorithms using the proposed method. We then saw how this classification allowed simple extension and combination of routing algorithms. The table quickly showed header size as a weak point in the A1 algorithm, which was easily corrected, thereby making Algorithm A1 a much better routing technique.

Few algorithms work effectively on a general k-ary n-cube interconnection network. To alleviate this problem we have presented in [17] a new wormhole routing algorithm, called direction-first e-cube, which is based on two other routing methods. The algorithm is simple; many others use virtual channels to avoid deadlock [13]. The adaptiveness of the algorithm will route around many faults, as with west-first routing. Without the presence of faults, the algorithm is fully minimal, and only a few extra hops are required when faults are encountered. Because direction-first e-cube is for a general class of interconnection networks, it works for all subsets such as hypercubes and tori, wherein it will reduce to its basic components (e-cube and west-first routing, respectively). Direction-first e-cube has been shown to provide a good mix of features under the proposed classification scheme.

References

1 **Reed, D A and Grunwald, D C** 'The Performance of Multicomputer Interconnection Networks', *Computer* (June 1987) pp. 63–73

2 **Felperin, S A** *et al.* 'Routing Techniques for Massively Parallel Communication', *Proc. of the IEEE*, Vol 79 No 4 (April 1991) pp. 488–502

3 **Ni, L M and McKinley, P K** 'A Survey of Wormhole Routing Techniques in Direct Networks', *IEEE Comp.*, Vol 26 No 2 (Feb. 1993) pp. 62–79

4 **Gaughan, P T and Yalamanchili, S** 'Adaptive Routing Protocols for Hypercube Interconnection Networks', *Computer* (May 1993) pp. 12–23

5 **Chen, M and Shin, K G** 'Depth-First Search Approach for Fault-Tolerant Routing in Hypercube Multicomputers', *IEEE Trans. on Parallel and Dist. Syst.* Vol 1 No 2 (April 1990) pp. 152–159

6 **Meliksetian, D S and Chen, C Y R** 'Optimal Routing Algorithm and the Diameter of the Cube-Connected Cycles', *IEEE Trans. on Parallel and Dist. Syst.*, Vol 4 No 10 (Oct. 1993) pp. 1172–1178

7 **Preparata and Vuillemin** 'The Cube-Connected Cycles: A Versatile Network for Parallel Computation', *Comm. of the ACM* (May 1981) pp. 300–309

8 **Duato, J** 'A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks', *IEEE Trans. on Parallel and Dist. Syst.*, Vol 4 No 12 (Dec. 1993) pp. 1320–1331

9 **Cypher, R and Gravano, L** 'Adaptive, Deadlock-Free Packet Routing in Torus Networks with Minimal Storage', *1992 Int. Conf. on Parallel Proc.* Vol 3 (1992) pp. 204–211

10 **Kim, J and Shin, K G** 'Deadlock-Free Fault-Tolerant Routing in Injured Hypercubes', *IEEE Trans. on Computers* Vol 42 No 9 (Sept. 1993) pp. 1078–1088

11 **Li, Q** 'Minimum Deadlock-Free Message Routing Restrictions in Binary Hypercubes', *Jl. of Parallel and Dist. Computing* **15** (1992) pp. 153–159.

12 **Dally, W, and Seitz, C** 'Deadlock-Free Message Routing in Multiprocessor Interconnection Networks', *IEEE Trans. on Computers* Vol C-36 No 5 (May 1987) pp. 547–553

13 **Dally, W** 'Virtual-Channel Flow Control', *IEEE Trans. on Parallel and Dist. Syst.* Vol 3 No 2 (March 1992) pp. 194–205

14 **Bruck, J, Cypher, R, and Ho, C** 'Fault-Tolerant Meshes and Hypercubes with Minimal Numbers of Spares', *IEEE Trans. on Computers* Vol 42 No 9 (Sept. 1993), pp. 1089–1104

15 **Bruck, J, Cypher, R, and Ho, C** 'Efficient Fault-Tolerant Mesh and Hypercube Architectures', *22^{nd} Int. Symp. on Fault-Tolerant Computing* (July 1992) pp. 162–169

16 **Alam, M, and Melhem, R** 'Routing in Modular Fault Tolerant Multiprocessor Systems', *22^{nd} Int. Symp. on Fault-Tolerant Computing* (July 1992) pp. 185–193

17 **Demaine, E and Srinivas, S** 'Direction-First e-cube: A New Routing Algorithm for k-ary n-cube Networks', *High Perf. Computing Symp.* (July 1995)

Acknowledgment

This work has been supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada under Grant OGP0138637 and the Research Development Fund for the Sciences, Dalhousie University, under Grant RDFS R1-24-3770.