SIAM J. COMPUT.
Vol. 27, No. 5, pp. 1303–1333, October 1998

007

ON THE FAULT TOLERANCE OF SOME POPULAR BOUNDED-DEGREE NETWORKS*

F. THOMSON LEIGHTON†, BRUCE M. MAGGS‡, AND RAMESH K. SITARAMAN§

Abstract. In this paper, we analyze the fault tolerance of several bounded-degree networks that are commonly used for parallel computation. Among other things, we show that an N-node butterfly network 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. The same result is proved for the shuffle-exchange network. 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 network of the same type and size with a slowdown of $2^{O(\log^* N)}$. These emulation schemes combine the technique of redundant computation with new algorithms for routing packets around faults in hypercubic networks. We also present techniques for tolerating faults that do not rely on redundant computation. These techniques tolerate fewer faults but are more widely applicable because they can be used with other networks such as binary trees and meshes of trees.

Key words. fault tolerance, network emulation, butterfly network

AMS subject classifications. 68M07, 68M10, 68M15, 68Q68

PII. S0097539793255163

1. Introduction.

In this paper, we analyze the effect of faults on the computational power of bounded-degree networks such as the butterfly network, the shuffle-exchange network, and the 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 a 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 faults since an edge fault can always be simulated by disabling the node at each end of the edge.

There are several ways to measure the effect of faults on a network. In this paper, we are primarily concerned with the amount by which a collection of faults can slow down some computation on the network. For example, if a butterfly network

* Received by the editors September 9, 1993; accepted for publication (in revised form) July 20, 1996; published electronically May 19, 1998.
http://www.siam.org/journals/sicomp/27-5/25516.html

† Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (ftl@math.mit.edu). This author was supported in part by Army contract DAAH04-95-0607 and by ARPA contract N00014-95-1-1246.

‡ School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 (bmm@cs.cmu.edu). This author was supported in part by the Air Force Material Command (AFMC) and ARPA under contract F196828-93-C-0193, by ARPA contracts F33615-93-1-1330 and N00014-95-1-1246, and by an NSF National Young Investigator Award, CCR-94-57766, with matching funds provided by NEC Research Institute. This research of this author was conducted while the author was employed at NEC Research Institute.

§ Department of Computer Science, University of Massachusetts, Amherst, MA 01003 (ramesh@cs.umass.edu). This author was supported in part by NSF grant CCR–94–10077. The research of this author was conducted while the author was a student at Princeton University.

containing some faults is to be used for packet routing, we are concerned with how many nodes can send and receive packets, and how much longer it takes the faulty butterfly to deliver all of the packets than it takes a fault-free butterfly to perform the same task. More generally, we are interested in the length of time it takes an impaired network to emulate a single step of a fault-free network of the same type and size. 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 is to understand the relationship between slowdown and the number of faults for commonly used networks. In particular, we prove bounds on the number of faults that can be tolerated without losing more than a constant factor in speed.

We have two approaches for emulating a fault-free network G on an isomorphic faulty network H. The first approach is to find an embedding of G into H that avoids the faults in H. The second approach uses redundant computation; i.e., we allow H to emulate some of the nodes of G in more than one place. At first glance this approach seems disadvantageous, since H ends up performing more work. As we shall see, however, the freedom to emulate a node of G in more than once place allows H to have results ready where and when they are needed and can greatly reduce the slowdown of the emulation.

1.1. Emulations based on embeddings. An *embedding* maps the nodes of G to nonfaulty nodes of H and the edges of G to nonfaulty paths in H. A good embedding is one with minimum load, congestion, and dilation, where 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, and 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 [30] that if there is an embedding of G in H with load l, congestion c, and dilation d, then H can emulate any computation on G with slowdown $O(l + c + d)$.

In this paper, we are most interested in embeddings for which the load, congestion, and dilation are all constant (independent of the size of the network). In particular, we 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 load, congestion, and dilation. A similar result is also proved for the 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 (two- or three-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, congestion, or dilation [22, 25, 31], and
2. the N-node hypercube can be reconfigured around $\log^{O(1)} N$ worst-case faults with constant load, congestion, and dilation [2, 12].

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 toward 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 around more than $\log^{O(1)} N$ faults with constant load, congestion, and dilation using (not necessarily level-preserving) embedding techniques remains an interesting open question.

1.2. Fault-tolerant routing algorithms. In section 3, we shift our attention to the routing capabilities of hypercubic networks containing faults. The algorithms developed in this section are later used by the emulation schemes based on redundant computation. 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 $N - 6f$ 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 [6, 28], but no such results were known for butterflies or other hypercubic networks. A corollary of this result is that an N-input butterfly with $N/12$ worst-case faults 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 partition an N-input butterfly into connected 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 a $2N$-input $O(1)$-dilated Beneš network contains $N^{1-\epsilon}$ worst-case faults (for any $\epsilon > 0$), there is still a set of $2N - o(N)$ inputs I and a set of $2N - o(N)$ outputs O such that for any one-to-one mapping $\phi : I \mapsto O$ it is possible to route edge-disjoint paths from i to $\phi(i)$ for all $i \in I$. This result substantially improves upon previous algorithms for fault-tolerant circuit switching in Beneš networks [41, 49], which dealt with a constant number of faults by adding an extra stage to the network.

1.3. Emulations using redundant computation. 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 is 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 [19, 26, 38, 39, 40, 47].

The techniques developed in section 4 also have applications for hypercubes. For example, in section 4.4, we use them to show than an N-node hypercube with $N^{1-\epsilon}$ worst-case faults can emulate T steps of any normal algorithm [27] in $O(T + \log N)$ time. (The set of normal algorithms include FFT, bitonic sort, and other important ascend–descend algorithms.) Previously, such results were known only for hypercubes containing $\log^{O(1)} N$ faults [2, 12, 13]. 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 indepen-

dently 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 remains an interesting open question. Until recently, no results along these lines were known for the butterfly, unless routing is allowed through faulty nodes [5], which simplifies matters substantially. Tamaki [51] has recently discovered an emulation scheme with slowdown $O((\log\log N)^{8.2})$. He also introduced a class of bounded-degree networks called cube-connected arrays [52] 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.

1.4. 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, [2, 5, 9, 14, 23, 24, 25, 35, 44, 52] show how to reconfigure a network with faults so that it can emulate a fault-free network of the same type and size. A fault-tolerant area-universal network is presented in [53]. References [4, 10, 11, 16, 17] show how to design a network H that contains G as a subnetwork even if H contains some faults. Algorithms for routing messages around faults appear in [1, 6, 8, 15, 24, 25, 28, 34, 36, 41, 43, 44, 49]. The fault-tolerance of sorting networks is studied in [7, 32]. Finally, [12, 56, 57] show how to perform certain computations in hypercubes containing faults.

1.5. Network definitions. In this section, we review the structure of some of the networks that we study in this paper. In all of these networks, the edges are assumed to be undirected (or bidirectional).

An N-input $(\log N)$-dimensional *butterfly* network has $N(\log N + 1)$ nodes arranged in $(\log N) + 1$ levels.[1] An 8-input butterfly is shown in Figure 1.1. Each node in the butterfly has a distinct label (w, i), where i is the *level* of the node $(0 \le i \le \log N)$ and w is a $(\log N)$-bit binary number that denotes the *row* of the node. All edges connect pairs of nodes on adjacent levels. Each node (w, i) is connected by a *straight edge* to node $(w, i + 1)$, provided that $i < \log N$. In the figure, straight edges are drawn horizontally. Each node (w, i) is also connected by a *cross edge* to node $(w', i + 1)$, where w and w' differ only in the bit in position i, provided that $i < \log N$. (The most significant bit is in position 0, and the least significant is in position $(\log N) - 1$.) In the figure, cross edges are drawn diagonally. The nodes in level 0 are called the *inputs* of the butterfly, and those in level $\log N$ are called the *outputs*. Sometimes the input and output nodes in each row are assumed to be the same node. In this case, the butterfly has only $N \log N$ nodes. Our results hold whether or not the butterfly wraps around in this way.

In an N-node *hypercube*, each node is labeled with a distinct $(\log N)$-bit binary number. Two nodes in the hypercube are connected by an edge if and only if their labels differ in exactly one bit. The hypercube is the only network considered in this paper in which the degree of each node is not constant.

As in the N-node hypercube, each node in an N-node *shuffle-exchange* network is labeled with a distinct $(\log N)$-bit binary number. An 8-node shuffle-exchange

[1] Throughout this paper, log denotes the base-2 logarithm function, $\log_2$.

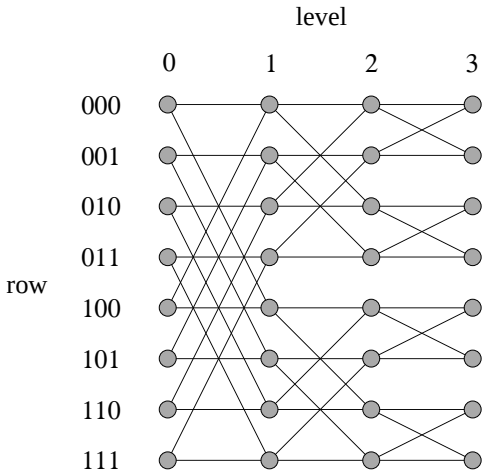

FIG. 1.1. *An* 8*-input butterfly network.*

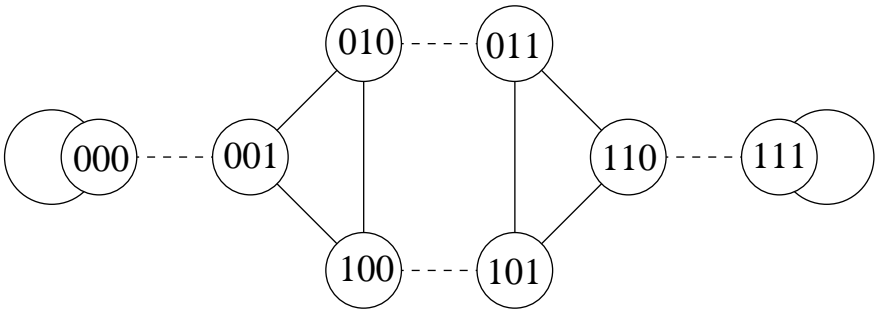

FIG. 1.2. *An* 8*-node shuffle-exchange network.*

network is shown in Figure 1.2. In the shuffle-exchange network, a node labeled u is connected by an *exchange* edge to the node labeled u', where u and u' differ only in the bit in the least significant position (position $(\log N)-1$). Node u is also connected by *shuffle* edges to the nodes labeled u_l and u_r, where u_l and u_r are the one-bit left and right cyclic shifts of u. (If $u_l = u_r$ then there is only one shuffle edge.) In the figure exchange edges are dotted and shuffle edges are solid.

An $N \times N$ *mesh of trees* network [27] is formed by first arranging N^2 nodes (but no edges) in a grid of N rows and N columns. Then for each row, an N-leaf complete binary tree, called a *row tree*, is added. The leaves of the row tree are the nodes of the corresponding row. Similarly, for each column an N-leaf *column tree* is added. The leaves of the column tree are the nodes of the column. Hence, the node at position (i, j) in the grid is a leaf of the ith row tree and jth column tree for $0 \le i, j \le N-1$.

A *circuit-switching network* is used to establish edge-disjoint paths (called circuits) between its inputs and outputs. We call the nodes in a circuit-switching network *switches* to signify that they are used only for routing and not for performing computation. Each switch in a circuit-switching network has a set of incoming edges and a set of outgoing edges. Inside the switch, the incoming edges can be connected to the outgoing edges in any one-to-one fashion. The switches at the first level of the network are called the *input switches*. The switches at the last level are called the

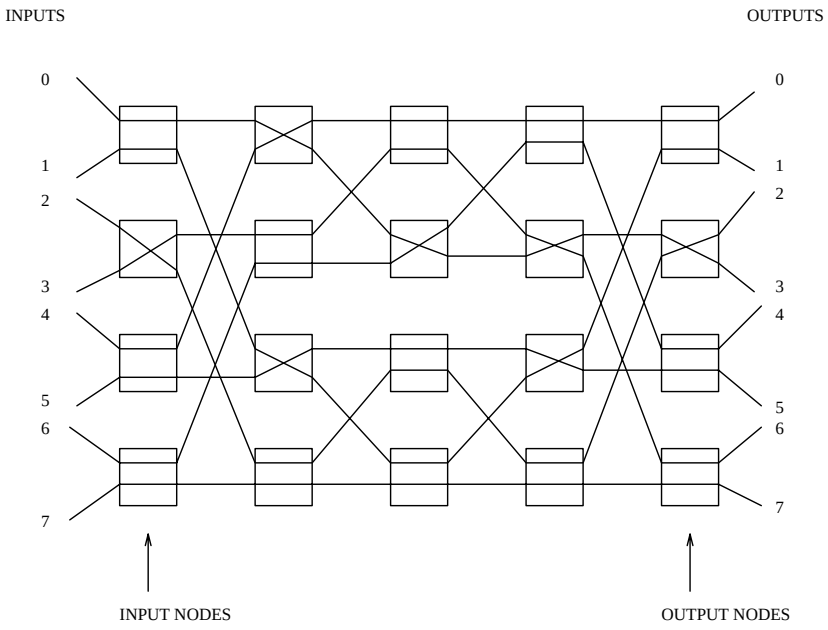

FIG. 1.3. *An 8-input (two-dimensional) Beneš network.*

output switches. The edges into each input switch are called *input edges*, or *inputs*. The edges out of each output switch are called *output edges*, or *outputs*. (Note, however, that in a butterfly network, we use the terms "inputs" and "outputs" to refer to nodes, not edges.) By setting the connections inside 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 connecting the ith input to the $\phi(i)$th output for $0 \leq i \leq N-1$.

The *Beneš* network is a classic example of a rearrangeable network. 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. Hence a Beneš network consists of two back-to-back butterflies sharing level $\log N$. We refer to the switches in levels 0, $\log N$, and, $2\log N$ as the *input switches*, *middle switches*, and *output switches*, respectively. Figure 1.3 shows an 8-input Beneš network in which the inputs are connected to the outputs according to the following mapping ϕ: $\phi(0) = 3$, $\phi(1) = 1$, $\phi(2) = 2$, $\phi(3) = 6$, $\phi(4) = 0$, $\phi(5) = 5$, $\phi(6) = 4$, $\phi(7) = 7$.

2. Emulation by embedding. In this section, we show how to embed a fault-free binary tree, butterfly, or mesh of trees into a faulty network of the same type and size 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 arbitrary computations of a fault-free network on a faulty network of the same type and size with only constant slowdown. We first consider embedding a complete binary tree in a complete binary tree with faults only at its leaves. This result also holds for fat-trees [21, 33] 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 main result of this section is a proof that an N-node butterfly or mesh of trees network can tolerate $\log^{O(1)} N$ worst-case faults and still emulate a fault-free network of the same type and size with only constant slowdown.

2.1. The binary tree. In Theorem 2.1.2, we show that a fault-free 2^n-leaf complete binary tree can be embedded in another 2^n-leaf complete binary tree containing $S(n,b)$ or fewer faults at its leaves with load and congestion at most 2^b and dilation 1, where $S(n,b)$ is defined 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 $n \geq 0$. The following lemma provides a useful bound on the growth of $S(n,b)$.

LEMMA 2.1.1. *For all* $n \geq b \geq 1$, $\binom{n}{b} \leq S(n,b) \leq \binom{n+b}{b}$.

Proof. The proof is by induction on n. For $n = 1$, the only possible value of b is 1. In this case, $S(1,1) = 1$ and $\binom{1}{1} = 1 < \binom{2}{1}$. For $n > 1$, there are three cases to consider. First, for $b = 1$, $S(n,1) = n$, and $\binom{n}{1} = n < \binom{n+1}{1}$. Second, for $n = b$, $S(n,n) = 2^n - 1$ and $\binom{n}{n} \leq 2^n - 1 < \binom{2n}{n}$. Finally, for $n > b > 1$, the inequalities are proved inductively using the fact that $\binom{x}{y} = \binom{x-1}{y} + \binom{x-1}{y-1}$, for all $x > y > 0$, and $\binom{x}{y} + 1 \leq \binom{x+1}{y}$ for $x \geq y > 0$. ∎

Using the inequalities $(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$ for $x \geq y > 0$, we see that for any constant $b > 0$, $S(n,b) = \Theta(n^b)$.

THEOREM 2.1.2. *Given a* 2^n*-leaf complete binary tree* T *with a set of at most* $S(n,b)$ *worst-case faults at the leaves, where* $n \geq b \geq 0$*, it is possible to embed a fault-free* 2^n*-leaf complete binary tree* T' *in* T *so that*

1. *nodes on level* i *of* T' *are mapped to nonfaulty nodes on level* i *of* T*, for* $0 \leq i \leq n$;
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. For $n = 0$, the only possible value of b is 0 and $S(0,0) = 0$. In this case T is a single fault-free node and T' can be embedded in T with load 1, congestion 0, and dilation 0. For $n > 0$, there are three cases to consider. First, for $b = 0$, $S(n,0) = 0$, so T has no faults. In this case, T' can be embedded in T with load 1, congestion 1, and dilation 1. Second, for $b = n$, $S(n,n) = 2^n - 1$, so there is a single nonfaulty leaf l in T. In this case, all of the 2^n leaves of T' are mapped to l, and the rest of the tree is mapped to the path from the root of T to l. The embedding has load 2^n, congestion 2^n, and dilation 1. Finally, suppose that $n > b > 0$. 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. Otherwise, if one 2^{n-1}-leaf subtree (say the left subtree) has $S(n-1,b)+1$ or more faults, then by the 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 with 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. ∎

Rewriting the number of leaves as $N = 2^n$, we see that for any constant $b > 0$, it is possible to embed a fault-free N-leaf complete binary tree T' in an N-leaf

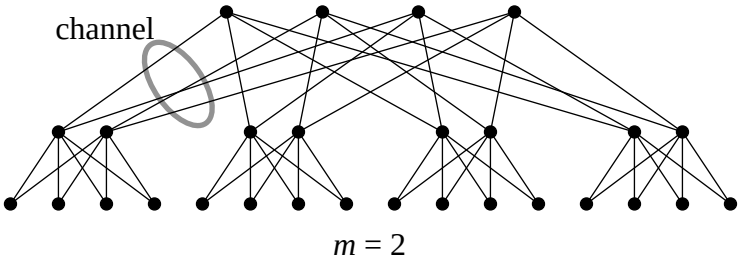

FIG. 2.1. *A 4-ary fat-tree network with depth $m = 2$ in which $r_0 = 4$, $r_1 = 2$, and $r_2 = 1$.*

complete binary tree T containing $S(\log N, b) = \Theta(\log^b N)$ faults with constant load, congestion, and dilation.

This result can be extended to a class of networks called *fat-trees*. A fat-tree of depth m is specified by a sequence of numbers $r_0, r_1, \ldots, r_m$, where $r_m = 1$. (Typically $r_0 \geq r_1 \geq \cdots \geq r_m$.) 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 m is constructed as follows. At the root of the fat-tree there is a set of r_0 nodes. The subtrees of the root are identical and are constructed recursively. Each is a fat-tree of depth $m - 1$ with number sequence $r_1, \ldots, r_m$. The r_0 root nodes of the fat-tree are connected to the r_1 root nodes of each subtree by a *channel* of edges. There may be any number of edges in the channel, and they may form any pattern of connections, but the channels to each subtree must be isomorphic. Figure 2.1 shows a fat-tree in which $r_0 = 4$, $r_1 = 2$, and $r_2 = 1$. In this figure, the root has four subtrees, as do the roots of these subtrees. Hence the figure shows a 4-ary fat-tree. This fat-tree was chosen for the figure because a fat-tree of this form has been shown to be area universal [33, 21, 30]. Corollary 2.1.3 is stated for binary fat-trees (i.e., fat-trees in which the root has two subtrees), but similar results can be proven for 4-ary fat-trees.

COROLLARY 2.1.3. *A $(\log N)$-depth binary fat-tree can be embedded 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 the 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 distinct leaf of the complete binary tree. Given a fat-tree F with some faulty leaves, let T be a complete binary tree whose leaf is faulty if and only if 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 binary fat-tree of depth $\log N$ with $r_i = 2^{\log N - i}$. 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 Theorem 2.1.2 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, which we call an *external product graph*.

As external product graph is defined as follows. Let G be a graph in which some set of N nodes have been labeled as *external* nodes. For example, if G is a tree, the leaves could be the external nodes, or, if G is a butterfly, the nodes on level $\log N$ (the outputs) could be the external nodes. Given a graph G with N external nodes, the external product graph of G (denoted PG) is constructed as follows. Make $2N$ copies of G, $G_{i,j}$ for $i = 1, 2$ and $0 \leq j \leq N-1$. Number the external nodes of each copy from 0 to $N-1$. Now identify the kth external node in $G_{1,j}$ with the jth external node in $G_{2,k}$ for all $0 \leq j, k \leq N-1$. (By "identify" we mean make them the same node of the graph PG.) The resulting graph is the external product graph of G. As an example, when the graph G is a tree and its leaves are the external nodes, the graph PG is a mesh of trees network. As another example, when G is a butterfly and its outputs are external nodes, PG is a butterfly with twice the dimension.

We now show that if G can tolerate faults in its external nodes, then PG can tolerate faults at any of its nodes.

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* lc*, 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 up of graphs isomorphic to G called $G_{i,j}$ 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 jth external node of CG is declared to be faulty if and only if 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 an external node of PG can appear both as the kth leaf of $G_{1,j}$ and as the jth leaf of $G_{2,k}$). 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 jth 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 to map the individual nodes of $G'_{i,j}$ to $G_{i,\phi(j)}$ for $i = 1, 2$ and $0 \leq j \leq N-1$. (Hence, the mapping Φ is used twice.) 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 our mapping is well defined, i.e., that it doesn't map an external node of PG' to more than one node of PG. The kth external node of $G'_{1,j}$ is the same as the jth 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. Hence, the mapping is well defined. 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 map l nodes onto any particular node of $G_{i,\phi(j)}$. Therefore, the load is at most l^2, and the congestion is at most lc. □

Theorem 2.2.1 is readily applied to the butterfly and mesh of trees networks. For simplicity, we state the result for a two-dimensional mesh of trees. The same techniques, however, can be used to show that any constant-dimension mesh of trees can tolerate $\log^{\Theta(1)} N$ worst-case faults with only 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 follows from Corollary 2.1.4 and Theorem 2.2.1. □

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. □

The results of this subsection can also be shown by using the fact that the butterfly and the mesh of trees can be expressed as the layered cross product [18] of two complete binary trees (or variations thereof) [3] and proving a theorem analogous to Theorem 2.2.1 for layered cross product graphs.

2.3. Limitations of level-preserving embeddings. We do not know whether 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. The proof uses a construction called an arrow diagram.

Given an N-leaf binary tree T with faults at its leaves, an *arrow diagram* has arrows drawn from some nodes of T to their siblings, with no pairs of antiparallel arrows allowed. 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 (called an outgoing arrow).
2. On any path with no outgoing arrow, there can be at most b incoming arrows.

An arrow diagram is called *legal* if it is b-legal for any $0 \leq b \leq n$.

Suppose that an adversary is allowed to place faults at the leaves of a 2^n-leaf complete binary tree. Let $T(n, b) + 1$ be the minimum number of faults needed by the adversary to make it impossible to construct a b-legal arrow diagram for the tree. Note that if a diagram is illegal for some set of faults then it cannot be made legal by adding another fault. Hence allowing more faults only makes the adversary more powerful. We bound the value of $T(n, b)$ as follows.

LEMMA 2.3.1. *For $n \geq b \geq 0$,*

$$T(n,b) \leq \begin{cases} 0 & \text{for } b = 0, \\ T(n-1,b) + T(n-1,b-1) + 1 & \text{for } 0 < b < n, \\ 2^n - 1 & \text{for } b = n. \end{cases}$$

Proof. First, suppose that $b = 0$. If the tree has one or more faults, then any legal arrow diagram must have at least one arrow. If the diagram has at least one arrow, then there must be a 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 that is closest to the root of the tree, and let this arrow be directed from a node m' to its sibling m. 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 suffices.) Thus, a tree with a fault cannot have a 0-legal arrow diagram. Thus $T(n, 0) \leq 0$.

Next, let $n = b$. If there are 2^n faults, then every leaf is faulty, and it is not possible to draw an arrow diagram with an outgoing arrow on the path from the root to every faulty leaf. Thus, $T(n, n) \leq 2^n - 1$.

Finally, suppose that $0 < b < n$. We show that there is a way of placing $T(n-1, b) + T(n-1, b-1) + 2$ faults at the leaves such that in any legal arrow diagram either there must be at least $b+1$ incoming arrows on some path without any outgoing arrow 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 that 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 that 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 would be no outgoing arrow for the path from the root of T to this fault. 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 would 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. □

COROLLARY 2.3.2. *For all $n \geq b \geq 1$, $T(n, b) = O(n^b)$.*

Proof. The recurrence for $T(n, b)$ is bounded from above by the recurrence that we had for $S(n, b)$ in section 2.1 and hence $T(n, b) = O(S(n, b))$. Using Lemma 2.1.1, $T(n, b)$ is $O(n^b)$. □

THEOREM 2.3.3. *For any constants l and d, there is a constant $k = d + (l-1)2^d$ such that there is a way of placing $O(\log^k N)$ 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 load l and dilation d.*

Proof. We begin by placing a set of faults of cardinality $O(\log^k N)$ at the leaves of T such that this fault pattern has no k-legal arrow diagram, where $k = d + (l-1)2^d$ and $N = 2^n$. This is possible because $T(n, k) + 1$ is $O(\log^k N)$. Now suppose for the sake of contradiction that there is an embedding of T' into T with load l and dilation d with the property that nodes on level i of T' are mapped to nodes on level i of T and no nodes of T' are mapped to faulty nodes of T. 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 larger 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 now show that the annotated tree is b-legal for some $b \leq k$, which is a contradiction. 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. Hence, the first criterion in the definition of a b-legal tree is satisfied. Let b be the maximum number of incoming arrows on a path without an outgoing arrow. We ignore the last d levels of the tree. Therefore, there is a path in T, $m_0, m_1, \ldots, m_{n-d}$, where m_0 is the root and m_i is a node in level i of the tree, that 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_i, then the split of leaves of T' is even and hence $l_i = l_{i-1}$. Suppose there is an incoming arrow into node m_i from its sibling m'_i. Then $l_i > l_{i-1}$. Further, $l_i \geq l_{i-1} + 2^{-d}$. To see why, consider the subtrees of T' rooted at level $i+d$. The nodes in each of these subtrees can be mapped entirely within either the subtree rooted at m_i or entirely within the subtree rooted at m'_i but never to the nodes in both. To see why, note that if a node in one of these subtrees that was mapped to the subtree rooted at m_i and a neighbor at an adjacent level was mapped to the subtree rooted at m'_i, then the dilation of the edge between them would be more than d. (In fact, the dilation would have to be at least $2d+3$, since the subtrees are separated by a distance of $2d+2$, and any two nodes connected by an edge in T' must be mapped to different levels in T.) Thus, the subtree rooted at m_i must have at least 2^{n-i-d} more leaves of T' mapped to it than the subtree rooted at m'_i. Hence, $l_i \geq l_{i-1} + (2^{n-i-d}/2^{n-i}) = l_{i-1} + 2^{-d}$. Since there are at least $b-d$ incoming arrows on the path, $l_{n-d} \geq 1 + (b-d)2^{-d}$. Note that there is at least one leaf in the subtree rooted at m_{n-d} that has load at least l_{n-d}. Therefore, $l \geq l_{n-d}$. This implies that $b \leq d + (l-1)2^d = k$. But there can be no k-legal arrow placement for the fault pattern chosen for T. This is a contradiction. □

THEOREM 2.3.4. *For any constants l and d, there is a constant $k = d + (l-1)2^d$ 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 load l and dilation d.*

Proof. The proof is similar to that of Theorem 2.3.3. Let B be a butterfly with faults and let 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 subbutterflies 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 distinct node in level $\log N$ of the butterfly B. We choose the same set of worst-case faults in the leaves of T as in Theorem 2.3.3. The faulty nodes of B are the nodes in level $\log N$ of B that correspond to the faulty leaves of T. Given a level-preserving embedding of B' into B with load l 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 subbutterfly of B represented by tree node m than the subbutterfly represented by tree node m'. Let m and m' be on level j of T. As before, due to dilation considerations, the smaller subbutterflies of B' spanning levels $j+d$ to n must be mapped entirely within the subbutterfly of B represented by m or within the subbutterfly represented by m' but never to both. The rest of the proof is similar to Theorem 2.3.3. □

The following theorem is stated for two-dimensional meshes of trees. An analogous theorem can be proved for any constant-dimension mesh of trees.

THEOREM 2.3.5. *For any constants l and d, there is a constant $k = d + (l-1)2^d$ such that there is a way of choosing $\Theta(\log^k N)$ faults in a $\sqrt{N} \times \sqrt{N}$ mesh of trees M such that there is no level-preserving embedding of a $\sqrt{N} \times \sqrt{N}$ mesh of trees M' in M with load l and dilation d.*

Proof. The proof is similar to that of Theorem 2.3.4. Let M be a mesh of trees with faults and let 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 two-dimensional $\sqrt{N} \times \sqrt{N}$ mesh. We 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 vertically or horizontally, until reaching individual mesh nodes, which are each represented by a distinct leaf of the tree. We choose the same set of worst-case faults in the leaves of T as in Theorem 2.3.3. The faulty nodes of M are the mesh nodes of M that correspond to the faulty leaves of T. Given a level-preserving embedding of M' into M with load l 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 there are more mesh nodes of M' mapped to the submesh of M represented by tree node m than the submesh represented by tree node m'. As before, due to dilation considerations, the smaller submeshes of M' must be mapped entirely within the submesh of B represented by m or within the submesh represented by m' but never to both. The rest of the proof is similar to Theorem 2.3.3. □

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 a butterfly network with faulty nodes, while section 3.2 presents algorithms for establishing edge-disjoint paths between the inputs and outputs of an $O(1)$-dilated 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. In particular, we focus on the problem of routing packets between the nodes of the network in a one-to-one fashion. This type of routing is also called *permutation routing.* (See [37] for references to permutation routing algorithms.) In a permutation routing problem, every node is the origin of at most one packet and the destination of at most one packet. We show that there is some set of $N - 6f$ rows (where $0 \leq f \leq N/6$) such that it is possible to route any permutation between the nodes in these rows in $O(\log N)$ steps using constant-size queues, with high probability. The same result (without the high probability caveat) was previously shown for the expander-based multibutterfly network [28]. A special case of this result is that when $f \leq N/12$ we can route arbitrary permutations between a majority of nodes in the butterfly. Note that this is optimal to within constant factors since N faults on level $(\log N)/2$ partitions the butterfly into many disjoint small connected components.

It will be convenient for us to view the packets as being routed on a larger network with $4 \log N + 1$ levels and N rows. The network consists of four stages. Stage i consists of those nodes in levels $i \log N$ through $(i + 1) \log N$ for $0 \leq i \leq 3$. Note that each pair of consecutive stages shares a level of nodes. The nodes in stages 0 and 3 are connected by straight edges only. Stages 1 and 2 consist of a pair of back-to-back butterflies isomorphic to the Beneš network. In analogy with the Beneš network, the nodes in levels $\log N$, $2 \log N$, and $3 \log N$ are called input nodes, middle nodes, and output nodes, respectively. Note that this larger network can be embedded in a butterfly network so that the jth row of the larger network is mapped to the jth row of the butterfly, and at most one node from each stage of the larger network is mapped to each node of the butterfly. The embedding has load 4, congestion 4, and dilation 1.

We start by describing Valiant's algorithm [54] for permutation routing on a butterfly without faults. Each node in stage 0 is the source of at most one packet, and each node in stage 3 is the destination of at most one packet. (Thus in the underlying butterfly, each node is both the source and destination of at most one packet.) In

stage 0, a packet travels along 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 the output node m' in the row of its destination. In stage 3, the packet travels along the row of m' until it reaches 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 networks such as the butterfly with $O(\log N)$ levels, as long as the (leveled) paths of the packets have congestion $O(\log N)$, a Ranade-type queuing protocol can be used to route the packets in $O(\log N)$ steps using constant-size queues, with high probability [29]. Therefore, it is sufficient to derive high-probability bounds on the congestion of the paths in a routing scheme.

Our goal is to identify a large set of "good" nodes in a faulty butterfly between which we can route permutations using an algorithm like Valiant's. A node in the four-stage network is faulty if the corresponding node in the underlying butterfly is faulty. 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 every input node m, let $REACH(m)$ be defined to be the set of middle nodes reachable from m using fault-free paths of length $\log N$ from level $\log N$ to level $2 \log N$. Also, for every output node m', let $REACH(m')$ be the set of middle nodes reachable using fault-free paths from level $3 \log N$ back to level $2 \log N$. Note that if m and m' lie in the same row, then $REACH(m) = REACH(m')$, because the fault pattern in stage 2 is the mirror image of the fault pattern in stage 1. If $|REACH(m)| < 4N/5$ for any input node m, then we declare m and all other nodes in its row to be bad. Any node not declared bad is considered *good*. Note that there are no faults in rows containing good nodes, and every good input or output node can reach at least $4N/5$ middle nodes via fault-free paths. A row in the underlying butterfly is good if the corresponding row in the four-stage network is good.

We now show that only $6f$ 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 $5f$ 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

$$\sum_{i=0}^{N-1} n_i \geq \left(\frac{fN}{t} + 1 \right) (t + 1).$$

A fault at any level of the butterfly lies on precisely N paths from input nodes to output nodes. Hence $\sum_{0=1}^{N-1} n_i = fN$. Combining the equation with the inequality yields $fN \geq (fN/t + 1)(t + 1)$, which is a 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, in order to route a packet from input m to output m', we randomly select a middle node m'' from $REACH(m) \cap REACH(m')$. (One way to do this is to store at each input m a table containing information about $REACH(m) \cap REACH(m')$ for each output m'.) Since m and m' are good input and output nodes $|REACH(m) \cap REACH(m')|$ is at least $3N/5$. We now prove that the paths selected in this manner have congestion $O(\log N)$ with high probability.

LEMMA 3.1.2. *For any constant $k > 0$, the randomly selected paths have congestion $O(\log N)$ with probability at least $1 - 1/N^k$. Furthermore, these paths are leveled and have length at most $4 \log N$.*

Proof. The lengths of the paths are clearly at most $4 \log N$ 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. Thus, the congestion of any node in stages 0 and 3 is trivially at most $(\log N + 1)$. Consider a node s in level l of the butterfly in stage 1. There are $2^l(\log N + 1)$ packets that could pass through this node. A packet passes through this node if and only if 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, $|REACH(m)| \geq 4N/5$ and $|REACH(m')| \geq 4N/5$. This implies that $|REACH(m) \cap REACH(m')| \geq 3N/5$. Thus the probability that the packet chooses a middle node that is reachable from s is at most $2^{\log N - l}/(3N/5)$. Therefore, the expected number of packets that passes through a node in stage 1 is at most $(2^l(\log N + 1))(2^{\log N - l})/(3N/5) = 5(\log N + 1)/3$. We can use Chernoff-type bounds [46] to show that the number of packets that pass through s in stage 1 is $O(\log N)$ with probability at least $1 - 1/2N^k$ for any constant k. The calculation for a node in stage 2 is exactly analogous. Thus the congestion is $O(\log N)$ with probability at least $1 - 1/N^k$. □

The following theorem summarizes the result presented in this section.

THEOREM 3.1.3. *In an N-input butterfly network with f worst-case faults, where $0 \leq f \leq N/6$, there is some set of $N - 6f$ "good" rows whose nodes can serve as the origins and destinations of any permutation routing problem. Furthermore, there is a routing algorithm such that, for any constant $k > 0$, there is a constant $C > 0$ such that the algorithm routes all the packets in any permutation (using routing tables) in at most in $C \log N$ steps using constant-size queues, with probability at least $1 - 1/N^k$.*

Proof. Lemma 3.1.1 shows how to identify the $N - 6f$ rows that are to serve as the sources and destinations of the packets. The lemma is applied for the case $t = N/5$. To route from an input node m to an output node m', a packet must select a random intermediate destination m'' that can be reached from both m and m'. The choice is made by consulting a table of all such m''. Lemma 3.1.2 shows that, with high probability, these paths have congestion $O(\log N)$. Finally, once the paths are selected, the algorithm for routing on leveled networks [29] can be applied to deliver the packets in $O(\log N)$ steps, with high probability using constant-size queues. □

3.1.1. Packet routing without routing tables. In the previous algorithm, every good input node m was required to store a table containing information about $REACH(m) \cap REACH(m')$ for every good output node m'. In this section, we show that is possible to route packets in a faulty butterfly without using such routing tables. The information about the placement of the faults is used only during the reconfiguration when the good and bad nodes are identified. This information is not needed for the routing itself. We assume that any packet that attempts to go through a fault is simply lost. We further assume that a node that receives a packet sends back an acknowledgement message (ACK) to the sender. Each ACK message follows the path of the corresponding packet in reverse. An ACK is generally smaller than a message and requires at most $O(\log N)$ bits to specify its path and the identity of the message that it is acknowledging. The algorithm for routing proceeds in rounds. There are a total of $(A \log \log N) + 1$ rounds (for some constant A). Each round consists of the following steps (R takes values from 0 to $A \log \log N$ 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 but in reverse.

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

THEOREM 3.1.4. *For any constant $k > 0$, there are constants A and B such that the algorithm routes any permutation on the nodes of the $N - 6f$ good rows of an N-input butterfly with f faults in $O(\log N \log\log N)$ steps using constant-size queues, with probability at least $1 - \frac{1}{N^k}$, without using routing tables.*

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 m and m' denote the input and output nodes that the packet passes through, respectively. In stage 1, if the packet chooses any middle node in $REACH(m) \cap REACH(m')$ then it succeeds in reaching its destination. Since $|REACH(m) \cap REACH(m')| \geq 3N/5$ the probability of this happening is at least $3/5$. Suppose the packet did not get through after $A \log\log N$ rounds. Then $2^{A \log\log N} = \log^A N$ copies of the packet are 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 $1/N^{k+3}$, for any constant $k > 0$ and for an appropriate choice of the constant A. Thus, the probability that some packet does not reach its destination is at most $N \log N/N^{k+3}$, which is less than $1/N^{k+1}$.

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 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 expected number of copies that are sent from a row in stage 0 or that are destined for a row in stage 4 that do 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., at most $1/AN^{k+2}$, for any constants A and k, using Chernoff-type bounds. At the beginning of the next stage, each unsent copy is duplicated and hence, with high probability, the number of packets in any row in the next round is at most $q \log N$. Since there are $(A \log\log N)+1$ rounds, the probability that the inductive hypothesis does not hold in the beginning of any one round is at most $((A \log\log N) + 1)/AN^{k+2}$, which is less than $1/N^{k+1}$.

Now we assume that the inductive hypothesis is true and show that each round takes only $B \log N$ steps, for some constant B, 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 2^l input nodes, each packet with probability 2^{-l}. The total number of packets that pass through an input node is at most $q \log N$ by our inductive hypothesis. Therefore, the expected number of packets that pass through a node at level l is $q \log N 2^l 2^{-l} = q \log N$. 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 at most $1/AN^{k+2}$, using Chernoff-type 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 the algorithm for routing on leveled networks [29] to schedule the packets, the routing completes in $C \log N$ steps with probability at least $1-2/AN^{k+2}$ for an appropriate constant C. The ACKs follow the paths of the packets in the reverse direction. Therefore, the congestion in any node due to ACKs can be no larger than the congestion due to packets and is also therefore $O(\log N)$. Since we are using the algorithm for routing on leveled networks to schedule the packets, the probability that some ACK does not reach its destination in $D \log N$ time is also at most $2/AN^{k+2}$ 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 $4A \log \log N/AN^{k+2}$, which is less than $1/N^{k+1}$.

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 at most $3/N^{k+1}$, which is less than $1/N^k$. Thus, the algorithm successfully routes every packet to its destination in $O(\log N \log \log N)$ steps with probability at least $1 - 1/N^k$. □

If the number of worst-case faults is smaller then there is a simpler way of routing in $O(\log N)$ steps without using routing tables or creating duplicate packets.

THEOREM 3.1.5. *Given an N-input butterfly with $N^{1-\epsilon}$ worst-case faults (for any constant $\epsilon > 0$), it is possible to identify $N - o(N)$ good rows in the butterfly such that any permutation routing problem on the good nodes can be routed in $O(\log N)$ steps using constant-size queues with probability greater than $1 - 1/N^k$, for any fixed constant k, without using routing tables.*

Proof. We define the "good" nodes in the butterfly as follows. A row is good if it contains no faults and the input in that row can reach all but $N^{1-\epsilon/2}$ middle nodes. (Previously a good input was required to reach all but $N/5$ middle nodes.) Using Lemma 3.1.1 with $f = N^{1-\epsilon}$ and $t = N^{1-\epsilon/2}$, we see that the number of good rows is least $N - N^{1-\epsilon/2}$. The algorithm is the same as the previous algorithm except that we don't create any duplicate packets and we now need only a constant number of routing rounds with high probability. This is because each unsent packet at each round has probability at most $\Theta(1/N^{\epsilon/2})$ of hitting a fault. Therefore, it is sufficient to have $\Theta(k/\epsilon)$ rounds (a constant) before every packet is delivered, with probability 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 main result of this section is a proof 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 $N - 4N^{1-\delta}$ input and output switches 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.) At each input switch, two of the b incoming edges are used as inputs, and at each output switch, two of the b outgoing edges are used as 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 that originate from input edges that share an input switch cannot use the same sub-Beneš network. The same is true for paths that end on output edges that share the same output switch. A full permutation consists of a set of $2N$ input–output pairs to be connected by edge-disjoint paths. The standard algorithm for setting the switches in a Beneš network, due to Waksman [55], uses bipartite graph matching to split the set of $2N$ pairs into two sets of N pairs which are then each routed recursively in one of the smaller sub-Beneš networks.

We now present Waksman's algorithm with a twist. We 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 switch (or output switch). 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. (See [42] for a nice proof that such a split is possible.) 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 opposite. 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 switches (switches 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. □

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 classify the input and output switches of the Beneš network as either good or bad depending on whether they can reach a sufficiently large number of middle switches. In a fault-free Beneš network, there is a path from each input (and output) switch to each of the N middle switches. The middle switches in fact form the leaves of a complete binary tree with the input (or output) switch as the root. The faults could make some of these paths unusable. We declare an input (or output) switch *bad* if the number of middle switches that it cannot reach exceeds a certain threshold. The threshold is 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 two inputs coming into an input (or output) switch are good or bad depending on whether the corresponding input (or output) switch is good or bad.)

Let $BAD(t)$ be the set of input and output switches for which more than t middle switches are unreachable. The first and last $\log N + 1$ levels of the Beneš network each

form a $\log N$-dimensional butterfly. Applying Lemma 3.1.1 to each of these butterflies separately, we know that $|BAD(t)| < 2fN/t$. (Note that these butterflies share the middle level of switches and hence might share some faults.)

THEOREM 3.2.2. *For any constants $0 < \delta < \epsilon \leq 1$, there exists a constant $b = \lceil 1 + (2-\epsilon)/(\epsilon - \delta) \rceil$ such that a $2N$-input b-dilated Beneš network with $N^{1-\epsilon}$ worst-case switch faults has a set of $N - 4N^{1-\delta}$ input switches and output switches between whose input and output edges it is possible to route any permutation using edge-disjoint paths.*

Proof. We declare any input or output switch in $BAD(N^{1+\delta-\epsilon}/2)$ to be bad. Since we need the number of good input switches and good output switches to be equal we may have to declare some extra input switches or output switches to be bad. From Lemma 3.1.1, we know that $|BAD(N^{1+\delta-\epsilon}/2)| \leq 4N^{1-\delta}$. Thus, the number of good input switches (or output switches) is at least $N - 4N^{1-\delta}$.

We now prove that we can route any permutation between the good inputs and good outputs using edge-disjoint paths. In this proof, we 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 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 appears on at most two paths. These paths are not permissible and must be rerouted in the second round using RANDSET. Note that every good input switch (or output switch) has at most $N^{1+\delta-\epsilon}/2$ unreachable middle switches. 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 hits 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 in the second round is at most $2N^{1-\epsilon}N^{-(\epsilon-\delta)}$. This implies that with nonzero probability RANDSET finds a set of paths such that at most $2N^{1-\epsilon-(\epsilon-\delta)}$ paths hit faults. Note that 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 nonzero probability RANDSET routes all of the paths without hitting any faults. In particular, such a set of paths exists. The expected number of paths that hit faults in the ith 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 bth round is less than 1. Therefore, all paths are routed by the end of the bth 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 obtain edge-disjoint paths for the permutation. □

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 finds them with nonzero probability. In this section we construct a deterministic algorithm that finds these paths using the technique due to Raghavan [45] and Spencer [50] to remove the randomness. Like Waksman's algorithm for finding the switch settings in a fault-free Beneš network with N input switches, the algorithm runs in $O(N \log N)$ time.

Let P be the random variable that denotes the number of paths that go through

faults at some round 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 upper bound on $E(P)$ that is defined as $U(P) = E(P_l) + E(P_r)$. In the proof of Theorem 3.2.2, we used the fact that with nonzero probability RANDSET finds a set of paths in which at most $U(P)$ paths hit faults. We define an algorithm DSET (for deterministic switch SETting) that deterministically finds 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 at most $U(P)$ paths hit faults, and DSET has the same asymptotic running time as RANDSET.*

Proof. We prove the theorem by induction on the size of the Beneš network. The base case is trivial. Now consider a $2N$-input Beneš network with a (partial) permutation ϕ to route. Let P_f and P_e denote the number of paths that hit faults in the first and last levels of the Beneš network, respectively. Also, let P_t and P_b denote the number of paths that hit faults in the top and bottom sub-Beneš networks (but not in the first or last levels of the Beneš network), let $P_{t,l}$ and $P_{t,r}$ denote the number of paths that hit faults in the left and right halves of the top sub-Beneš network, and let $P_{b,l}$ and $P_{b,r}$ denote the number of paths that hit faults in the left and right halves of the bottom sub-Beneš network. Finally, let i be the value chosen for X in step 2 of DSET. The total number of paths that hit faults P is bounded as follows:

$$P \le P_f + U(P_t|(X = i)) + U(P_b|(X = i)) + P_e \tag{3.1}$$

$$= P_f + E(P_{t,l}|(X = i)) + E(P_{t,r}|(X = i)) + E(P_{b,l}|(X = i)) + E(P_{b,r}|(X = i)) + P_e \tag{3.2}$$

$$= E(P_l|(X = i)) + E(P_r|(X = i)) \tag{3.3}$$

$$= U(P|(X = i)) \tag{3.4}$$

$$\le U(P). \tag{3.5}$$

These inequalities have the following explanation. Independent of the choice of i, P_f paths are blocked at the first level and P_e are blocked at the last. Once i is chosen, we know by induction that at most $U(P_t|(X = i))$ paths are blocked in the top sub-Beneš network and at most $U(P_b|(X = i))$ are blocked in the bottom sub-Beneš network. Hence inequality (3.1) holds. Equation (3.2) is derived by substituting the definitions of $U(P_t|(X = i))$ and $U(P_b|(X = i))$. Equation (3.3) is derived by observing that

$$E(P_l|(X = i)) = P_f + E(P_{t,l}|(X = i)) + E(P_{b,l}|(X = i))$$

and

$$E(P_r|(X = i)) = P_e + E(P_{t,r}|(X = i)) + E(P_{b,r}|(X = i)).$$

Equation (3.4) follows from the definition of $U(P|(X = i))$. Finally, (3.5) holds by the choice of i.

Now we deal with the question of how fast $U(P|(X = i))$ can be calculated in step 2 of DSET for $i \in \{0, 1\}$. For every switch m in the Beneš network, let $REACH(m)$ be the set of middle switches reachable from switch m using fault-free paths. We can precompute $|REACH(m)|$ as follows. The value for the middle switches 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 $|REACH(m)|$. 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 does 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. □

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 $(\log N)$-dimensional butterfly (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 both the guest and host 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) $\langle v, t\rangle$ that 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 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) $[e, t]$ that contains the message transmitted along edge e at time step t.

The host H emulates 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 of 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 first divide H into subbutterflies of dimension $(\epsilon \log N)/2$ spanning levels $(i\epsilon \log N)/2$ through $((i + 1)\epsilon \log N)/2$ for integer $i = 0$ to $2/\epsilon - 1$. (Without loss of generality, we assume that $2/\epsilon$ and $\epsilon \log N/4$ are integers.) Note that every input node of a subbutterfly is also an output node of another subbutterfly and vice versa. Each band of $((\epsilon \log N)/2) + 1$ levels consists of $N^{1-\epsilon/2}$ disjoint subbutterflies. Thus, there are a total of $(2/\epsilon)N^{1-\epsilon/2}$ subbutterflies. The faults in the network may make some of these subbutterflies unusable. We identify "good" and "bad" subbutterflies according to the following rules.

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

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 switches, the middle switches, and the output switches of the Beneš network are all 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 subbutterfly that contains a node that has a bad input/output at the end of its butterfly row is a bad subbutterfly.

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 $N^{1-\epsilon} + 4N^{1-2\epsilon/3}$.*

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 (they are the same nodes) is at most $4N^{1-\delta} = 4N^{1-2\epsilon/3}$. □

LEMMA 4.1.2. *For any $\epsilon > 0$, at least half the subbutterflies of H are good for sufficiently large N.*

Proof. The total number of subbutterflies is $(2/\epsilon)N^{1-\epsilon/2}$. By Lemma 4.1.1, the number of rows containing either a fault or a bad input or output is at most $N^{1-\epsilon} + 4N^{1-2\epsilon/3}$. Since each bad row passes through $2/\epsilon$ different subbutterflies, the total number of subbutterflies identified as bad by Rules 1 and 2 cannot exceed $2(N^{1-\epsilon}+4N^{1-2\epsilon/3})/\epsilon$. Observe that $2(N^{1-\epsilon}+4N^{1-2\epsilon/3})/\epsilon \leq N^{1-\epsilon/2}/\epsilon$ for sufficiently large N. □

Now we divide the guest G into overlapping subbutterflies of dimension $(\epsilon \log N)/2$ and map them to the good subbutterflies of H. For any node v of G, at most three nodes of H receive the initial state of the computation of node v, i.e., s-pebble $\langle v, 0\rangle$. (A node v of G can appear as an input in one subbutterfly, an output in another, and a middle node in a third.) These nodes in H create the s-pebbles for v. The mapping proceeds as follows. Take the guest G and cut it into subbutterflies at levels $i\epsilon \log N/2$ for integers $i = 0$ to $2/\epsilon - 1$. Map each subbutterfly to a good subbutterfly of H so that each subbutterfly of H gets at most two subbutterflies 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 subbutterflies to good subbutterflies of H as before. At most eight 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 level $i\epsilon \log N/4$, for $i = 0$ to $4/\epsilon - 1$, *boundary nodes* since they lie on the boundary of some subbutterfly 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 subbutterflies, i.e., level $i\epsilon \log N/2$ 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 (i.e., not on the boundary) of a good subbutterfly of H. Then the neighbors of m in H also create the s-pebbles of the neighbors of v in G. In this case m receives the required c-pebbles from 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 at least two nodes of H, there is another node m' of H that also creates an s-pebble for v. In particular, *m' is necessarily a node in the center of a subbutterfly of H*, i.e., in level $\epsilon \log N/4$ of the subbutterfly. Node m' of H forwards a copy of the c-pebble $[e, t]$ to node m for each of the edges e into v.

To facilitate the transmission of c-pebbles, we use the results of section 3.2 to establish constant-congestion fault-free paths in H 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 subbutterfly boundaries in that row, which is at most $8 \cdot 2/\epsilon = 16/\epsilon$, i.e., a constant. Similarly the number of paths ending in any row is $16/\epsilon$. We can divide the paths into $8/\epsilon$ sets such that each set has at most two paths originating in a row and two paths ending in a row. Note that all paths start and end in rows that have good inputs and outputs for doing Beneš-type routing. Therefore, each set can be routed with dilation $4 \log N$ and with congestion $O(1)$ 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 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 if and only if for every node v of G, an s-pebble $\langle v, T\rangle$ has been created somewhere in H. The emulation algorithm is executed by every node m of H and proceeds as a sequence of macrosteps. Each macrostep consists of the following four substeps.

1. *Computation step.* For each node v of G that has been assigned to m, if m contains an s-pebble $\langle v, t-1\rangle$ and c-pebbles $[e, t-1]$ for every edge e into v and m has not already created an s-pebble $\langle v, t\rangle$, then it does so.
2. *Communication step.* For each node v whose s-pebble was updated from $\langle v, t-1\rangle$ to $\langle v, t\rangle$ in the computation step and for each edge g from v to a neighbor u of v, node m sends a c-pebble $[g, t]$ to its neighbor in H that creates s-pebbles for u (if such a neighbor exists).
3. *Routing step.* If m has any c-pebble that was created on the previous copy step or that was received on the previous routing step but whose final destination is not m, then m forwards the c-pebble to the next node on the pebble's path to its destination.
4. *Copy step.* If m is a node in the center level (level $\epsilon \log N/4$ in the subbutterfly) and in the communication step m received a c-pebble $[e, t]$ for an edge e into a node v that has been assigned to m, then m makes two copies of the c-pebble, one for each of the two nodes m' and m'' that also create s-pebbles for v. On the next routing step, m forwards each copy to the next node on the path from m to m' or m''.

LEMMA 4.3.1. *Each macrostep takes only a constant number of time steps to execute.*

Proof. At most eight s-pebbles mapped are to each node. Therefore, the computation step takes constant time. Every s-pebble that is updated can cause at most four c-pebbles to be sent in the communication step. Therefore, the communication step takes only constant time. Since only a constant number of paths passes through any node, only a constant number of c-pebbles enters a particular path at any macrostep, and every c-pebble that has not yet reached its destination moves in every macrostep, the number of c-pebbles entering any node during any macrostep is at most a constant. Thus, the routing step and the copy step both take 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 time $O(T + \log N)$ by H.*

Proof. We show that only $O(T + \log N)$ macrosteps are required to emulate a T-step computation of G. The final result then follows from Lemma 4.3.1.

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 a c-pebble $[e, t-1]$ for each edge e into node v in G. Each c-pebble is sent by an s-pebble $\langle u, t-1\rangle$ where u is a neighbor of v in G. The dependency tree of $\langle v, t\rangle$ is defined recursively as follows. The root of the tree is $\langle v, t\rangle$ and the subtrees of the root are the dependency trees of $\langle v, t-1\rangle$ and all s-pebbles $\langle u, t-1\rangle$.

Let the emulation of T steps of G take T' macrosteps on H. Let $\langle v, T\rangle$ be an s-pebble that was updated in the last macrostep. We now look at the dependency tree of $\langle v, T\rangle$. We choose a *critical path*, $s_T, s_{T-1}, \ldots, s_0$, of tree nodes from the root to the leaves of the tree as follows. The first node on the path s_T is $\langle v, T\rangle$. 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 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 that 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. The last s-pebble on the path s_0 is one that was present initially, i.e., at time step 0. We 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. For every tree node s, we can associate a time (in macrosteps) $\tau(s)$ when that s-pebble was created. From the definition of our critical path and because a c-pebble moves once in every macrostep, $\tau(s_i) - \tau(s_{i-1}) = l_i$. Thus,

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

Now suppose that some l_i is greater than one. This corresponds to a long path taken by some c-pebble to go from $\phi(s_{i-1})$ in the center level of a subbutterfly of H to $\phi(s_i)$ in $\mathcal{B}_{\mathcal{H}}$. Thus l_i is the length of one of the constant congestion paths and is at most $(4 \log N)+1$. (In fact, the paths are somewhat shorter, but $(4 \log N)+1$ is a convenient quantity to work with.) 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 $1 \le j \le (\epsilon \log N)/4 - 1$. Thus, the extra path length of $4 \log N$ can be amortized over $(\epsilon \log N)/4$ s-pebbles. Hence, $T' = \sum_{0<i\le T} l_i \le (16/\epsilon + 1)T + 4 \log N + 1 = O(T + \log N)$. □

We can extend these results to the shuffle-exchange network using Schwabe's proof [26, 47] 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 that takes time T can be emulated in $O(T + \log N)$ time by an N-node shuffle-exchange network H with $N^{1-\epsilon}$ worst-case faults for any constant $\epsilon > 0$.*

Proof. Schwabe [26, 47] shows how to emulate any computation of a butterfly on a shuffle-exchange network with constant slowdown and vice versa. First we use Schwabe's result to map the computation of a butterfly B to the faulty shuffle-exchange network H. Any node of B that is mapped to a faulty node of H is declared

faulty. If there are any routing paths required by the emulation that pass through a faulty node of H, we declare the nodes of B that use this path to be faulty. The faulty nodes of B do no computation, and H is not required to emulate them. Hence, the faulty shuffle-exchange network H can emulate the faulty butterfly network B with constant slowdown. The number of faults in B is only a constant factor larger than $N^{1-\epsilon}$, since both the load and the congestion of the paths used in Schwabe's emulation are constant. Now we use Theorem 4.3.2 to emulate a fault-free butterfly B' on B. Finally, use Schwabe's result (in the other direction) 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 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* [13]). 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 including bitonic sort, FFT, and tree-based algorithms like branch-and-bound can be implemented on the hypercube as normal algorithms [27]. 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 [48] 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 also require the following well-known result concerning the embedding of a butterfly in a hypercube. (See [20] for the 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 T steps of any normal algorithm on an N-node fault-free hypercube in $O(T + \log N)$ steps.*

Proof. Let the faulty N-node hypercube be H and the fault-free N-node hypercube be G. H emulates any normal algorithm of G by using a sequence of constant slowdown emulations. Let an N-node butterfly B be embedded in H in the manner of Lemma 4.4.2. Any node of B that is mapped to a faulty node of H is considered faulty. (And H is not required to emulate these faulty nodes.) 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 with constant slowdown using the constant load, congestion, and dilation 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 emulations together, we obtain a constant slowdown emulation of any normal algorithm on G on the faulty hypercube H. □

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 [51].

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 subbutterflies. 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 $4(\log N/\log M_k) - 4$, there is a band of disjoint M_k-input subbutterflies spanning levels $(i \log M_k)/4$ through $((i+4)\log M_k)/4$. We call these subbutterflies the *band* i subbutterflies. Note that each band i subbutterfly shares $M_k^{3/4}$ rows with $M_k^{1/4}$ different band $i-1$ subbutterflies and $M_k^{3/4}$ rows with $M_k^{1/4}$ different band $i+1$ subbutterflies.

Each M_k-input subbutterfly in B_k is emulated by the corresponding subbutterfly in H. We say that an M_k-input subbutterfly in B_k *fails* if more than $\alpha\sqrt{M_k}\log M_k$ nodes inside the corresponding M_k-input subbutterfly in H fail, where α is a constant that will be determined later. If a subbutterfly in B_k fails, then H is not required to emulate any of the nodes that lie in that subbutterfly. As we shall see, if it does not fail, then the corresponding subbutterfly 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 subbutterfly in B_k fails.

LEMMA 5.1.1. *For $\alpha > 4e$, an M_k-input subbutterfly in B_k fails with probability at most $1/\log^{(k-1)} N$.*

Proof. An M_k-input subbutterfly fails if the corresponding M_k-input subbutterfly in H contains too many faults. An M_k-input subbutterfly in H contains a total of $M_k(1+\log M_k) \le 4(\log^{(k)} N)^2(\log^{(k+1)} N)$ nodes, each of which fails with probability $1/\log^{(k)} N = 1/\sqrt{M_k}$. Thus, the expected number of nodes that fail is at most $2\sqrt{M_k}\log M_k = 4(\log^{(k)} N)(\log^{(k+1)} N)$. Since each node fails independently, we can bound the probability that more than $\alpha\sqrt{M_k}\log M_k$ nodes fail using a Chernoff-type bound. For $\alpha > 4e$, 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 [46]). Since $\alpha\sqrt{M_k}\log M_k > \log^{(k)} N$, this probability is at most $2^{-\log^{(k)} N} = 1/\log^{(k-1)} N$. □

The next lemma shows that if a subbutterfly in B_k does not fail, then the corresponding subbutterfly in H can emulate it with constant slowdown.

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

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

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 subbutterflies in B_k that 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 subbutterfly in B_k that has not failed can be emulated by the corresponding subbutterfly 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 subbutterflies in H.) In order to emulate the entire network B_k, it is also necessary to emulate the connections between the subbutterflies. As in section 4, let $M_k^{1-\epsilon}$ denote the number of faults in an M_k-input subbutterfly of H. For a subbutterfly that has not failed, $\epsilon \approx 1/2$. By Lemma 4.1.1, the number of rows in the subbutterfly containing either a fault or an input or output that is bad for Beneš routing is at most $M_k^{1-\epsilon}/2 + 4M_k^{1-2\epsilon/3}$, which is approximately $4M_k^{2/3}$. Each band i subbutterfly that does not fail shares $M_k^{3/4}$ rows with each of the band $i-1$ subbutterflies (and band $i+1$ subbutterflies) with which it overlaps. Thus, for each pair of overlapping butterflies that do not fail, most of the shared rows are both fault-free and good for routing in both subbutterflies. In order to emulate an M_k-input subbutterfly of H, the emulation strategy of section 4 covers it with smaller subbutterflies, each having $M_k^{\epsilon/2}$ inputs. If a smaller subbutterfly is used in the emulation, then none of the rows that pass through it contain either a fault or a bad input or output. Thus, the $M_k^{3/4}$ connections between each pair of overlapping M_k-input subbutterflies in bands i 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 subbutterfly fails with probability $1/\log^{(k-1)} N$. A node in B_k fails if it lies in a subbutterfly that fails. Since each node in B_k lies in at most five $(\log^{(k)} N)^2$-input subbutterflies (once as an input, once as an output, and three times as an interior node), we have reduced the expected number of faults from $(N \log N)/\log^{(k)} N$ in H to fewer than $(5N \log N)/\log^{(k-1)} N$ 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 cover butterfly B_j with $(\log^{(j)} N)^2$-input subbutterflies. For ease of notation, let $M_j = (\log^{(j)} N)^2$. We say that an M_j-input subbutterfly in B_j fails if the corresponding M_j-input subbutterfly in B_{j+1} contains more than $\alpha\sqrt{M_j}$ M_{j+1}-input subbutterflies that have failed. The following three lemmas are analogous to Lemmas 5.1.1 through 5.1.3.

LEMMA 5.2.1. *For $\alpha > 8e$, the probability that an M_j-input subbutterfly 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 backward to $j = 0$. The base case is given by Lemma 5.1.1. For each value of i from 0 to $4(\log M_j/\log M_{j+1}) - 4$, there is a band of disjoint M_{j+1}-input subbutterflies 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 subbutterflies can be partitioned into eight *classes* according to their band numbers.

Two bands of subbutterflies belong to the same class if their band numbers differ by a multiple of eight. There are at most

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

subbutterflies in each of these classes, and within a class the subbutterflies are disjoint. By induction, each subbutterfly fails with probability at most $1/\log^{(j)} N$. Thus, in any particular class, the expected number of subbutterflies that fail is at most $(\log^{(j)} N)/2(\log^{(j+1)} N)(\log^{(j+2)} N)$, which is less than $\log^{(j)} N$. Using Chernoff-type bounds as in Lemma 5.1.1, for $\alpha > 16e$, the probability that more than $(\alpha \log^{(j)} N)/8 = (\alpha\sqrt{M_j})/8$ of these subbutterflies fail is at most $2^{-(\alpha \log^{(j)} N)/8}$, which is less than $1/8 \log^{(j-1)} N$. Thus, the probability that a total of $\alpha \log^{(j)} N$ subbutterflies fail in the eight classes is at most $1/\log^{(j-1)} N$. □

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

Proof. If an M_j-input subbutterfly does not fail, then at most $\alpha\sqrt{M_j}$ of the overlapping M_{j+1}-input subbutterflies in the corresponding M_j-input subbutterfly in H fail. Each of these subbutterflies contains $M_{j+1}(1+\log M_{j+1}) \leq 4(\log^{(j+1)} N)^2 \log^{(j+2)} N$ nodes. Since the total number of nodes in all of these subbutterflies is at most $4\alpha \log^{(j)} N(\log^{(j+1)} N)^2 \log^{(j+2)} N$, 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 $\epsilon \approx 1/2$. □

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 subbutterfly 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 subbutterflies 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 subbutterflies as if they were worst-case faults. In this case, the total number of worst-case faults is at most $4N^{1-\gamma}(\log N)^2 \log\log N$. 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 a butterfly tolerate random faults with constant failure probability and still emulate a fault-free butterfly of the same size 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 emulate a fault-free network of the same type and size with constant slowdown?
3. Can a $2N$-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?

Acknowledgments. 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

[1] G. B. Adams, III and H. J. Siegel, *The extra stage cube: A fault-tolerant interconnection network for supersystems*, IEEE Trans. Comput., C–31 (1982), pp. 443–454.

[2] W. Aiello and T. Leighton, *Coding theory, hypercube embeddings, and fault tolerance*, in Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, July 1991, pp. 125–136.

[3] W. A. Aiello, personal communication, July 1992.

[4] 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 Proc. 33rd Annual Symposium on Foundations of Computer Science, Oct. 1992, pp. 693–702.

[5] F. Annexstein, *Fault tolerance in hypercube-derivative networks*, Computer Architecture News, 19(1)(1991), pp. 25–34.

[6] S. Arora, T. Leighton, and B. Maggs, *On-line algorithms for path selection in a nonblocking network*, SIAM J. Comput., 25 (1996), pp. 600–625.

[7] S. Assaf and E. Upfal, *Fault-tolerant sorting network*, in Proc. 31st Annual Symposium on Foundations of Computer Science, Oct. 1990, pp. 275–284.

[8] Y. Aumann and M. Ben-Or, *Asymptotically optimal PRAM emulation on faulty hypercubes*, in Proc. 32nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, Oct. 1991, pp. 440–457.

[9] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, *Tolerating faults in synchronization networks*, Parallel Processing: CONPAR92-VAPPV, Lyon, France, Lecture Notes in Comput. Sci. 634, Springer, Berlin, 1992, pp. 1–12.

[10] J. Bruck, R. Cypher, and C.-T. Ho, *Fault-tolerant meshes with minimal numbers of spares*, IEEE Trans. Comput., 42 (1993), pp. 1089–1104.

[11] J. Bruck, R. Cypher, and C.-T. Ho, *Fault-tolerant meshes with small degree*, SIAM J. Comput., 26 (1997), pp. 1764–1784.

[12] J. Bruck, R. Cypher, and D. Soroker, *Running algorithms efficiently on faulty hypercubes*, Computer Architecture News, 19(1)(1991), pp. 89–96.

[13] J. Bruck, R. Cypher, and D. Soroker, *Tolerating faults in hypercubes using subcube partitioning*, IEEE Trans. Comput., 41 (1992), pp. 599–605.

[14] R. Cole, B. Maggs, and R. Sitaraman, *Reconfiguring arrays with faults part* I: *Worst-case faults*, SIAM J. Comput., 26 (1997), pp. 1581–1611.

[15] R. Cole, B. Maggs, and R. Sitaraman, *Routing on butterfly networks with random faults*, in Proc. 36th Annual Symposium on Foundations of Computer Science, Oct. 1995, pp. 558–570.

[16] S. Dutt and J. P. Hayes, *On designing and reconfiguring k-fault-tolerant tree architectures*, IEEE Trans. Comput., C–39 (1990), pp. 490–503.

[17] S. Dutt and J. P. Hayes, *Designing fault-tolerant systems using automorphisms*, J. Parallel and Distributed Computing, 12 (1991), pp. 249–268.

[18] S. Even and A. Litman, *Layered cross product—A technique to construct interconnection networks*, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, July 1992, pp. 60–69.

[19] M. R. Fellows, *Encoding Graphs in Graphs*, Ph.D. thesis, Department of Computer Science, University of California, San Diego, CA, 1985.

[20] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg, *Optimal embeddings of butterfly-like graphs in the hypercube*, Math. Systems Theory, 23 (1990), pp. 61–77.

[21] R. I. Greenberg and C. E. Leiserson, *Randomized routing on fat-trees*, in Randomness and Computation, Advances in Computing Research, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT, 1989, pp. 345–374.

[22] J. W. Greene and A. El Gamal, *Configuration of VLSI arrays in the presence of defects*, J. Assoc. Comput. Mach., 31 (1984), pp. 694–717.

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

[24] J. Hastad, T. Leighton, and M. Newman, *Fast computation using faulty hypercubes*, in Proc. 21st Annual ACM Symposium on Theory of Computing, May 1989, pp. 251–263.

[25] 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 Proc. 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, Oct. 1990, pp. 285–296.

[26] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg, *Work-preserving emulations of fixed-connection networks*, J. Assoc. Comput. Mach., 44 (1997), pp. 104–147.

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

[28] F. T. Leighton and B. M. Maggs, *Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks*, IEEE Trans. Comput., 41 (1992), pp. 578–587.

[29] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao, *Randomized routing and sorting on fixed-connection networks*, J. Algorithms, 17 (1994), pp. 157–205.

[30] F. T. Leighton, B. M. Maggs, and S. B. Rao, *Packet routing and job-shop scheduling in O(congestion + dilation) steps*, Combinatorica, 14 (1994), pp. 167–180.

[31] T. Leighton and C. E. Leiserson, *Wafer-scale integration of systolic arrays*, IEEE Trans. Comput., C–34 (1985), pp. 448–461.

[32] T. Leighton, Y. Ma, and C. G. Plaxton, *Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults*, J. Comput. System Sci., 54 (1997), pp. 265–304.

[33] C. E. Leiserson, *Fat-trees: Universal networks for hardware-efficient supercomputing*, IEEE Trans. Comput., C–34 (1985), pp. 892–901.

[34] G. Lin, *Fault tolerant planar communication networks*, in Proc. 24th Annual ACM Symposium on the Theory of Computing, May 1992, pp. 133–139.

[35] M. Livingston, Q. Stout, N. Graham, and F. Hararay, *Subcube Fault-Tolerance in Hypercubes*, Tech. report CRL-TR-12-87, University of Michigan Computing Research Laboratory, Ann Arbor, Sept. 1987.

[36] Y.-D. Lyuu, *Fast fault-tolerant parallel communication and on-line maintenance using information dispersal*, Math. Systems Theory, 24 (1991), pp. 273–294.

[37] B. M. Maggs and R. K. Sitaraman, *Simple algorithms for routing on butterfly networks with bounded queues*, in Proc. 24th Annual ACM Symposium on the Theory of Computing, May 1992, pp. 150–161; SIAM J. Comput., to appear.

[38] F. Meyer auf der Heide, *Efficiency of universal parallel computers*, Acta Inform., 19 (1983), pp. 269–296.

[39] F. Meyer auf der Heide, *Efficient simulations among several models of parallel computers*, SIAM J. Comput., 15 (1986), pp. 106–119.

[40] F. Meyer auf der Heide and R. Wanka, *Time-optimal simulations of networks by universal parallel computers*, in Proc. 6th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 349, Springer, Berlin, 1989, pp. 120–131.

[41] D. C. Opferman and N. T. Tsao-Wu, *On a class of rearrangeable switching networks–part* II*: Enumeration studies and fault diagnosis*, Bell System Tech. J., 50 (1971), pp. 1601–1618.

[42] N. Pippenger, *Telephone switching networks*, in Proc. Symposia in Applied Mathematics, Vol. 26, American Mathematical Society, Providence, RI, 1982, pp. 101–133.

[43] N. Pippenger and G. Lin, *Fault-tolerant circuit-switching networks*, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 229–235.

[44] M. O. Rabin, *Efficient dispersal of information for security, load balancing, and fault tolerance*, J. Assoc. Comput. Mach., 36 (1989), pp. 335–348.

[45] P. Raghavan, *Probabilistic construction of deterministic algorithms: Approximate packing integer programs*, J. Comput. System Sci., 37 (1988), pp. 130–143.

[46] P. Raghavan, *Lecture Notes on Randomized Algorithms*, Research Report RC 15340 (#68237), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, Jan. 1990.

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

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

[49] S. Sowrirajan and S. M. Reddy, *A design for fault-tolerant full connection networks*, in Proc. International Conference on Science and Systems, IEEE Computer Society Press, Piscataway, NJ, Mar. 1980, pp. 536–540.

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

[51] H. Tamaki, *Efficient self-embedding of butterfly networks with random faults*, in Proc. 33rd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Piscataway, NJ, Oct. 1992, pp. 533–541.

[52] H. Tamaki, *Robust bounded-degree networks with small diameters*, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 247–256.
[53] S. Toledo, *Competitive fault-tolerance in area-universal networks*, in Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 236–246.
[54] L. G. Valiant, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350–361.
[55] A. Waksman, *A permutation network*, J. Assoc. Comput. Mach., 15 (1968), pp. 159–163.
[56] A. Wang and R. Cypher, *Fault-Tolerant Embeddings of Rings, Meshes and Tori in Hypercubes*, Tech. report IBM RJ 8569, IBM Almaden Research Center, San Jose, CA, Jan. 1992.
[57] A. Wang, R. Cypher, and E. Mayr, *Embedding complete binary trees in faulty hypercubes*, in Proc. 3rd IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press, Piscataway, NJ, Dec. 1991, pp. 112–119.