On the Fault Tolerance of Some Popular Bounded-Degree Networks

Tom Leighton[1]

Mathematics Department, and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
ftl@math.mit.edu

Bruce Maggs

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
bruce@research.nj.nec.com

Ramesh Sitaraman[2]

Dept. of Computer Science
Princeton University
Princeton, NJ 08544
rks@cs.princeton.edu

Research Report CS-TR-385-92
September 1992

Abstract

In this paper, we analyze the ability of several bounded-degree networks that are commonly used for parallel computation to tolerate faults. Among other things, we show that an N-node butterfly containing $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) can emulate a fault-free butterfly of the same size with only constant slowdown. Similar results are proved for the shuffle-exchange graph. Hence, these networks become the first connected bounded-degree networks known to be able to sustain more than a constant number of worst-case faults without suffering more than a constant-factor slowdown in performance. We also show that an N-node butterfly whose nodes fail with some constant probability p can emulate a fault-free version of itself with a slowdown of $2^{O(\log^* N)}$, which is a very slowly increasing function of N. The proofs of these results combine the technique of redundant computation with new algorithms for (packet) routing around faults in hypercubic networks. Techniques for reconfiguring hypercubic networks around faults that do not rely on redundant computation are also presented. These techniques are less tolerant of faults but are more widely applicable since they can be used with other networks such as binary trees and meshes of trees.

[1]Tom Leighton is supported in part by the Defense Advanced Research Projects Agency under Contracts N00014–87–K–825 and N00014-89-J-1988, by the Air Force under Contracts OSR–89–0271, and by NEC Research Institute. [2]Ramesh Sitaraman is supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center - NSF-STC88-09648.

1 Introduction

In this paper, we analyze the effect of faults on the computational power of bounded-degree networks such as the butterfly, shuffle-exchange network, and mesh of trees. The main objective of our work is to devise methods for circumventing faults in these networks using as little overhead as possible, and to prove lower bounds on the effectiveness of optimal methods. We consider both worst-case and random fault patterns, and we always assume that faulty components are totally disabled (e.g., a faulty node cannot be used to transport a packet of data through the network). We also assume that the faults in the network are static and detectable and that information concerning the location of faults can be used when reconfiguring the network to circumvent the faults. For simplicity, we restrict our attention to node failures since an edge fault can always be simulated by disabling the node at either end of the edge.

There are several ways to measure the effect of faults on a network. In this paper, we will be primarily concerned with the amount by which a collection of faults can slow down some computation in the network. For example, if a butterfly network is being used for packet routing, we will be concerned with how much longer it takes a faulty butterfly to deliver all of the packets than it takes a fault-free butterfly to perform the same task. More generally, we will be interested in the length of time it takes an impaired network to emulate a single step of a fault-free network of the same size and type. In particular, we define the *slowdown* caused by a set of faults in a network G to be the minimum value of S such that any computation that takes T steps on G when there are no faults can be performed in at most $S \cdot T$ steps on G when faults are present. One of our main goals will be to understand the relationship between slowdown and the number of faults for commonly-used networks. In particular, we will prove bounds on the number of faults that can be tolerated without losing more than a constant factor in speed.

We will use two methods for emulating a fault-free network G on an isomorphic faulty network H. The first approach is to embed G into H so that the nodes of G are mapped to non-faulty nodes of H, and so that the edges of G are mapped to non-faulty paths in H. The goal is to find an embedding with minimum load, congestion, and dilation. The *load* of an embedding is the maximum number of nodes of G that are mapped to any single node of H. The *congestion* of an embedding is the maximum number of paths that pass through any edge e of H. The *dilation* of an embedding is the length of the longest path. The load, congestion, and dilation of the embedding determine the time required to emulate each step of G on H. In particular, Leighton, Maggs, and Rao have shown [LMR88] that if there is an embedding of G in H with congestion c, load l, and dilation d, then H can emulate any computation on G with slowdown $O(c + l + d)$.

In this paper, we will be most interested in embeddings for which the congestion, load, and dilation are all constant (independent of the size of the network). In particular, we will show in Section 2 how to embed a fault-free N-input butterfly into an N-input butterfly containing $\log^{O(1)} N$ worst-case faults using constant congestion, load, and dilation. (In other words, we will show how to reconfigure an N-input butterfly around $\log^{O(1)} N$ worst-case faults so that the resulting degradation in performance is at most an $O(1)$ factor in speed.) A similar result will also be proved for an N-node mesh of trees. Hence, these networks can tolerate $\log^{O(1)} N$ worst-case faults with constant slowdown.

Previously, no connected bounded-degree networks were known to be able to tolerate more than a constant number of worst-case faults without suffering more than a constant-factor loss in performance. Indeed, it was only known that

1. any embedding of an N-node (2 or 3-dimensional) array into an array of the same size containing more than a constant number of worst-case faults must have more than constant load or dilation [GE84, KKL+90], and

2. the N-node hypercube can be reconfigured around $\log^{O(1)} N$ worst-case faults with constant load, congestion, and dilation [AL91].

The embeddings that we use in Section 2 are level-preserving, i.e., nodes in a particular level of the fault-free network are mapped to nodes on the same level of the faulty network. We take a significant step towards proving the limitation of embedding techniques for the emulation of these networks by showing that no level-preserving embedding strategy with constant load, congestion, and dilation can tolerate more than $\log^{O(1)} N$ worst-case faults. Whether or not there is a natural low-degree N-node network (the hypercube included) that can be reconfigured using (not necessarily level-preserving) embedding techniques around more than $\log^{O(1)} N$ faults with constant congestion, load, and dilation remains an interesting open question.

In Section 3, we shift our attention to the routing capabilities of hypercubic networks containing faults. First we prove in Section 3.1 that an

N-input butterfly with f worst-case faults can support an $O(\log N)$-step randomized packet routing algorithm for the nodes in $N - O(f)$ rows of the butterfly. The ability of the butterfly to withstand faults in this context is important because butterflies are often used solely for their routing abilities. Previously, it was known that expander-based multibutterfly networks can tolerate large numbers of worst-case faults without losing their routing powers [ALM90, LM92], but no such results were known for butterflies or other hypercubic networks. A corollary of this result is that an N-input butterfly with αN worst-case faults (for some small constant α) can support an $O(\log N)$-step randomized routing algorithm for a majority of its nodes. Note that the number of faults is optimal to within a constant factor, since it is possible to partiton an N-input butterfly into components of size $O(\sqrt{N} \log N)$ with N faults. In Section 3.2 we show that butterflies with faults can also be used for circuit switching. In particular, we show that even if an N-input $O(1)$-dilated Beneš network contains $N^{1-\epsilon}$ worst-case faults (for any $\epsilon > 0$), there is still a set of $N - o(N)$ inputs I and a set of $N - o(N)$ outputs O such that for any one-to-one map $\pi : I \mapsto O$ it is possible to route edge-disjoint paths from i to $\pi(i)$ for all $i \in I$. This result substantially improves upon previous algorithms for fault-tolerant circuit switching in Beneš networks [OT71, SR80] which dealt with a constant number of faults by adding an extra stage to the network.

In Section 4, we use the fault-tolerant routing algorithm from Section 3.2 to show that an N-input butterfly with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$) can emulate a fault-free butterfly of the same size with only constant slowdown. A similar result is proved for the shuffle-exchange network. These results are stronger than the reconfiguration results proved in Section 2 because the number of faults tolerated is much larger. The approach used in Section 4 differs from the embedding-based approaches in Section 2 in that a single node of the fault-free butterfly will be emulated by (possibly) several nodes in the faulty butterfly. Allowing redundant computation provides greater flexibility when embedding one network in another (thereby attaining greater fault tolerance) but also adds the complication of ensuring that replicated computations stay consistent (and accurate) over time. This technique was previously used in the context of (fault-free) work-preserving emulations of one network by another [KLM$^+$89, Sch90].

The techniques developed in Section 4 also have applications for hypercubes. For example, we can use the techniques to show than an N-node hypercube with $N^{1-\epsilon}$ worst-case faults can emulate any normal algorithm with constant slowdown. (The set of normal algorithms include FFT, bitonic sort, and other important ascend-descend algorithms [Lei92].) Previously, such results were known only for hypercubes containing $\log^{O(1)} N$ faults [AL91, BCS90, BCS92]. Whether or not an N-node hypercube can tolerate more than $\log^{O(1)} N$ faults with constant slowdown for general computations remains an important unresolved question.

In Section 5, we show that even if each node in an N-input butterfly fails independently with probability $p = 1/\log^{(k)} N$, where $\log^{(k)}$ denotes the logarithm function iterated k times, the faulty butterfly can still emulate a fault-free N-input butterfly with slowdown $2^{O(k)}$ with high probability. For $k = O(\log^* N)$ the node failure probability is constant, and the slowdown is $2^{O(\log^* N)}$, which grows very slowly with N. Whether or not this result can be improved and whether or not there is a bounded-degree network that can sustain constant-probability random faults with only constant expected slowdown remain interesting open questions. Until very recently, no results along these lines were known for the butterfly (unless routing is allowed through faulty nodes [Ann89], which simplifies matters substantially). Tamaki [Tam92a] has recently discovered an emulation scheme with slowdown $O((\log\log N)^{8.2})$. He has also introduced a class of bounded-degree networks called cube-connected arrays [Tam92b], and showed that an N-node network in this class with constant-probability random faults can emulate itself with expected slowdown approximately $\log\log N$. (These networks can also tolerate up to $\log^{O(1)} N$ worst-case faults with approximately $\log\log N$ slowdown.)

Several proofs and results have been omitted from this abstract due to space limitations. For a complete version of the paper, see [LMS92].

1.1 Additional previous work

There is a substantial body of literature concerning the fault-tolerance of communication networks. We do not have the space to review all of this literature here, but we would like to cite the papers that are most relevant. In particular, [AL91, Ann89, BCLR92, HLN87, HLN89, KKL+90, LSGH87, Rab89, Tam92b] show how to reconfigure a network around faults so that it can emulate a fault-free network of the same size and type. References [AAB+92, BCH91, DH90, DH91] show how to design a network H that will contain G as a subnetwork even if H contains some faults. Algorithms for routing messages around faults appear in [AS82, ALM90, AB91, HLN89, KKL+90, LM92, Lin92, Lyu90, OT71, Rab89, SR80]. The fault-tolerance of sorting networks is studied in [AU90, LMP91]. Finally,

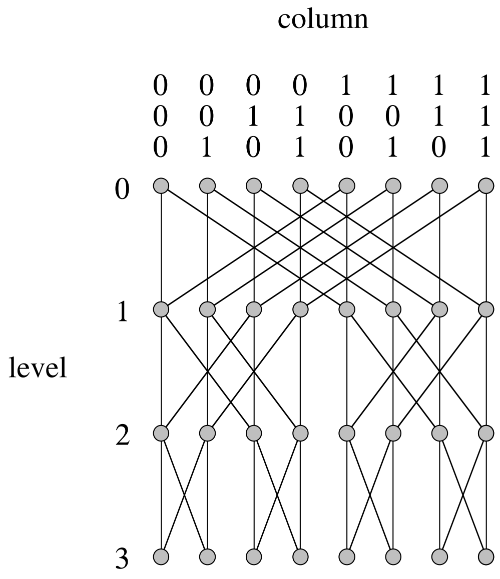

Figure 1: An 8-input butterfly network.

[BCS90, WC92, WCM91] show how to perform certain computations in hypercubes containing faults.

1.2 Network definitions

In this section, we will review the structure of some of the networks that we study in this paper. An N-input ($\log N$-dimensional) *butterfly* has $(\log N + 1)N$ nodes arranged in $\log N + 1$ levels. Each node has a distinct label (w, i), where i is the *level* of the node ($0 \leq i \leq \log N$) and w is a $\log N$-bit binary number that denotes the *row* of the node. All nodes of the form (w, i), $0 \leq i \leq \log N$, are said to belong to row w. Two nodes (w, i) and (w', i') are linked by an edge if and only if $i' = i + 1$ and either w and w' are identical (straight edge) or w and w' differ in precisely bit i' (cross edge). Sometimes the level 0 and $\log N$ nodes in each row are assumed to be the same node. Our results will hold with or without this wrap-around. Another popular network, the hypercube, is related to the butterfly. An N-node *hypercube* is obtained by labeling each node with a distinct $\log N$-bit binary number and connecting with an edge any two nodes whose binary numbers differ in exactly one bit. Another network closely related to the butterfly is the *shuffle-exchange network*. A $\log N$-dimensional shuffle-exchange network has N nodes, each node of which is labeled with a distinct $\log N$-bit binary number. Two nodes labeled u and v are linked by an edge if either u and v differ in precisely the last bit (exchange edge) or if u is a left or right cyclic shift of v (shuffle edge). Another popular interconnection network on which many fast parallel algorithms have been devised [Lei92] is the *meshes of trees network*. The nodes of the meshes of trees network are arranged in the form of an $N \times N$ grid with N rows and N columns. The nodes in each row or column form the leaves of a distinct complete binary tree.

A *circuit-switching network* is used to establish edge-disjoint paths (circuits) between its inputs and outputs. Each switch in a circuit-switching network has two incoming and two outgoing edges. The incoming edges can be connected to the outgoing edges in one of two ways: either crossing or straight through. The switches at the first level of the network are called the *input switches*. The switches at the last level are called the *output switches*. The two edges into each input switch are called *input edges*, or *inputs*. The two edges out of each output switch are called *output edges*, or *outputs*. By setting the switches, each input edge can be connected to an output edge via a path through the network. A circuit-switching network with N inputs and N outputs is said to be *rearrangeable* if for any one-to-one mapping ϕ from the inputs to the outputs, it is possible to construct edge-disjoint paths in the network linking the i^{th} input to the $\phi(i)^{th}$ output, for $1 \leq i \leq N$. A classical example of a rearrangeable network is the Beneš network. The Beneš network consists of back-to-back butterflies. A $\log N$-dimensional Beneš network has $2N$ inputs and $2N$ outputs. Its switches are arranged in $2\log N + 1$ levels of N switches each. The first and last $\log N + 1$ levels each form a $\log N$-dimensional butterfly. Level $\log N$ is shared by these butterflies. We will refer to the nodes (switches) in level 0, $\log N$, and, $2\log N$ as the *input nodes*, *middle nodes*, and *output nodes* respectively.

2 Emulation by Embedding

In this section, we will show how to embed a fault-free binary tree, butterfly, or mesh of trees network into a faulty version of itself with constant load, congestion and dilation. As noted in the introduction, finding a constant load, congestion, and dilation embedding is the simplest way of emulating

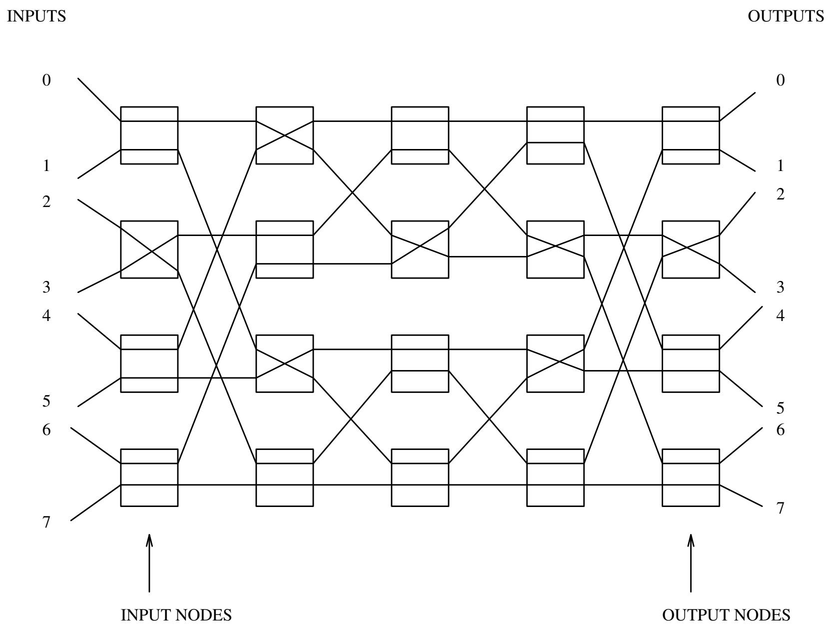

Figure 2: An 8-input (2-dimensional) Beneš network.

arbitrary computations of a fault-free network on a faulty version of itself with only constant slowdown. We will first consider embedding a complete binary tree in a complete binary tree with faults only at its leaves. This result will also hold for fat-trees [GL89] with faults at the leaves. We use this result to find reconfigurations of butterflies and meshes of trees in which faults may occur at any node. The primary result of this section is that an N-node butterfly or mesh of trees network can tolerate $\log^{O(1)} N$ worst-case faults and still emulate itself with only constant slowdown.

2.1 The Binary Tree

Define $S(n,b)$ for $n \geq b \geq 0$ by the recurrence

$$S(n,b) = S(n-1,b) + S(n-1,b-1) + 1$$

for $n > b > 0$ with boundary conditions $S(n,0) = 0$ and $S(n,n) = 2^n - 1$, for $n \geq 1$. The following lemma provides a useful asymptotic bound on the growth of $S(n,b)$ for large n.

Lemma 2.1.1 *For all $n \geq b \geq 1$, $\binom{n}{b} \leq S(n,b) \leq \binom{n+b}{b}$. Hence, $S(n,b) = \Theta(n^b)$ for any constant $b > 0$.*

Proof: The proof is by induction on n and follows trivially from the fact that $\binom{x}{y} = \binom{x-1}{y} + \binom{x-1}{y-1}$, for all $x > y > 0$. $\Box$

Theorem 2.1.2 *Given an N-leaf complete binary tree T with a set of at most $S(\log N, b) = \Theta(\log^b N)$ worst-case faults at the leaves, it is possible to embed a fault-free N-leaf complete binary tree T' in T so that*

1. *nodes on level i of T' are mapped to non-faulty nodes on level i of T, for $0 \leq i \leq \log N$,*
2. *the congestion and the load of the embedding are at most 2^b, and*
3. *the dilation of the embedding is 1.*

Proof: The proof is by induction on $n = \log N$ and b. For $n = 0$, the tree is a single node, and the number of faults, $S(0,0)$, is zero. Now suppose that we are given a complete binary tree T with 2^n leaves of which a set of at most $S(n,b)$ are faulty. If both 2^{n-1}-leaf subtrees of T have at most $S(n-1,b)$ faulty leaves, then we use the result inductively in both subtrees. If one 2^{n-1}-leaf subtree (say the left subtree) has $S(n-1,b)+1$ or more faults, then by definition of $S(n,b)$ the other subtree (the right subtree) has at most $S(n-1,b-1)$ faulty leaves. Hence we can use induction to embed a 2^{n-1}-leaf complete binary tree on the right subtree using dilation 1 and load and congestion 2^{b-1}. By doubling the congestion and the load, we can embed two 2^{n-1}-leaf complete binary trees in the right subtree. This means that we can embed T' in T with dilation 1 and load and congestion 2^b (using only the root and the right subtree of T in this case). $\Box$

This result can be extended to a class of binary-tree-like structures called *fat-trees*. A fat-tree of depth n is specified by a sequence of numbers $m_0, m_1, \cdots, m_n = 1$. (Typically $m_0 \geq m_1 \geq \cdots \geq m_n$.) A fat-tree of depth 0 is a single node, which is both the root node and the leaf node of the tree. A fat-tree of depth n is constructed as follows. At the root of the fat-tree there is a set of m_0 nodes. The left and right subtrees are identical and are constructed recursively. Each is a fat-tree of depth $n-1$ with number sequence $m_1, \cdots, m_n$. The root nodes of the fat-tree are connected to the root nodes of the left subtree with any number of edges in an arbitrary fashion (multiple edges are allowed). An isomorphic set of edges is used to connected the root nodes of the tree to the root nodes of the right subtree.

Corollary 2.1.3 *A* $\log N$*-depth fat-tree can be embeded in a level-preserving fashion in an isomorphic fat-tree with* $S(\log N, b)$ *worst-case faults at its leaves with load and congestion* 2^b *and dilation 1.*

Proof: Associate the nodes of the fat-tree with nodes of an N-node complete binary tree as follows. Associate all root nodes of the fat-tree with the root of the complete binary tree. Recursively associate the nodes of the left (right) subtree of the fat-tree with the nodes of the left (right) subtree of the complete binary tree. Note that every leaf of the fat-tree is associated with a unique leaf of the complete binary tree. Given a fat-tree F with faulty leaves, let T be a complete binary tree whose leaf is faulty iff the corresponding leaf in F is faulty. A fault-free complete binary tree T' can be embedded in T by embedding subtrees of T' into subtrees of T using the procedure given in Theorem 2.1.2. The same embedding can be used to embed a fault free fat-tree, F' in F by embedding the corresponding subtrees of F' into the corresponding subtrees of F. The dilation and load are the same as that of the complete binary tree embedding. □

Corollary 2.1.4 *A* $\log N$*-dimensional butterfly can be embedded in a level-preserving fashion in an isomorphic butterfly with* $S(\log N, b)$ *worst-case faults at level* $\log N$ *with load and congestion* 2^b *and dilation 1.*

Proof: A $\log N$-dimensional butterfly is a fat-tree of depth $\log N$ with $m_i = 2^{\log N - i}$. The leaves of the fat-tree are the nodes in level $\log N$ of the butterfly. □

2.2 The mesh of trees and the butterfly

We can use the previous result to show that the mesh of trees and the butterfly network can tolerate $\log^{O(1)} N$ worst-case faults with constant slowdown, even when a fault can occur at *any node* of the network. The proof uses the fact that both the mesh of trees and the butterfly can be viewed as a special kind of product graph together with Theorem 2.1.2. We will denote the leaves of a tree and the nodes in level $\log N$ of a butterfly as *external* nodes. Given a graph G with $2^n = N$ external nodes, the *external product graph* of G (denoted PG) is defined as follows. Replicate the graph G into $2N$ copies, $G_{i,j}$, for $i = 1, 2$ and $1 \leq j \leq N$. The external nodes of each copy are also numbered from 1 to N. Now identify the k^{th} external node in $G_{1,j}$ with the j^{th} external node in $G_{2,k}$, for all $1 \leq j, k \leq N$. (By "identify" we mean make them the same node of the product graph PG.) The resulting graph is the external product graph of G. We now show that if we can tolerate faults in the external nodes of G, then we can tolerate faults anywhere in PG.

Theorem 2.2.1 *If a graph G' can be embedded in a level-preserving fashion with load l, congestion c, and dilation d in an isomorphic graph G with f worst-case faults located in its external nodes, then it is possible to embed the product graph PG' in a level-preserving fashion with load l^2, congestion c^2, and dilation d in an isomorphic graph PG with $f/2$ worst-case faults located in any of its nodes.*

Proof: Let PG and PG' be made of up of graphs isomorphic to G, $G_{i,j}$ and $G'_{i,j}$, respectively, for $i = 1, 2$ and $1 \leq j \leq N$. Let CG and CG' also be graphs isomorphic to G. The j^{th} external node of CG is declared to be faulty iff either $G_{1,j}$ or $G_{2,j}$ contains a fault. If PG has $f/2$ faults, then CG has at most f faults (since each external node can appear in two copies of $G_{i,j}$). Let Φ be a level-preserving embedding of the fault-free graph CG' into CG with load l, congestion c, and dilation d and define ϕ so that Φ maps the j^{th} external node of CG' to the $\phi(j)^{th}$ external node of CG. We embed PG' into PG by mapping $G'_{i,j}$ to $G_{i,\phi(j)}$ using Φ, for $i = 1, 2$ and $1 \leq j \leq N$. It follows from the definition of faults in CG that $G_{i,\phi(j)}$ is fault-free. Therefore no nodes of PG' are mapped to faulty nodes of PG. We need to verify that every external node of PG' is mapped to a unique node of PG. The k^{th} external node of $G'_{1,j}$ is the same as the j^{th} external node of $G'_{2,k}$. The former is mapped to the $\phi(k)^{th}$ external node of $G_{1,\phi(j)}$ and the latter to the $\phi(j)^{th}$ external node of $G_{2,\phi(k)}$. These nodes are the same node of PG. The dilation of the mapping is d. The number of copies $G'_{i,j}$ of PG' mapped to any particular $G_{i,\phi(j)}$ is at most l. Each copy can put l nodes on to any particular node of $G_{i,\phi(j)}$. Therefore the load is at most l^2 and the congestion is at most c^2. $\square$

Two important networks can be expressed as external product graphs. When the graph G' is a tree the graph PG' is a mesh of trees network. If G' is a butterfly, then PG' is a butterfly with twice the dimension. For simplicity, we state the result for a 2-dimensional mesh of trees. However the same techniques can be used to show that any constant dimensional mesh of trees can tolerate $\log^{\Theta(1)} N$ worst-case faults with only a constant slowdown.

Theorem 2.2.2 *A $2 \log N$-dimensional butterfly can be embedded in a level-preserving fashion in a $2 \log N$-dimensional butterfly containing $S(\log N, b) =$*

$\Theta(\log^b N)$ *worst-case faults with load and congestion* 2^{2b} *and dilation* 1.

Proof: The proof for follows from Corollary 2.1.4 and Theorem 2.2.1. $\square$

Theorem 2.2.3 *An* $N \times N$ *mesh of trees can be embedded in a level-preserving fashion in an* $N \times N$ *mesh of trees containing* $S(\log N, b) = \Theta(\log^b N)$ *worst-case faults with load and congestion* 2^{2b} *and dilation* 1.

Proof: The proof follows from Theorems 2.1.2 and 2.2.1. $\square$

The results of this subsection can also be formulated using the fact that the butterfly and the mesh of trees can be expressed as the Layered Cross Product [EL92] of two complete binary trees (or variations thereof) [Aie92].

2.3 Limitations on level-preserving embeddings

We do not know whether or not Theorems 2.1.2, 2.2.2, and 2.2.3 can be improved if the level-preserving constraint is removed. However, we can show that the bounds in Theorems 2.1.2, 2.2.2, and 2.2.3 are tight if the embedding is forced to be level preserving.

Given an N-leaf binary tree T with faults at its leaves, an arrow diagram has arrows drawn from a some nodes of T to their siblings. We define a b-legal arrow diagram as follows:

1. On any path from the root to a faulty leaf, there is an arrow from a node on the path to a node not on the path (outgoing arrow).
2. On any path with no outgoing arrow, there can be at most b incoming arrows.

Let $T(n, b)$ be the maximum number of faults that can be placed at the leaves of a 2^n-leaf binary tree (where $2^n = N$) without making it impossible to construct a b-legal arrow diagram for the tree. We bound the value of $T(n, b)$ as follows.

Lemma 2.3.1 *For all* $n \geq b \geq 1$, $T(n, b) = O(n^b)$.

Proof: We will first show that $T(n, 0) = 0$. If there are no faults in the tree, then a tree with no arrows is a 0-legal arrow diagram. However, even if the tree has only one fault it necessarily has at least one arrow. If the tree has at least one arrow, then there must be path from the root of the tree to a leaf having at least one incoming arrow and no outgoing arrow.

Such a path can be recursively constructed as follows. Choose the arrow from node m' to its sibling m such that m is the closest node to the root of the tree. The constructed path is the path from the root of the tree to m concatenated with the path constructed recursively in the subtree rooted at m. (If there is no arrow in the subtree rooted at m a path from m to any leaf of the subtree will be sufficient.) Thus, a tree with a fault cannot have a 0-legal arrow diagram. Thus $T(n,0) = 0$. $T(n,n) = 2^n - 1$, since if we allow n incoming arrows without an outgoing arrow we can place n incoming arrows on every node of the path from the non-faulty node to the root. For $n > b > 0$, we show that

$$T(n,b) \leq T(n-1,b) + T(n-1,b-1) + 1.$$

We show this by proving that there is a way of placing $T(n-1,b) + T(n-1,b-1) + 2$ faults at the leaves such that either there must be at least $b+1$ incoming arrows on some path without any outgoing arrows or there must be a faulty leaf with no outgoing arrow in its path. We place $T(n-1,b)+1$ worst-case faults in the left subtree and $T(n-1,b-1)+1$ worst-case faults in the right subtree. Assume that it is possible to place arrows in the tree such that every path to a faulty leaf has an outgoing arrow and every path from the root to a leaf which has no outgoing arrows has at most b incoming arrows. We look at the placement of arrows in the left subtree. Since there are more than $T(n-1,b)$ faults, there must be a path from the root of this subtree to a leaf which has $b+1$ incoming arrows and no outgoing arrows or there must be path from the root of this subtree to a fault with no outgoing arrow. Either of these cases imply that the root of the left subtree must have an arrow from itself to its sibling. Now look at the right subtree. It cannot be the case that there is a path from the root of the right subtree to a faulty leaf with no outgoing arrow, since then there will be no outgoing arrow for the path from this fault to the root of T. Further, no path from the root of the right subtree to a leaf of the right subtree can have more than $b-1$ incoming arrows without having an outgoing arrow, since otherwise there will be a path from the root of the tree to that leaf with more than b incoming arrows without an outgoing arrow. Thus the right subtree must be $(b-1)$-legal. However, the right subtree has more than $T(n-1,b-1)$ worst-case faults. This is a contradiction. The recurrence we have for $T(n,b)$ is similar to what we had for $S(n,b)$ in Section 2.1 and is $O(S(n,b))$. Using Lemma 2.1.1, $T(n,b)$ is $O(n^b)$. $\square$

Theorem 2.3.2 *For any constants, c, l, and d, there is a constant k such that there is a way of placing* $\log^k N$ *faults in the leaves of an N-leaf complete binary tree T such that there is no level-preserving embedding of an N-leaf fault-free complete binary tree T′ in T with congestion c, load l, and dilation d.*

Proof: We are given an $N = 2^n$ node complete binary tree T with faults at its leaves and its fault-free version T'. Let $k = \lceil d + (l-1)2^{d-1} \rceil$. We will choose a worst-case set of faults in T of cardinality $\Theta(\log^k N)$ such that this fault pattern has no k-legal arrow diagram. Clearly this is possible, since $T(n,k)+1$ is $O(\log^k N)$. Suppose, for contradiction, that there is an embedding of T' to T with the property that the nodes in level i of T' are mapped to nodes in level i of T, no nodes of T' are mapped to faulty nodes of T, with dilation (d), congestion (c) and load (l). Annotate the tree T with arrows as follows. For any two siblings in the tree, draw an arrow from the sibling whose subtree has a smaller number of leaves of T' mapped to it to the sibling that has a greater number of leaves mapped to it. If the number of leaves mapped to each of the two subtrees is equal then no arrow is drawn.

We will now show that the annotated tree is b-legal, for some b less than or equal to k. The path from the root of a tree to any faulty leaf must have an outgoing arrow, since no node of T' is mapped to a faulty leaf. Let b be the maximum number of incoming arrows on a path without an outgoing arrow. We will ignore the last d levels of the tree. Therefore, there is a path in T, $m_0, m_1, \cdots, m_{n-d}$, where m_0 is the root and m_i is some node in level i of the tree, which has at least $b-d$ incoming arrows without any outgoing arrows. Let l_i, $0 \leq i \leq n-d$, denote the average number of leaves of T' that are embedded into each leaf of the subtree of T rooted at node m_i. Clearly, l_0 is 1. If there is no incoming arrow into node m_k, then the split of leaves of T' is even and hence $l_k = l_{k-1}$. Suppose there is an incoming arrow into node m_k from its sibling $m_k{}'$, then l_k is greater than l_{k-1}. Further, l_k is at least $l_{k-1} + 2^{-d+1}$. To see why, consider the subtrees of T' rooted at level $k+d-1$. The nodes in each of these subtrees can be mapped entirely within either the subtree rooted at m_k or entirely within the subtree rooted at $m_k{}'$ but never to the nodes in both. The reason is that it is not possible for a node of T' to be mapped to one of the sub-trees (say m_k) to reach a node mapped at an adjacent level in the other subtree (say $m_k{}'$) with dilation d or less. Thus subtree m_k should have at least $2^{n-k-d+1}$ more leaves of T' mapped to it than subtree $m_k{}'$. Thus, since there are at least $b-d$ arrows,

l_{n-d} must be at least $1+(b-d)2^{-d+1}$. Note that there is at least one leaf in the subtree rooted at m_{n-d} which has load at least l_{n-d}. Therefore, l_{n-d} is at most l. This implies that b is at most $d+(l-1)2^{d-1}$, i.e., at most k. But there can be no k-legal arrow placement for the fault pattern chosen for T. This is a contradiction. $\square$

Theorem 2.3.3 *For any constants, c, l and d, there is a constant k such that there is a way of choosing $\Theta(\log^k N)$ faults in an N-input butterfly B such that there is no level-preserving embedding of an N-input butterfly B' in B with congestion c, load l, and dilation d.*

Proof: The proof is similar to that of Theorem 2.3.2. Let B be a butterfly with faults and B' be the fault-free version of B. We can associate a tree T with B as follows: The root of T represents the entire butterfly B. Its children represent the two sub-butterflies of dimension $\log N - 1$ (between levels 1 and $\log N$). Each child is subdivided recursively until each leaf of the tree T represents a unique node in level $\log N$ of the butterfly B. We will choose the same set of worst-case faults in the leaves of T as in Theorem 2.3.2. The faulty nodes of B will be the nodes in level $\log N$ of B which correspond to the faulty leaves of T. Given a level preserving embedding of B' into B with load l, congestion c and dilation d, we can produce a b-legal placement of arrows in T in a manner similar to the previous proof. Given two siblings m and m'. Draw an arrow from m' to m if the there are more nodes in level $\log N$ of B' mapped to the sub-butterfly of B represented by tree node m than the sub-butterfly represented by treenode m'. Let m and m' be on level j of T. As before, due to dilation considerations, it is true that smaller sub-butterflies of B' spanning levels $j+d-1$ to n must be mapped entirely within the sub-butterfly of B represented by m or within the sub-butterfly represented by m' but never to both. The rest of the proof is similar to Theorem 2.3.2. $\square$

Theorem 2.3.4 *For any constants, c, l and d, there is a constant k such that there is a way of choosing $\Theta(\log^k N)$ faults in an $\sqrt{N}$ by $\sqrt{N}$ mesh of trees M such that there is no level-preserving embedding of an $\sqrt{N}$ by $\sqrt{N}$ mesh of trees M' in M with congestion c, load l, and dilation d.*

Proof: The proof is similar to that of Theorem 2.3.3. Let M be a mesh of trees with faults and M' be the fault-free version of M. The nodes in level $\log N$ of M and M' are arranged in the form of a 2-dimensional $\sqrt{N}$ by $\sqrt{N}$

mesh. We will refer to these nodes as the mesh nodes. We can associate a tree T with the mesh nodes of M as follows: The root of T represents the entire mesh. Divide the mesh vertically into two equal parts and let each child represent one of the halves. At the next level of the tree divide each of the halves horizontally into two equal parts. Divide alternately, either horizontally or vertically, until you reach individual mesh nodes which are each represented by a unique leaf of the tree. We will choose the same set of worst-case faults in the leaves of T as in Theorem 2.3.2. The faulty nodes of M will be the mesh nodes of M which correspond to the faulty leaves of T. Given a level preserving embedding of M' into M with load l, congestion c and dilation d, we can produce a b-legal placement of arrows in T in a manner similar to the previous proofs. Given two siblings m and m'. Draw an arrow from m' to m if the there are more mesh nodes of M' mapped to the sub-mesh of M represented by tree node m than the sub-mesh represented by tree node m'. As before, due to dilation considerations, it is true that smaller sub-meshes of M' must be mapped entirely within the sub-mesh of B represented by m or within the sub-mesh represented by m' but never to both. The rest of the proof is similar to Theorem 2.3.2. $\square$

3 Fault-tolerant routing

In this section, we present algorithms for routing around faults in hypercubic networks. Section 3.1 presents algorithms for routing packets in butterfly networks with faulty nodes, while Section 3.2 presents algorithms for establishing edge-disjoint paths between the inputs and outputs of a Beneš network with faulty switches.

3.1 Fault-tolerant packet routing

In this section we show how to route packets in an N-input butterfly network with f worst-case faults. A widely studied class of routing problems is the *permutation routing problem* (see [MS92] for references to permutation routing algorithms). A permutation routing problem between a set of nodes in the network has exactly one packet originating at every node in this set. Each such packet needs to be routed to a destination node in the set such that no two packets share the same destination. We show that there is some set of $N - O(f)$ rows such that it is possible to route any permutation between the nodes in these rows in $O(\log N)$ steps, with high probability. This is comparable to the results for fault-tolerant routing in a multi-butterfly[LM92]. A special case of this result is that when f equals αN (for any $\alpha < 2/9$) we can route arbitrary permutations between a majority of nodes in the butterfly. Note that this is optimal within constant factors since N faults in level $\log N/2$ can bisect the butterfly into many small components.

We start by describing Valiant's algorithm [Val82] for permutation routing in a butterfly without faults. It will be convenient for us to view the packets in this scheme as being routed on a bigger network with $4\log N + 1$ levels. Between level 0 and level $\log N$, and between levels $3\log N$ and $4\log N$, nodes are connected only by straight edges. Between levels $\log N$ and $3\log N$ the network is a pair of back-to-back butterflies isomorphic to the Beneš network. In analogy with the Beneš network the nodes in $\log N$, $2\log N$ and $3\log N$ are called input nodes, middle nodes and output nodes respectively. Note that this bigger network is easily simulated by a single butterfly. In Stage 0, a packet goes down its row to the input node in that row (say m). In Stage 1, the packet goes from m to a random middle node (say m''). In Stage 2, the packet goes from m'' to an output node m' in the row of its destination. In Stage 3, the packet goes from the row of m' to its destination. Valiant showed that these paths, which have length at most $4\log N$, also have congestion $O(\log N)$ with high probability. In leveled networks such as the butterfly, as long as the (leveled) paths of the packets are selected such that the maximum length is $O(\log N)$ and congestion $O(\log N)$, a Ranade-type queuing protocol can be used to route the packets in $O(\log N)$ steps[LMR88]. Therefore it will be sufficient to derive high probabilty bounds on the maximum length and congestion of the paths of a routing scheme.

In a faulty butterfly, we would like to use an algorithm like Valiant's to route packets between the "good" nodes. Let F be the set of worst-case faults and $f = |F|$. Since Stages 0 and 3 require a fault-free row, any node in a row with a fault is declared to be *bad*. Furthermore, in Stage 1, every packet needs a sufficient number of random choices of middle nodes. For any input (or output) node m, let $REACH(m)$ be defined to be the set of middle nodes reachable from m using fault-free paths. If $|REACH(m)| < 4N/5$ for any input (or output) node m, then we define m and all other nodes in its row to be bad. Any node not defined as bad is the defined to be *good*. Note that there are no faults in rows containing good nodes, and every good input (output) node can reach at least $4N/5$ middle nodes via fault-free paths.

We will now show that only $O(f)$ rows contain bad nodes. This follows from the fact that only f rows can contain faults and from the fact that $|REACH(m)| \geq 4N/5$ for all but $O(f)$ input nodes m. The latter fact is proved by setting $t = N/5$ in the following lemma, which will also be used in Section 3.2.

Lemma 3.1.1 *In an N-input butterfly with f worst-case faults, at least $N - fN/t$ input nodes (nodes in level 0) can each reach at least $N - t$ output nodes (nodes in level $\log N$) via fault-free paths of length $\log N$, for any $t \leq N$.*

Proof: For each input node i, let n_i represent the number of output nodes in level $\log N$ that i cannot reach. If the lemma were false, then we would have

$$\sum_{i=1}^{N} n_i \geq (\frac{fN}{t} + 1)(t + 1).$$

A fault at any level of the butterfly lies on precisely N paths from input nodes to output nodes. Hence $\sum_{i=1}^{N} \leq fN$. Combining the inequalities yields $fN \geq (fN/t + 1)(t + 1)$ which is a clear contradiction. Hence the lemma must be true. □

In order to route any permutation between the good nodes, we use Valiant's algorithm except that in Stage 1, we randomly select a middle node m'' from $REACH(m) \cap REACH(m')$. (This step requires us to a store at each node m a table containing information about $REACH(m) \cap REACH(m')$ for each node m'.) Since m and m' are good input and output nodes $|REACH(m) \cap REACH(m')|$ is at least $3N/5$. We will now prove that the paths selected in this manner have congestion $O(\log N)$ with high probability. This also implies that the routing will complete in $O(\log N)$ steps with high probability [LMR88].

Theorem 3.1.2 *The paths selected have length $4 \log N$ and the congestion is $O(\log N)$ with probability more than $1 - 1/N^k$ (for any constant k).*

Proof: The lengths of the paths are clearly $4 \log N$ since the paths traverse the butterfly four times, once in each stage. We bound the congestion as follows.Every good node sends and receives one packet. The congestion of any node in Stages 0 and 3 is trivially at most $\log N$. Consider a node s in level l of the butterfly in Stage 1. There are $2^l \log N$ packets that could pass through this node. A packet passes through this node iff it selects as a random middle node, one of the $2^{\log N - l}$ middle nodes reachable from this node.

Note that the set of possible choices of middle nodes for any input node m and output node m' is $REACH(m) \cap REACH(m')$. Since both m and m' are good nodes, the cardinality of both $REACH(m)$ and $REACH(m')$ is at least $4N/5$. This implies that the cardinality of $REACH(m) \cap REACH(m')$ is at least $3N/5$. Thus the probability that the chosen middle node is reachable from s is at most $2^{\log N - l}/(3N/5)$. Therefore the average number of packets passing through a node in Stage 1 is at most $2^l \log N 2^{\log N - l} 5/(3N)$ which is $5 \log N/3$. We can use Chernoff bounds [Rag90] to show that the number of packets through s in Stage 1 is $O(\log N)$ with probability at least $1 - o(1/N^k)$. The calculation for a node in Stage 2 is exactly analagous. Thus the congestion is $O(\log N)$ with probability at least $1 - 1/N^k$. $\square$

3.1.1 Packet routing without routing tables

In the previous algorithm, each good input node m was required to store a table containing information about $REACH(m) \cap REACH(m')$ for every other node m'. In this section, we will show that is possible to route packets in a faulty butterfly even without using such routing tables. The information about the placement of the faults will be used only during the reconfiguration when the good and bad nodes are identified. This information will not be needed for the routing itself. We will assume that any packet that attempts to go through a fault is simply lost. We will further assume that a node which receives a packet sends back an acknowledgement message (ACK) to the sender. The ACK messages will follow the path of the packet in the reverse. The algorithm for routing proceeds in rounds. There will be a total of $A \log\log N$ rounds (for some constant A). Each round consists of the following steps (R takes values from 0 to $A \log\log N - 1$ and denotes the round number).

SEND-PACKET: In Stage 0, if packet p has not yet been delivered to its destination, send 2^R identical copies of p to the input node m in its row. In Stage 1, send each copy of the packet independently to a random middle node. In Stage 2, send each copy to the appropriate output node m'. In Stage 3, send each copy to the appropriate destination node in that row.

RECEIVE-PACKET: If a packet is received send an ACK along the same path that the packet came through in the reverse.

WAIT: Wait $B \log N$ steps before starting the next round.

Theorem 3.1.3 *For appropriate choices of the constants A and B, the above algorithm routes any permutation on the $(N - O(f)) \log N$ good nodes of the butterfly in $O(\log N \log\log N)$ steps with probability at least $1 - \frac{1}{N^k}$, for any fixed constant $k > 0$.*

Proof: First we show that there is very little probability that a packet survives $A \log\log N$ rounds without reaching its destination, where A is an appropriately chosen constant. A packet can never encounter a fault in Stages 0 and 3, since its source and destination rows are fault-free. Let the input node that this packet passes through be m and the output node be m'. In Stage 1, if the packet chooses any middle node in $REACH(m) \cap REACH(m')$ it will succeed in reaching its destination. Since the cardinality of $REACH(m) \cap REACH(m')$ is at least $3N/5$ the probability of this happening is at least $3/5$. Suppose the packet did not get through after $A \log\log N - 1$ rounds. Then $2^{A \log\log N}$ copies of the packet will be transmitted in the last round. Note that the probability of each copy surviving is independent of the others. Hence the probability that none of these copies reach their destination is at most $(1-3/5)^{\log^A N}$ which is at most $1/N^{k+2}$ for an appropriate choice for the constant A. Thus, the probability that some packet does not reach its destination is at most $N \log N/N^{k+2}$ which is $o(1/N^k)$.

Next we show that each round takes $O(\log N)$ time with high probability. We assume inductively that at the beginning of round i the total number of packets (counting each copy once) to be transmitted from any row in Stage 0 of the algorithm or received by any row in Stage 3 of the algorithm is at most $q \log N$ for some constant $q > 1$. Clearly the basis of the induction is true since at the beginning of the first round since there are exactly $\log N$ packets sent by each row in Stage 0 and received by each row in Stage 3. The average number of copies that were sent from a row in Stage 0 or that were destined for a row in Stage 4, that did not get through is at most $2q \log N/5$. The value of q is chosen such that the probability that more than $q \log N/2$ copies do not get through in any row can be shown to be small, i.e., $o(1/N^{k+1})$, using Chernoff bounds. At the beginning of the next stage, each unsent copy will be duplicated and hence with high probability, the number of packets in any row in the next round will be at most $q \log N$. Since there are $\log\log N$ rounds the probability that the inductive hypothesis will not hold in the beginning of any one round is at most $A \log\log N o(1/N^{k+1})$ which is $o(1/N^k)$.

Now we assume that the inductive hypothesis is true and show that each round takes only $O(\log N)$ steps with high probability. Consider any round i. From the inductive hypothesis, the congestion of any node in Stage 0 or 3 is at most $q \log N$ which is $O(\log N)$. In Stage 1, a node at level l can receive packets from any one of the 2^l input nodes, each packet with a probability of 2^{-l}. The total number of packets which pass through an input node is at most $q \log N$ by our inductive hypothesis. Therefore the average number of packets passing through the node is $q \log N 2^l 2^{-l}$ which is $q \log N$. The value of q is chosen so that the probability that any node gets more than $2q \log N$ packets can be shown to be $o(1/N^{k+1})$, using Chernoff bounds. The analysis for Stage 2 is similar. Thus we have shown that if the inductive hypothesis is true, the congestion of any node is $O(\log N)$ with high probability. Therefore, using Ranade's scheme to schedule the packets, the routing will complete $C \log N$ steps with probability at least $o(1/N^{k+1})$, for an appropriate constant C. The ACKs follow the path of packet in the reverse direction. Therefore the congestion in any node due to ACKs can be no more than the congestion due to packets and is also therefore $O(\log N)$. Since we are using Ranade's algorithm to schedule the packets, the probability that the ACKs will not reach in $D \log N$ time is at most $o(1/N^{k+1})$, for some suitably large constant D. We choose the constant B in the algorithm to be at least $C + D$ so that the algorithm waits long enough for both the packet routing and the routing of ACKs to finish before starting the next round of routing. The probability that either the packet routing or the ACK routing fails to complete in some round is at most $2A \log\log N o(1/N^{k+1})$ which is $o(1/N^k)$.

The probability that either some packet remains untransmitted after the last round or that the inductive hypothesis does not hold for some round or that some round fails to complete in $B \log N$ steps is simply $3o(1/N^k)$ which is $O(1/N^k)$. Thus the algorithm succesfully routes every packet to its destination in $O(\log N \log\log N)$ steps with probability at least $1 - O(1/N^k)$. □

If the number of worst-case faults is smaller there is a simpler way of routing without any routing tables.

Theorem 3.1.4 *Given a butterfly with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$), it is possible to identify $N - o(N)$ good nodes in the butterfly such that any permutation routing problem on the good nodes can be routed in $O(\log N)$ steps with probability greater than $1 - 1/N^k$, for any fixed constant k, without any routing tables.*

Proof: We choose the good nodes in the butterfly much in the same way as before except that our threshold is much smaller. Any good node must have fault-free row and the input node in the row of a good node must not be in $BAD(F, N^{1-\epsilon/2})$. Using Lemma 3.1.1, we can show that the number of good nodes is at least $N - \Theta(N^{1-\epsilon/2})$. The algorithm is the same as the previous algorithm except that we now need only a constant number of routing rounds with high probability. This is because each unsent packet at each round has only $\Theta(1/N^{\epsilon/2})$ probability of hitting a fault. Therefore it is sufficient to have $\Theta(k/\epsilon)$, i.e. some constant, number of rounds before every packet is delivered with a probability of at least $1 - 1/N^k$. □

3.2 Fault-tolerant circuit-switching

In this section, we examine the ability of the Beneš network to establish disjoint paths between its inputs and outputs when some of its switches fail. We assume that no path can pass through a faulty switch. The primary result of this section is that for arbitrarily small positive constants ϵ and δ, there is a constant b such that given a b-dilated $\log N$-dimensional Beneš network with $f = N^{1-\epsilon}$ worst-case switch failures, we can identify a set of $N - 2N^{1-\delta}$ input and output nodes such that it is possible to route edge-disjoint paths in any permutation between the corresponding input and output edges. (A *b-dilated* Beneš network is one in which each edge is replaced by b parallel edges, and each 2×2 switch is replaced by a $2b \times 2b$ switch.) As an example, a 2-dilated Beneš network can tolerate up to $\sqrt{N}/4$ worst-case switch failures and still route arbitrary permutations between 75 percent of the inputs and outputs.

In a $\log N$-dimensional Beneš network, levels 1 through $2\log N - 1$ can be decomposed into two disjoint sub-Beneš networks of dimension $\log N - 1$, a top sub-Beneš network, and a bottom sub-Beneš network. Note that the two paths which originate from input edges that share an input node cannot use the same sub-Beneš network. The same is true for paths that end on output edges that share the same output node. For a full permutation, there are $2N$ input-output pairs which require paths to be routed between them. The standard algorithm for setting the switches in a Beneš network, due to Waksman [Wak68], uses bipartite graph matching to split the set of $2N$ pairs into two sets of N pairs which are each then routed recursively in one of the smaller sub-Beneš networks.

We now present Waksman's algorithm with a twist. We will call this algorithm RANDSET (for RANDom switch SETting). The way RANDSET differs from Waksman's algorithm is that it randomly chooses which of the two sets of N pairs to route through the top (and bottom) sub-Beneš network. The input to RANDSET is a permutation ϕ represented as a $2N \times 2N$ bipartite graph. The nodes of the graph represent the $2N$ input edges and the $2N$ output edges of the network. An edge in the bipartite graph from input i to output $\phi(i)$ indicates that a path must be routed from i to $\phi(i)$ in the network. The first step is to merge pairs of nodes in the bipartite graph that correspond to input edges (or output edges) that share the same input (or output). The result is a 2-regular $N \times N$ bipartite graph. The second step is to split the edges of this graph into two perfect matchings, M_0 and M_1. Next, we pick a binary value for random variable X at random. If $X = 0$ then we recursively route the paths in matching M_0 through the top sub-Beneš network and those in M_1 through the bottom sub-Beneš network. If $X = 1$ we do the reverse. The following lemma shows that RANDSET chooses the path from i to $\phi(i)$ uniformly from among all possible paths.

Lemma 3.2.1 *For any i, the path chosen by algorithm RANDSET between input i and output $\phi(i)$ in a $2N$-input Beneš network passes through any of the N middle nodes (nodes in level $\log N$) with equal probability ($1/N$).*

Proof: At the first stage, the path from i to $\phi(i)$ goes to the top or the bottom sub-Beneš network with probability $1/2$ depending on whether the matching that contains the edge corresponding to this input-output pair is chosen to be routed through the top or the bottom. The decisions made at the succeeding levels of the recursion are similar and independent of all other decisions. $\square$

It is important to remember that given a permutation, the *paths themselves could be highly correlated* and determining one path gives some information about the others.

We will classify the input and output nodes of the Beneš network as either good or bad depending on whether they can reach a sufficiently large number of middle nodes. Let F be the set of faulty switches in the Beneš network. In a fault-free Beneš network, there is a path from each input (and output) node to each of the N middle nodes. The middle nodes in fact form the leaves of a complete binary tree with the input (or output) node as the root. The faults could make some of these paths unusable. We will declare an input (or output) node *bad* if the number of middle nodes that it cannot reach exceeds a certain threshold. The threshold will be chosen so that it is possible to establish edge-disjoint paths between the *good*(i.e., not bad) inputs and outputs in any permutation. (The 2 inputs coming into an input (or output) node are good or bad depending on whether the corresponding input (or output) node is good or bad).

Let $BAD(t)$ be the set of input and output nodes for which more than t middle nodes are unreachable. The first and last $\log N + 1$ levels of the Beneš network each form a $\log N$-dimensional butterfly. By using Lemma 3.1.1 for each of these butterflies, we know that $|BAD(t)| < fN/t$.

Theorem 3.2.2 *For any constants $0 < \epsilon < \delta \leq 1$, there exists a constant $b = \lceil 1 + (2-\epsilon)/(\epsilon - \delta) \rceil$ such that a b-dilated Beneš network with $N^{1-\epsilon}$ worst-case switch faults has a set of $N - 2N^{1-\delta}$ input nodes and output nodes between whose input and output edges it is possible to route any permutation using edge-disjoint paths.*

Proof: We will declare any input or output node in $BAD(N^{1+\delta-\epsilon}/2)$ to be bad. Since we need the number of good input nodes and good output nodes to be equal we may have to declare some extra input nodes or output nodes to be bad. From Lemma 3.1.1, we know that $BAD(N^{1+\delta-\epsilon}/2)$ is at most $2N^{1-\delta}$. Thus, the number of good input nodes (or output nodes) is at least $N - 2N^{1-\delta}$.

We now prove that we can route any permutation between the good inputs and good outputs using edge-disjoint paths. In this proof, we will simply show that for every permutation, such a set of paths *exists*, without showing how to compute these paths efficiently. Later, we give an efficient procedure for computing these paths.

Given a permutation ϕ on the good inputs and outputs, we will select paths using RANDSET in b rounds. In the first round, we route all the paths using RANDSET. Some of these paths pass through faults in the network. The number of paths that pass through faults is at most $2N^{1-\epsilon}$, since each fault can kill at most 2 paths. These paths are not permissible and have to be rerouted in the second round using RANDSET. Note that every good input node (or output node) has at most $N^{1+\delta-\epsilon}/2$ unreachable middle nodes. Thus, from Lemma 3.2.1, the probability that any one of the paths hits a fault in the first $\log N + 1$ levels is at most $N^{-(\epsilon-\delta)}/2$. The probability that it hits a fault in the second $\log N + 1$ levels is also at most $N^{-(\epsilon-\delta)}/2$. The net probability that the path will hit a fault is at most $N^{-(\epsilon-\delta)}$. Even though the probabilities that any two paths hit a fault is correlated, the expected number of paths that hit faults is at most $2N^{1-\epsilon}N^{-(\epsilon-\delta)}$. This implies that there is a non-zero probability that RANDSET will find paths such that at most $2N^{1-\epsilon-(\epsilon-\delta)}$ paths hit faults. Note this also means that there *exists* a way of selecting the paths so that at most $2N^{1-\epsilon-(\epsilon-\delta)}$ paths hit faults. We select paths such that this criterion is satisfied and route the paths that hit faults again using RANDSET. We continue to do the rerouting until the expected number of paths that hit faults drops below 1. At this point with non-zero probability RANDSET routes all of the paths without hitting any faults. In particular, such a set of paths exist. The expected number of paths that hit faults in the i^{th} round is $2N^{1-\epsilon-(i-1)(\epsilon-\delta)}$. Thus, for $b = \lceil 1 + (2-\epsilon)/(\epsilon-\delta) \rceil$, the number of paths that hit faults at the end of the b^{th} round is less than 1. Therefore all paths will be routed by the end of the b^{th} round. Since we use at most b rounds of routing and since each edge of the Beneš network has been replaced by b edges we will obtain edge-disjoint paths for the permutation. $\square$

3.2.1 Derandomizing RANDSET

In the proof of Theorem 3.2.2, we show the existence of edge-disjoint paths by using the fact that algorithm RANDSET will find them with non-zero probability. In this section we construct a deterministic algorithm that always finds these paths using the technique due to Raghavan [Rag88] and Spencer [Spe87] to remove the randomness. Further, like Waksman's algorithm for finding the switch settings in a fault-free Beneš network with N input nodes, the algorithm runs in $O(N \log N)$ time.

Let P be the random variable which denotes the number of paths which go through faults at some stage of rerouting. Let X be the binary random variable used by RANDSET to make its random decision to select which matching is to be routed through which sub-Beneš network. Let us further define two random variables, P_l and P_r to denote the number of paths that RANDSET routes through faults in the left butterfly and the right butterfly respectively. Let $U(P)$ be an upperbound on $E(P)$ that is defined as $E(P_l) + E(P_r)$. In the proof of Theorem 3.2.2, we used the fact that there is a non-zero probability that RANDSET will find a set of paths with at most $U(P)$ paths hitting faults. We will define a procedure DSET (for Deterministic switch SETting) that will deterministically find such a set of paths. Algorithm DSET is the same as RANDSET except that instead of selecting a random value for X, we select the "better" choice for X as follows. We compute $U(P|(X=i)) = E(P_l|(X=i)) + E(P_r|(X=i))$, for $i = \{0, 1\}$. We then choose X to be the value of i that yields the minimum of the two values computed above.

Theorem 3.2.3 *Given a (partial) permutation ϕ to be routed, Algorithm DSET deterministically computes paths such that not more than $U(P)$ paths hit faults, and has the same asymptotic running time as RANDSET.*

Proof: We will prove the theorem by induction on the size of the Beneš network. The base case is trivial. Consider an N-input Beneš network with a (partial) permutation ϕ to route. From the definition of $U(P)$ we have that the following.

$$\begin{aligned} U(P) &= E(P_l) + E(P_r) \\ &= \frac{1}{2}(E(P_l|(X=0)) + E(P_l|(X=1))) \\ &\quad + \frac{1}{2}(E(P_r|(X=0)) + E(P_r|(X=1))) \\ &= \frac{1}{2}(U(P|(X=0)) + U(P|(X=1))) \end{aligned} \tag{1}$$

Let i be value chosen for X in Step 2 of DSET. From Equation 1 it is clear that $U(P|(X=i)) \leq U(P)$. The choice of the value for X splits the set of paths P into three parts, $P_f \cup P_e, P_t$, and P_b, which are paths which pass through faults at the first stage or the last stage, paths which are chosen to be routed through the top sub-Beneš network, and paths which are chosen to be routed through the bottom sub-Beneš network respectively. Note that

$$U(P|(X=i)) = |P_f| + |P_e| + U(P_t) + U(P_b)$$

where $U(P_t)$ and $U(P_b)$ are analogously defined for the smaller $N/2$-input sub-Beneš networks. Algorithm DSET eliminates paths that hit faults at the first or last levels before recursively routing through smaller sub-Beneš networks. The number of paths eliminated in this manner is at most $|P_f| + |P_e|$. By the induction hypothesis, we can assume that no more than $U(P_t) + U(P_b)$ paths pass through faults in the smaller sub-Beneš networks. Hence DSET routes at most $U(P|(X=i))$ paths through faults. This is less than or equal to $U(P)$, which is what we were required to prove.

Now we deal with the question of how efficiently $U(P|(X=i))$, for $i \in \{0,1\}$, can be calculated in step 2 of DSET. For every node m in the Beneš network, let $REACH(m)$ be the set of middle nodes reachable from node m using fault-free paths. We can precompute the cardinality of $REACH(m)$ as follows. The value for the middle nodes are trivially known. We then compute the values for levels on both sides adjacent to levels where the values are known and continue in this manner. This takes only

$O(N \log N)$ steps of precomputation and does not affect the asymptotic time complexity of the algorithm. Given the values of $|REACH(m)|$, the values of $U(P|(X = i))$ can be easily calculated by summing up the appropriate values of $|REACH(m)|/N$. This is an $O(N)$ time computation. Since Step 1 of the algorithm takes N time just to set N switches in the first level, this will not affect the asymptotic time complexity . Hence using DSET yields the same asymptotic time complexity as RANDSET and takes time linear in the size of the Beneš network. $\square$

4 Emulations on faulty butterflies

In this section, we show that for any constant $\epsilon > 0$, a $\log N$-dimensional butterfly with $N^{1-\epsilon}$ worst-case faults (the host H), can emulate any computation of a fault-free version of itself (the guest G) with only constant slowdown. We assume that a faulty node cannot perform computations and that packets cannot route through faulty nodes. For simplicity we assume that the butterflies wrap around, i.e., the nodes of level 0 are identified with the nodes of level $\log N$.

We model the emulation of G by H as a pebbling process. There are two kinds of pebbles. With every node v of G and every time step t, we associate a *state pebble* (s-pebble), $\langle v, t\rangle$, which contains the entire state of the computation performed at node v at time t. The s-pebble contains local memory values, registers, stacks, and anything else that is required to continue the computation at v. We will view G as a directed graph by replacing each undirected edge between nodes u and v by two directed edges: one from u to v and the other from v to u. With each directed edge e and every time step t, we associate a *communication pebble* (c-pebble), $[e, t]$, which contains the message transmitted along edge e at time step t.

The host H will emulate each step t of G by creating an s-pebble $\langle v, t\rangle$ for each node v of G and a c-pebble $[e, t]$ for each edge e of G. A node of H can create an s-pebble $\langle v, t\rangle$ only if contains s-pebble $\langle v, t-1\rangle$ and all of the c-pebbles $[e, t-1]$, where e is an edge into v. It can create a c-pebble $[g, t]$ for an edge g out of v only if it contains an s-pebble $\langle v, t\rangle$. A node of H can also transmit a c-pebble to a neighboring node in H in unit time. A node of H is not permitted to transmit an s-pebble since an s-pebble may contain a lot of information. Note that H can create more than one copy of an s-pebble or c-pebble. The ability of H to create redundant pebbles is crucial to our emulation schemes. In our emulations, each node of H is assigned a fixed set of nodes of G to emulate, and creates s-pebbles for them for each time step.

4.1 Assignment of nodes of G to nodes of H

We now show how to map the computation of G to the faulty butterfly H. The host H has $N^{1-\epsilon}$ arbitrarily distributed faults. We will first divide H into sub-butterflies of dimension $\epsilon \log N/2$, at levels $i\epsilon \log N/2$, for integer $i = 0$ to $2/\epsilon - 1$. (Without loss of generality, we will assume that $2/\epsilon$ and $\epsilon \log N/4$ are integers.) The butterfly contains as subgraphs $N^{1-\epsilon/2}$ sub-butterflies between each of these levels of division. There are a total of $(2/\epsilon)N^{1-\epsilon/2}$ sub-butterflies in all. The faults in the network may make some of these sub-butterflies unusable. We will identify "good" and "bad" sub-butterflies according to the following rules.

RULE 1: A sub-butterfly that contains a node that lies in a butterfly row in which there is a fault is a *bad* sub-butterfly (even if the fault lies outside of the sub-butterfly).

RULE 2: In order to apply Rule 2, we embed a Beneš network in the butterfly. The edges of the first stage of the Beneš network traverse the butterfly in increasing order of dimension and the edges of the second stage in decreasing order of dimension. The input nodes, the middle nodes as well as the output nodes of the Beneš network are embedded in level 0 of the butterfly (which is the same as level $\log N$). For $\delta = 2\epsilon/3$, identify the set of bad inputs/outputs (they are the same set here) according to the procedure outlined in the proof of Theorem 3.2.2 in Section 3.2. Any sub-butterfly that contains a node that has a bad input/output at the end of its butterfly row is a bad sub-butterfly.

Lemma 4.1.1 *For any $\epsilon > 0$, the number of rows in which there is either a fault or a bad input or output is at most $N^{1-\epsilon} + 2N^{1-2\epsilon/3}$.*

Proof: The number of rows containing a fault is at most $N^{1-\epsilon}$, since there are at most $N^{1-\epsilon}$ faults. By Theorem 3.2.2, for $\delta = 2\epsilon/3$, the number of bad inputs and outputs is at most $2N^{1-\delta} = 2N^{1-2\epsilon/3}$. □

Lemma 4.1.2 *For any $\epsilon > 0$, at least half the sub-butterflies of H are good, for sufficiently large N.*

Proof: The total number of sub-butterflies is $(2/\epsilon)N^{1-\epsilon/2}$. By Lemma 4.1.1, the number of rows containing either a fault or bad input or output is at most $N^{1-\epsilon} + 2N^{1-2\epsilon/3}$. Since each bad row passes through $2/\epsilon$ different sub-butterflies, the total number of sub-butterflies identified as bad by Rules 1 and 2 cannot exceed $2(N^{1-\epsilon} + 2N^{1-2\epsilon/3})/\epsilon$. Observe that $2(N^{1-\epsilon} + 2N^{1-2\epsilon/3})/\epsilon$ is at most $N^{1-\epsilon/2}/\epsilon$, for sufficiently large N. $\square$

Now we will divide the guest G into overlapping sub-butterflies of dimension $\epsilon \log N/2$ and map them to the good sub-butterflies of H. For any node v of G, two nodes of H will receive the initial state of the computation of node v, i.e., s-pebble $\langle v, 0\rangle$. These two nodes in H will create the s-pebbles for v. The mapping proceeds as follows. Take the guest G and cut it into sub-butterflies at levels $i\epsilon \log N/2$, for integers $i = 0$ to $2/\epsilon - 1$. Map each sub-butterfly to a good sub-butterfly of H evenly such that each sub-butterfly of H gets at most two sub-butterflies of G. Now cut G again, this time at levels $(\epsilon/4 + i\epsilon/2)\log N$, for integers $i = 0$ to $2/\epsilon - 1$. Map the sub-butterflies evenly to good sub-butterflies of H as before. At most four nodes of G are mapped to each node of H.

4.2 Building constant congestion paths

We call the nodes v of G belonging to levels $i\epsilon \log N/4 - 1$ and $i\epsilon \log N/4$, for $i = 0$ to $4/\epsilon - 1$, *boundary nodes* since they lie on the boundary of some sub-butterfly that was cut out of G. Let the set of boundary nodes of G be denoted by $\mathcal{B}_\mathcal{G}$. Similarly we define the nodes in the levels where H was cut to form sub-butterflies, i.e., levels $i\epsilon \log N/2$, and $i\epsilon \log N/2 - 1$, for $i = 0$ to $2/\epsilon - 1$, the boundary nodes of H. Let us denote this set $\mathcal{B}_\mathcal{H}$.

Let ϕ be the function that maps an s-pebble, $\langle v, t\rangle$ to the node in H that creates it. The creation of $\langle v, t\rangle$ requires that node $\phi(\langle v, t\rangle)$ of H gets all the c-pebbles $[e, t]$ from some other node of H, for every edge e into v. Suppose that $\langle v, t\rangle$ is mapped to some node $m = \phi(\langle v, t\rangle)$ in the interior of a good sub-butterfly of H. Then the neighbors of m in H also contain the s-pebbles of the neighbors of v in G. Therefore m will receive the required c-pebbles from all its neighbors in H.

On the other hand, if the s-pebble for v is mapped to some node $m \in \mathcal{B}_\mathcal{H}$, the neighbors of m in H may not create the s-pebbles of the neighbors of v in G. However, since every node v of G is mapped to two nodes of H, there is another node m' of H which also creates an s-pebble for v. Node m' of H will send c-pebbles for *all* the edges e into v to node m. It is important to note that by the property of our mapping that *m' is necessarily a node in the center of a sub-butterfly of H*, i.e., in level $\epsilon \log N/4$ or $\epsilon \log N/4 - 1$ of the sub-butterfly.

To facilitate the transmission of c-pebbles we will establish constant congestion fault-free paths in H, using the results of Section 3.2, between all pairs of nodes m and m' of H that create the s-pebbles for the same node v in $\mathcal{B}_{\mathcal{G}}$. The number of paths originating in a row of H is simply the number of nodes mapped to sub-butterfly boundaries in that row, which is at most $4 \cdot 2/\epsilon$, i.e., a constant. Similarly the number of paths ending in any row is $8/\epsilon$. We can divide the paths into $8/\epsilon$ sets such that each set has at most one path originating in a row and one path ending in a row. Note that all paths start and end in rows which have good inputs and outputs for doing Beneš-type routing. Therefore, each set can be routed with constant congestion using the results of Section 3.2. Since there are only a constant number of such sets the total congestion is also a constant.

4.3 The emulation

We will now formally describe the emulation and prove its properties. Initially, nodes of H contain s-pebbles $\langle v, 0\rangle$ for nodes v of G. We say that H has emulated T steps of the computation of G iff for every node v, an s-pebble $\langle v, T\rangle$ has been created. The emulation algorithm is executed by every node m of H and proceeds as a sequence of macro-steps. Each macro-step consists of the following three sub steps.

1. Computation step. For each node v of G that has been assigned to m, m creates a new s-pebble $\langle v, t\rangle$, provided that m already contains s-pebble $\langle v, t-1\rangle$ and c-pebbles $[e, t-1]$ for every edge e into v.

2. Communication step. For every node v whose s-pebble was updated from $\langle v, t-1\rangle$ to $\langle v, t\rangle$ in the computation step, node m sends a c-pebble to each of its neighbors in the sub-butterfly of H that contains m. In addition, if m is a node in one of the center levels (levels $\epsilon \log N/4$ or $\epsilon \log N/4 - 1$ in the sub-butterfly), it starts four c-pebbles, one for each edge out of v, along on their way to the node m' that also creates s-pebbles for v.

3. Routing step. Node m moves every c-pebble $[e, t]$ passing through it en route to its destination, one step closer to its destination.

Lemma 4.3.1 *Each macro-step takes only a constant number of time steps to execute.*

Proof: There are at most 4 s-pebbles mapped to each node. Therefore the computation step takes constant time. Every s-pebble that is updated can cause at most 8 c-pebbles to be sent. Therefore the communication step takes only constant time. There can be only a constant number of c-pebbles in transit residing at any node at any time step. The reason is that there are only a constant number of paths passing through every node. Furthermore, since every c-pebble moves in every macro-step and only a constant number of c-pebbles enter a particular path at any macro-step, there can be only a constant number of c-pebbles on a particular path resident at a particular node at a particular time. Thus the routing step also takes only a constant number of time steps. □

Theorem 4.3.2 *Any computation on a fault-free butterfly G that takes time T can be emulated in $O(T + \log N)$ time by H.*

Proof: We will show that we require only $O(T + \log N)$ macro-steps for a T-step computation of G. The final result will then follow from Lemma 4.3.1.

Let ϕ be the function that maps an s-pebble, $\langle v, t\rangle$ to the node in H that creates it. The *dependency tree* of an s-pebble represents the functional dependency of this s-pebble on other s-pebbles and can be defined recursively as follows. As the base case, if $t = 0$, the dependency tree of $\langle v, t\rangle$ is a single node, $\langle v, 0\rangle$. If $t > 0$, the creation of s-pebble, $\langle v, t\rangle$, requires s-pebble $\langle v, t-1\rangle$ and c-pebbles $[e, t-1]$ such that e is an incoming edge of node v in G. These c-pebbles are sent by some other s-pebbles. There are two cases. If $\phi(\langle v, t\rangle)$ is a node in $\mathcal{B}_{\mathcal{H}}$, then s-pebble $\langle v, t\rangle$ gets all its c-pebbles from the other s-pebble for node v which we will denote by $\langle v, t-1\rangle'$. If $\phi(\langle v, t\rangle)$ is not a node in $\mathcal{B}_{\mathcal{H}}$, then it gets its c-pebbles from s-pebbles $\langle u, t-1\rangle$ such that u is a neighbor of v in G and $\phi(\langle u, t-1\rangle)$ is a neighbor of $\phi(\langle v, t\rangle)$ in H. The dependency tree of $\langle v, t\rangle$ is defined recursively as follows. The root of the tree is $\langle v, t\rangle$. The subtrees of this tree are the dependency trees of $\langle v, t-1\rangle$ and $\langle v, t-1\rangle'$ in the first case and the dependency trees of $\langle v, t-1\rangle$ and all s-pebbles $\langle u, t-1\rangle$ in the second case.

Let the emulation of T steps of G take T' macro-steps on H. Let $\langle v, T\rangle$ be an s-pebble that was updated in the last macro-step. We will now look at the dependency tree of $\langle v, T\rangle$. For every tree node s, we can associate a time (in macro-steps) $\tau(s)$ when that s-pebble was created. We choose a *critical path*, $s_T, s_{T-1}, \cdots, s_0$, of tree nodes from the root to the leaves of the tree as follows. The root of the tree, s_T, is $\langle v, T\rangle$. The creation of s_T requires the s-pebble $\langle v, T-1\rangle$ and c-pebbles $[e, T-1]$. If the s-pebble

$\langle v, T-1\rangle$ was created after all the c-pebbles were received then choose s_{T-1} to be $\langle v, T-1\rangle$. Otherwise, choose the s-pebble which sent the c-pebble that arrived last at node $\phi(\langle v, T\rangle)$. After choosing s_{T-1}, we choose the rest of the sequence recursively in the subtree with s_{T-1} as the root. We will define a quantity l_i as follows. If $\phi(s_i)$ and $\phi(s_{i-1})$ are the same node or neighbors in H, then $l_i = 1$. Otherwise, l_i is the length of the path by which a c-pebble generated by s_{i-1} is sent to s_i. From the definition of our critical path and because a c-pebble moves once in every macro-step, $\tau(s_i) - \tau(s_{i-1}) = l_i$. Thus,

$$T' = \sum_{0<i\leq T} (\tau(s_i) - \tau(s_{i-1})) = \sum_{0<i\leq T} l_i.$$

Now suppose that some l_i is greater than one. This corresponds to some long path taken by some c-pebble to go from $\phi(s_{i-1})$ in the center level of a sub-butterfly of H to $\phi(s_i)$ in $\mathcal{B}_{\mathcal{H}}$. Thus l_i is the length of the constant congestion paths and is at most $4\log N$. The key observation is that since $\phi(s_{i-1})$ is a node in the center level, working down the tree from s_{i-1} there can be no long paths until we reach an s-pebble mapped to the boundary $\mathcal{B}_{\mathcal{H}}$, i.e., $l_{i-j} = 1$ for $1 \leq j \leq \epsilon \log N/4 - 1$. Thus $T' = \sum_{0<i\leq T} l_i$ can be no more than $(16/\epsilon + 1)T + 4\log N$ which is $O(T + \log N)$. □

We can extend these results to the shuffle-exchange network using Schwabe's proof [Sch90] that an N-node butterfly can emulate an N-node shuffle-exchange network with constant slowdown, and vice versa.

Theorem 4.3.3 *Any computation on a fault-free N-node shuffle-exchange network G which takes time T can be emulated in $O(T + \log N)$ time by a an N-node shuffle-exchange network H with $N^{1-\epsilon}$ worst-case faults, for any constant $\epsilon > 0$.*

Proof: Schwabe [Sch90] shows how to emulate the computation of a butterfly on a shuffle-exchange network and vice versa. First we use this result to map the computation of a butterfly B to the faulty shuffle-exchange network H. Any node of B mapped to a faulty node of H is declared faulty. If there are any routing paths required for the emulation passing through this faulty node of H we declare the nodes of B that use this path as faulty. The number of faults in B will only be a constant factor more than $N^{1-\epsilon}$, since both the load and the congestion of the paths used in the emulation are constant. Now we use Theorem 4.3.2, to emulate a fault-free butterfly, B', on B. We again use Schwabe's result to emulate the fault-free shuffle-exchange network, G, on the fault-free butterfly B'. Each of these emulations has constant slowdown. Therefore the entire emulation of G on H has constant slowdown. □

4.4 Emulating normal algorithms on the hypercube

Many practical computations on the hypercube are structured. The class of algorithms in which every node of the hypercube uses exactly one edge for communication at every time step and further all of the edges used in a time step belong to the same dimension of the hypercube are called *leveled algorithms* (also known as *regular algorithms* [BCS92]). A useful subclass of leveled algorithms are *normal algorithms*. A normal algorithm has the additional restriction that the dimensions used in consecutive time steps are consecutive. Many algorithms such as bitonic sort, FFT, and tree-based algorithms like branch-and-bound can be implemented on the hypercube as normal algorithms[Lei92]. An additional property of normal algorithms is that they can be emulated efficiently by bounded-degree networks such as the shuffle-exchange network and the butterfly. We state a result due to Schwabe [Sch91] to this effect.

Lemma 4.4.1 *An N-node butterfly can emulate any normal algorithm of an N-node hypercube with constant slowdown.*

We further require a well known result of the butterfly in the hypercube (See [GHR90] for a stronger result that the butterfly is a subgraph of the hypercube).

Lemma 4.4.2 *An N-node butterfly can be embedded in an N-node hypercube with constant load, congestion, and dilation.*

Theorem 4.4.3 *An N-node hypercube with $N^{1-\epsilon}$ worst-case faults (for any $\epsilon > 0$) can emulate any normal algorithm on an N-node fault-free hypercube with only constant slowdown.*

Proof: Let the faulty N-node hypercube be H and the fault-free N-node hypercube be G. H has some $N^{1-\epsilon}$ faulty nodes. H emulates any normal algorithm of G in using a sequence of constant slowdown simulations. Let an N-node butterfly, B, be embedded in H in the manner of Lemma 4.4.2. Any node of B which is mapped on to a faulty node of H will be considered faulty. Since this is a constant load embedding the number of faulty nodes in B is $O(N^{1-\epsilon})$. Clearly, H can emulate any computation of B (the faulty nodes of B do no computation) with constant slowdown using the constant load, dilation and congestion embedding of B in H. Let B' be a fault-free N-node butterfly. From Theorem 4.3.2, B can emulate B' with a constant slowdown. Now, from Lemma 4.4.1, B' can emulate any normal algorithm of G with a constant slowdown. Putting all these simulations together, we can obtain a constant slowdown simulation of any normal algorithm on G on the faulty hypercube H. $\square$

5 Random faults

In this section we show that an N-input host butterfly H can sustain many random faults and still emulate a fault-free N-input guest butterfly G with little slowdown. In particular, we show that if each node in H fails independently with probability $p = 1/\log^{(k)} N$, where $\log^{(k)}$ denotes the logarithm function iterated k times, the slowdown of the emulation is $2^{O(k)}$, with high probability. For any constant k this slowdown is constant. Furthermore, for $k = O(\log^* N)$ the node failure probability, p, is constant, and the slowdown is $2^{O(\log^* N)}$. Previously, the most efficient self-emulation scheme known for an N-input butterfly required $\omega(\log\log N)$ slowdown [Tam92a].

The proof has the following outline. We begin by showing that the host, H, can emulate another N-input butterfly network B_k with constant slowdown. As in H, some of the nodes in B_k may fail at random (in which case it is not necessary for H to emulate them), but B_k is likely to contain fewer faults than H. In turn, B_k can emulate another butterfly B_{k-1} with even fewer faults. Continuing in this fashion, we arrive at B_1, which with high probability, contains so few faults that it can emulate the guest, G, with constant slowdown. There are $k+1$ emulations, and each incurs a constant factor slowdown, so the total slowdown is $2^{O(k)}$.

5.1 Emulating a butterfly with fewer faults

We begin by explaining how H emulates B_k. The first step is to cover the N-input butterfly B_k with overlapping $(\log^{(k)} N)^2$-input sub-butterflies. For ease of notation, let $M_k = (\log^{(k)} N)^2$ (For simplicity, we assume that $\log M_k$ is an integral multiple of 4). For each i from 0 to $4\log N/\log M_k$, there is a band of disjoint M_k-input sub-butterflies spanning levels $(i \log M_k)/4$ through $((i+4)\log M_k)/4$. We call these sub-butterflies the *band* i sub-butterflies. Note that each band i sub-butterfly shares $M_k^{3/4}$ rows with $M_k^{1/4}$

different band $i-1$ sub-butterflies, and $M_k^{3/4}$ rows with $M_k^{1/4}$ different band $i+1$ sub-butterflies.

Each M_k-input sub-butterfly in B_k will be emulated by the corresponding sub-butterfly in H. We say that an M_k-input sub-butterfly in B_k *fails* if more than $\alpha\sqrt{M_k}\log M_k$ nodes inside the corresponding M_k-input sub-butterfly in H fail, where α is a constant that will be determined later. If a sub-butterfly in B_k fails, then H is not required to emulate any of the nodes that lie in that sub-butterfly. As we shall see, if it does not fail, then the corresponding sub-butterfly in H contains few enough faults that we can treat them as worst-case faults, and apply the technique from Section 4 to reconfigure around them. The following lemma bounds the probability that a sub-butterfly in B_k fails.

Lemma 5.1.1 *For $\alpha > 2e$, an M_k-input sub-butterfly in B_k fails with probability at most $1/\log^{(k-1)} N$.*

Proof: An M_k-input sub-butterfly fails if the corresponding M_k-input sub-butterfly in H contains too many faults. An M_k-input sub-butterfly in H contains a total of $M_k \log M_k = 2(\log^{(k)} N)^2(\log^{(k+1)} N)$ nodes, each of which fails with probability $1/\log^{(k)} N = 1/\sqrt{M_k}$. Thus, expected number of nodes that fail is $\sqrt{M_k}\log M_k = 2(\log^{(k)} N)(\log^{(k+1)} N)$. Since each node fails independently, we can bound the probabilty that more than $\alpha\sqrt{M_k}\log M_k$ nodes fail using a Chernoff-type bound. For $\alpha > 2e$, the probability that more than $\alpha\sqrt{M_k}\log M_k$ nodes fail is at most $2^{-\alpha\sqrt{M_k}\log M_k}$ (for a proof, see [Rag90]). Since $\alpha\sqrt{M_k}\log M_k > \log^{(k)} N$, this probability is at most $2^{-\log^{(k)} N} = 1/\log^{(k-1)} N$. $\square$

The next Lemma shows that if a sub-butterfly in B_k does not fail, then the corresponding sub-butterfly in H can emulate it with constant slowdown.

Lemma 5.1.2 *If an M_k-input sub-butterfly in B_k does not fail, then the corresponding sub-butterfly in H can emulate it with constant slowdown.*

Proof: Since the number of faults in an M_k-input sub-butterfly that does not fail is most $\sqrt{M_k}\log M_k$, we can treat them as worst-case faullts and apply Theorem 4.3.2 with $\epsilon \approx 1/2$. $\square$

The next lemma shows that the host H can emulate any computation performed by an N-input butterfly network B_k with constant slowdown. Recall that H is not required to emulate nodes in B_k that lie in sub-butterflies in B_k that have have failed.

Lemma 5.1.3 *The host H can emulate B_k with constant slowdown.*

Proof: By Lemma 5.1.2, each M_k-input sub-butterfly in B_k that has not failed can be emulated by the corresponding sub-butterfly in H with constant slowdown using the technique of Section 4. (Note that each node in B_k may be emulated by as many as four different sub-butterflies in H.) In order to emulate the entire network B_k, it is also necessary to emulate the connections between the sub-butterflies. As in Section 4, let $M_k^{1-\epsilon}/2$ denote the number of faults in an M_k-input sub-butterfly of H. For a sub-butterfly that has not failed, $\epsilon \approx 1/2$. By Lemma 4.1.1, the number of rows containing either a fault or an input or output that is bad for Beneš routing is at most $M_k^{1-\epsilon}/2 + M_k^{1-2\epsilon/3}$, which is approximately $M_k^{2/3}$. Each band i sub-butterfly that does not fail shares $M_k^{3/4}$ rows with each of the band $i-1$ sub-butterflies (and band $i+1$ sub-butterflies) with which it overlaps. Thus, for each pair of overlapping butterflies, most of the shared rows are both fault-free and good for routing in both sub-butterflies. The emulation strategy of Section 4 covers each M_k-input sub-butterfly in H with smaller sub-butterflies, each having $M_k^{\epsilon/2}$ inputs. If a smaller sub-butterfly is used in the emulation, then none of the rows that pass through it contain either fault or a bad input or output. Thus, the $M_k^{3/4}$ connections between the two sub-butterflies in bands i and $i-1$ (and $i+1$) can be emulated by routing constant congestion paths of length $O(\log M_k)$ through the shared rows. The rest of the proof is similar to that of Theorem 4.3.2. □

5.2 Emulating a series of butterflies

So far we have shown that the host H can emulate an N-input butterfly B_k that contains some faulty nodes. Although our ultimate goal is to show that H can emulate the guest network G, which contains no faulty nodes, we have made some progress. In the host network, H, each node fails independently with probability $1/\log^{(k)} N$. In B_k, each $(\log^{(k)} N)^2$-input sub-butterfly fails with probability $1/\log^{(k-1)} N$. A node in B_k fails if it lies in a sub-butterfly that fails. Since each node in B_k lies in at most four $(\log^{(k)} N)^2$-input sub-butterflies we have reduced the expected number of faults from $(N \log N)/\log^{(k)} N$ in H to fewer than $4N\log N/\log^{(k-1)} N$ in B_k.

The next step is to show that butterfly B_k can emulate a butterfly B_{k-1} with even fewer faults. In general, we will cover butterfly B_j with $(\log^{(j)} N)^2$-input sub-butterflies. For ease of notation, let $M_j = (\log^{(j)} N)^2$. We say that an M_j-input sub-butterfly in B_j fails if the corresponding M_j-input sub-butterfly in B_{j+1} contains more than $\alpha\sqrt{M_j}$ M_{j+1}-input sub-butterflies that have failed. The following three lemmas are analogous to Lemmas 5.1.1 through 5.1.3.

Lemma 5.2.1 *For $\alpha > 2e$, the probability that an M_j-input sub-butterfly in B_j fails is at most $1/(\log^{(j-1)} N)$.*

Proof: The proof is by induction on j, starting with $j = k$ and working backwards to $j = 0$. The base case is given by Lemma 5.1.1. For each value of i from 0 to $4 \log M_j / \log M_{j+1}$, there is a band of disjoint M_{j+1}-input sub-butterflies in B_{j+1} that span levels $(i \log M_{j+1})/4$ through $((i+4) \log M_{j+1})/4$. These M_{j+1}-input sub-butterflies can be partitioned into four disjoint *classes* according to their band numbers. Two bands of sub-butterflies belong to the same class if their band numbers differ by a multiple of four. There are at most

$$M_j \log M_j / M_{j+1} \log M_{j+1} = (\log^{(j)} N)^2 / (\log^{(j+1)} N)(\log^{(j+2)} N)$$

sub-butterflies in each of these classes, and within a class, the sub-butterflies are disjoint. By induction, each sub-butterfly fails with probability at most $1/\log^{(j)} N$. Thus, in any particular class, the expected number of sub-butterflies that fail is at most $(\log^{(j)} N)/(\log^{(j+1)} N)(\log^{(j+2)} N)$, which is less than $\log^{(j)} N$. Using Chernoff-type bounds as in Lemma 5.1.1, for $\alpha > 2e$, the probability that more than $\alpha \log^{(j)} N = \alpha\sqrt{M_j}$ of these sub-butterflies fail is at most $2^{-\alpha \log^{(j)} N}$, which is less than $1/4 \log^{(j-1)} N$. Thus, the probability that a total of $\alpha \log^{(j)} N$ sub-butterflies fail in the four classes is at most $1/\log^{(j-1)} N$. $\square$

Lemma 5.2.2 *If an M_j-input sub-butterfly in B_j does not fail, then the corresponding M_j-input sub-butterfly in B_{j+1} can emulate it with constant slowdown.*

Proof: If a M_j-input sub-butterfly does not fail, then at most $\alpha\sqrt{M_j}$ of the overlapping M_{j+1}-input sub-butterflies in the corresponding M_j-input sub-butterfly in H fail. Each of these sub-butterflies contains $M_{j+1} \log M_{j+1} = 2(\log^{(j+1)} N)^2 \log^{(j+2)} N$ nodes. Since the total number of nodes in all of these sub-butterflies is at most $2\alpha \log^{(j)} N (\log^{(j+1)} N)^2 \log^{(j+2)} N$, i.e., approximately $\sqrt{M_j}$, we can treat them all as if they were worst-case faults and apply Theorem 4.3.2 with $\epsilon \approx 1/2$. $\square$

Lemma 5.2.3 *For $1 < j \leq k+1$, butterfly B_j can emulate B_{j-1} with constant slowdown.*

Proof: The proof is similar to the proof of Lemma 5.1.3 □

Theorem 5.2.4 *For any fixed $\gamma > 0$, with probability at least $1 - 1/2^{N^{1-\gamma}}$, an N-input butterfly in which each node fails with probability $1/\log^{(k)} N$ can emulate a fault-free N-input butterfly with slowdown $2^{O(k)}$.*

Proof: The host network, $H = B_{k+1}$, can emulate network B_1 with slowdown $2^{O(k)}$. In B_1, each sub-butterfly with $(\log N)^2$ inputs fails with probability $1/\log^{(0)} N = 1/N$. Using Chernoff-type bounds as in Lemma 5.1.1, the probability that more than $N^{1-\gamma}$ of these sub-butterflies fail is at most $1/2^{N^{1-\gamma}}$. If fewer than $N^{1-\gamma}$ of them fail, then we can treat the nodes contained in these sub-butterflies as if they were worst-case faults. In this case, the total number of worst-case faults is at most $2N^{1-\gamma}\log^2 N \log\log N$. Hence, by applying Theorem 4.3.2 with $\epsilon \approx \gamma$, B_1 can emulate the guest network G with constant slowdown. □

6 Open problems

Some of the interesting problems left open by this paper are listed below.

1. Can the butterfly (or any other bounded-degree network) tolerate random-faults with constant failure probability and still simulate itself with constant slowdown?

2. Can an N-node butterfly (or any other N-node bounded-degree network) tolerate more than $N^{1-\epsilon}$ worst-case faults (e.g., $N/\log N$) and still simulate itself with constant slowdown?

3. Can an N-input Beneš network tolerate $\Theta(N)$ worst-case switch failures and still route disjoint paths in any permutation between some set of $\Theta(N)$ inputs and outputs?

4. Can an N-input butterfly be embedded with constant load, congestion, and dilation in an N-input butterfly with more than $\log^{O(1)} N$ (e.g., $N^{1-\epsilon}$) worst-case faults?

7 Acknowledgements

Thanks to Bob Cypher and Joel Friedman for helpful discussions. We are also grateful to Bill Aiello for suggesting that our butterfly results imply results in fault-tolerant emulations of normal algorithms. The third author wishes to thank Bob Tarjan for his support and encouragement.

References

[AAB+92] M. Ajtai, N. Alon, J. Bruck, R. Cypher, C. T. Ho, M. Naor, and E. Szemerédi. Fault tolerant graphs, perfect hash functions and disjoint paths. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992. To appear.

[AB91] Y. Aumann and M. Ben-Or. Asymptotically optimal PRAM emulation on faulty hypercubes. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 440–457. IEEE Computer Society Press, October 1991.

[Aie92] W. A. Aiello, July 1992. Personal communication.

[AL91] W. Aiello and T. Leighton. Coding theory, hypercube embeddings, and fault tolerance. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 125–136, July 1991.

[ALM90] S. Arora, T. Leighton, and B. Maggs. On-line algorithms for path selection in a non-blocking network. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 149–158, May 1990.

[Ann89] F. Annexstein. Fault tolerance in hypercube-derivative networks. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 179–198, June 1989.

[AS82] G. B. Adams and H. J. Siegel. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Transactions on Computers*, C–31(5):443–454, May 1982.

[AU90] S. Assaf and E. Upfal. Fault-tolerant sorting network. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 275–284, October 1990.

[BCH91] J. Bruck, R. Cypher, and C.–T. Ho. Fault-tolerant meshes with minimal numbers of spares. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 288–295, December 1991.

[BCLR92] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Tolerating faults in synchronization networks. Technical Report 92–14, University of Massachusetts, Amherst, MA, April 1992.

[BCS90] J. Bruck, R. Cypher, and D. Soroker. Running algorithms efficiently on faulty hypercubes. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 37–44, June 1990.

[BCS92] J. Bruck, R. Cypher, and D. Soroker. Tolerating faults in hypercubes using subcube partitioning. *IEEE Transactions on Computers*, 41(5):599–605, May 1992.

[DH90] S. Dutt and J. P. Hayes. On designing and reconfiguring k-fault-tolerant tree architectures. *IEEE Transactions on Computers*, C–39(4):490–503, April 1990.

[DH91] S. Dutt and J. P. Hayes. Designing fault-tolerant systems using automorphisms. *Journal of Parallel and Distributed Computing*, 12:249–268, 1991.

[EL92] S. Even and A. Litman. Layered Cross Product – a technique to construct interconnection networks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 60–69, July 1992.

[GE84] J. W. Greene and A. El Gamal. Configuration of VLSI arrays in the presence of defects. *Journal of the ACM*, 31(4):694–717, October 1984.

[GHR90] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Mathematical Systems Theory*, 1990. To appear.

[GL89] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.*

Volume 5 of *Advances in Computing Research*, pages 345–374. JAI Press, Greenwich, CT, 1989.

[HLN87] J. Hastad, T. Leighton, and M. Newman. Reconfiguring a hypercube in the presence of faults. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 274–284, May 1987.

[HLN89] J. Hastad, T. Leighton, and M. Newman. Fast computation using faulty hypercubes. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 251–263, May 1989.

[KKL+90] C. Kaklamanis, A. R. Karlin, F. T. Leighton, V. Milenkovic, P. Raghavan, S. Rao, C. Thomborson, and A. Tsantilas. Asymptotically tight bounds for computing with faulty arrays of processors. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 285–296. IEEE Computer Society Press, October 1990.

[KLM+89] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 227–240, May 1989.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[Lin92] G. Lin. Fault tolerant planar communication networks. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 133–139, May 1992.

[LM92] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41(5):578–587, May 1992.

[LMP91] T. Leighton, Y. Ma, and C. G. Plaxton. Highly fault-tolerant sorting circuits. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 458–469. IEEE Computer Society Press, October 1991.

[LMR88] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–271. IEEE Computer Society Press, October 1988.

[LMS92] T. Leighton, B. Maggs, and R. Sitaraman. On the fault tolerance of some popular bounded-degree networks. Technical Report CS–TR–385–92, Department of Computer Science, Princeton University, Princeton, NJ, September 1992.

[LSGH87] M. Livingston, Q. Stout, N. Graham, and F. Hararay. Subcube fault-tolerance in hypercubes. Technical Report CRL-TR-12-87, University of Michigan Computing Research Laboratory, September 1987.

[Lyu90] Y.–D. Lyuu. Fast fault-tolerant parallel communication and on-line maintenance using information dispersal. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 378–387, July 1990.

[MS92] B. M. Maggs and R. K. Sitaraman. Simple algorithms for routing on butterfly networks with bounded queues. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 150–161, May 1992.

[OT71] D. C. Opferman and N. T. Tsao-Wu. On a class of rearrangeable switching networks–part II: Enumeration studies and fault diagnosis. *Bell System Technical Journal*, 50(5):1601–1618, May–June 1971.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), April 1989.

[Rag88] P. Raghavan. Probabilistic construction of deterministic algorithms: approximate packing integer programs. *Journal of Computer and System Sciences*, 37(4):130–143, October 1988.

[Rag90] P. Raghavan. Lecture notes on randomized algorithms. Research Report RC 15340 (#68237), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, January 1990.

[Sch90] E. J. Schwabe. On the computational equivalence of hypercube-derived networks. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 388–397, July 1990.

[Sch91] E. J. Schwabe. *Efficient Embeddings and Simulations for Hypercubic Networks.* PhD thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, June 1991.

[Spe87] J. Spencer. *Ten Lectures on the Probabilistic Method.* SIAM, Philadelphia, PA, 1987.

[SR80] S. Sowrirajan and S. M. Reddy. A design for fault-tolerant full connection networks. In *Proceedings of the International Conference on Science and Systems*, pages 536–540, March 1980.

[Tam92a] H. Tamaki. Efficient self-embedding of butterfly networks with random faults. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992. To appear.

[Tam92b] H. Tamaki. Robust bounded-degree networks with small diameters. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 247–256, July 1992.

[Val82] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[Wak68] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, January 1968.

[WC92] A. Wang and R. Cypher. Fault-tolerant embeddings of rings, meshes and tori in hypercubes. Technical Report IBM RJ 8569, IBM Almaden Research Center, January 1992.

[WCM91] A. Wang, R. Cypher, and E. Mayr. Embedding complete binary trees in faulty hypercubes. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 112–119. IEEE Computer Society Press, December 1991.