A Parallel Algorithm for Reconfiguring a Multibutterfly Network with Faulty Switches

Andrew V. Goldberg[a]
Bruce M. Maggs[b]
Serge A. Plotkin[c]

Abstract– This paper describes a deterministic algorithm for reconfiguring a multibutterfly network with faulty switches. Unlike previous reconfiguration algorithms, the algorithm is performed entirely by the network, without the aid of any off-line computation, even though many of the switches may be faulty. The algorithm reconfigures an N-input multibutterfly network in $O(\log N)$ time. After reconfiguratuion, the multibutterfly can tolerate f worst-case faults and still route any permutation between some set of $N - O(f)$ inputs and $N - O(f)$ outputs in $O(\log N)$ time.

1 Introduction

Recently Leighton and Maggs showed that a multibutterfly network can sustain many faults and still route packets efficiently [6]. In particular, they showed that an N-input multibutterfly network can tolerate f worst-case faults, and still route any $\log N$ permutations[1] from some set of $N - O(f)$ inputs to some set of $N - O(f)$ outputs in $O(\log N)$ time. In the case of random faults, the performance is even better. Even if every switch fails

[a]Department of Computer Science, Stanford University, Stanford, CA 94305. Research supported in part by ONR Young Investigator Award N00014-91-J-1855, by NSF Presidential Young Investigator Grant CCR-8858097 with matching funds from AT&T and DEC, Stanford University Office of Technology Licensing, by a grant form Mitsubishi Corp, and by NEC Research Institute.

[b]NEC Research Institute, Princeton NJ 08540.

[c]Department of Computer Science, Stanford University, Stanford, CA 94305. Research supported by NSF Research Initiation Award CCR-900-8226, by U.S. Army Research Office Grant DAAL-03-91-G-0102, by a grant from Mitsubishi Electric Laboratories, and by NEC Research Institute.

[1]Throughout this paper, $\log N$ denotes $\log_2 N$.

with some fixed constant probability, an N-input multibutterfly can route $\log N$ permutations between some set of $\Theta(N)$ inputs and $\Theta(N)$ outputs in $O(\log N)$ time. (For a description of these and related results see [6].)

The Leighton-Maggs strategy for tolerating faults consists of two parts. First the network is *reconfigured*. Reconfiguring a network consists of identifying those parts that contain too many faults to be useful for routing, and removing them from the network. The goal is to leave intact as much of the working hardware as possible, while maintaining the important structural properties of the network. In the case of the multibutterfly, the crucial property is *expansion* (defined in Section 2). The Leighton-Maggs reconfiguration algorithm reduces the expansion of the network somewhat. However, as long as the reconfigured network has some expansion it is possible to apply a routing algorithm that was designed to run on a fault-free multibutterfy. Thus, the second part of the strategy is to apply an off-the-shelf multibutterfly routing algorithm, such as Upfal's permutation routing algorithm [12].

One of the drawbacks of the Leighton-Maggs reconfiguration algorithm is that it is performed by an off-line computer with knowledge of the state of the entire routing network. This paper presents an on-line algorithm for reconfiguring the network in $O(\log N)$ time. The algorithm is performed entirely by the network, even though many of its switches may be faulty.

The remainder of this paper consists of two sections. In Section 2 we describe butterfly and multibutterfly networks. In Section 3 we review the reconfiguration algorithm of Leighton and Maggs and then describe the on-line algorithm.

2 Butterflies and Multibutterflies

An example of an 8-input butterfly network is illustrated in Figure 1. The nodes in this graph represent switches, and the edges represent wires. Each node in the network has a distinct label (r, l), where r is the row, and l is the level. In a butterfly with N inputs, the row is a $\log N$-bit binary number and the level is an integer between 0 and $\log N$. The nodes on level 0 and $\log N$ are called the *inputs* and *outputs*, respectively. For $l < \log N$, a node labeled (r, l) is connected to nodes $(r, l+1)$ and $(r^l, l+1)$, where r^l denotes r with bit l complemented (bit 0 is the most significant, bit $\log N - 1$ the least).

In a butterfly, messages are typically sent from the switches on level 0,

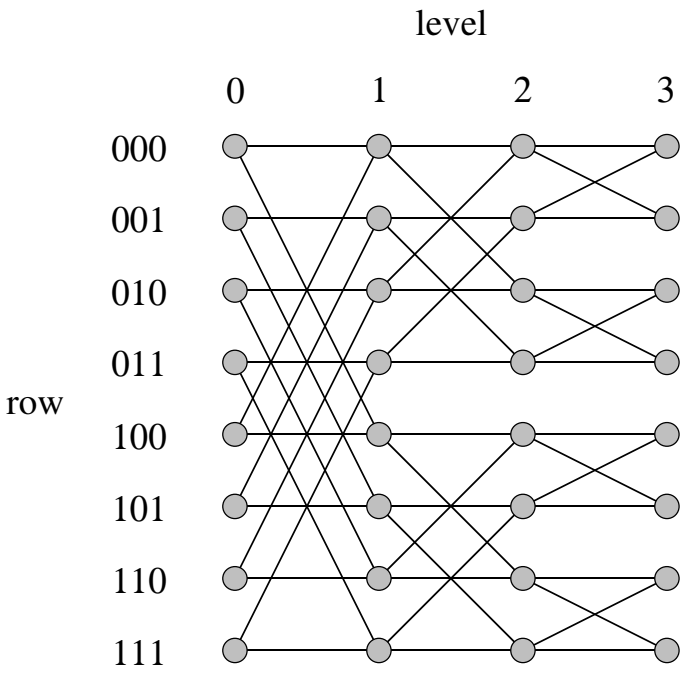

Figure 1: An 8-input butterfly network.

called the *inputs*, to those on level $\log N$, called the *outputs*. In a *one-to-one* routing problem, each input is the origin of at most one message, and each output is the destination of a most one message. One-to-one routing is also called *permutation routing*.

2.1 Dilated butterflies

Because message congestion is a common occurrence in real networks, the wires in butterfly networks are typically *dilated*, so that each wire is replaced by a *channel* consisting of 2 or more wires. In a d-dilated butterfly, each channel consists of d wires. Because it is harder to congest a channel than it is to congest a single wire in a butterfly, dilated butterflies are better routing networks than simple butterflies [3, 4, 10, 11].

2.2 Splitter networks

Butterfly and dilated butterfly networks belong to a larger class of networks called *splitter networks*. The switches on each level of a splitter network can be partitioned into *blocks*. All of the switches on level 0 belong to the

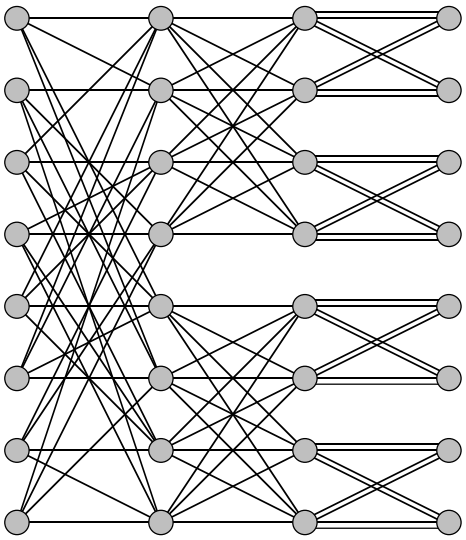

Figure 2: An 8-input splitter network with multiplicity 2.

same block. On level 1, there are two blocks, one consisting of the switches that are in the upper $N/2$ rows, and the other consisting of the switches that are in the lower $N/2$ rows. In general, the switches in a block B of size $M = N/2^l$ on level l have neighbors in two blocks, B_u and B_l, on level $l+1$, where u stands for *upper* and l for *lower*. The upper block, B_u, contains the switches on level $l+1$ that are in the same rows as the upper $M/2$ switches of B. The lower block, B_l, consists of the switches that are in the same rows as the lower $M/2$ switches of B. The edges from B to B_u are called the *up* edges, and those from B to B_l are called the *down* edges. The three blocks, B, B_u, and B_l, and the edges between them are collectively called a *splitter*. The switches in B are called the *splitter inputs*, and those in B_u and B_l are called the *splitter outputs*. In a splitter network with *multiplicity* d, each splitter input has d outgoing up edges and d outgoing down edges, and each splitter output has $2d$ incoming edges. In a d-dilated butterfly, the d up (and d down) edges out of each splitter input all lead to the same splitter output, but this need not be the case in general. For example, we have illustrated an 8-input splitter network with multiplicity 2 in Figure 2.

In a splitter network, each input and output are connected by a single

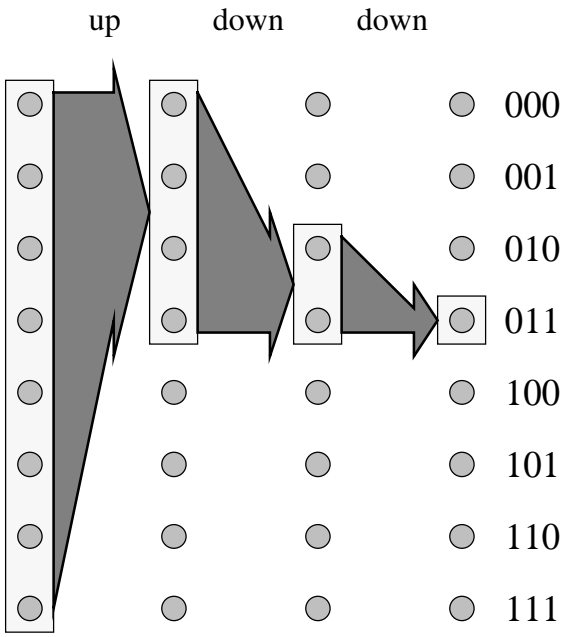

Figure 3: The logical path from any input to output 011.

logical (up-down) path through the blocks of the network. The path is determined by the address of the output. At the ith level of the network, an up edge is taken if the ith bit of the output's address is 0. Otherwise a down edge is taken. As an example, Figure 3 shows the logical path from any input to output 011, which consists of an up edge followed by two down edges. In a butterfly, this logical path specifies a unique path through the network, since only one up and one down edge emanate from each switch. (In fact, a splitter network with multiplicity one is very similar to a delta network [5].) In a general splitter network with multiplicity d, however, each switch will have d up and d down edges, and each step of the logical path can be taken on any one of d edges. Hence, one logical path can be realized by a myriad of physical paths in a general splitter network.

2.3 Randomly-wired splitter networks and multibutterflies

In this paper, we are primarily concerned with randomly-wired splitter networks. A *randomly-wired splitter network with multiplicity* d is a splitter network where the up and down edges within each splitter are chosen at random subject to the constraint that each splitter input has d up and d

down edges, and each splitter output has $2d$ incoming edges.

The crucial property that randomly-wired splitter networks are likely to possess [7, 12] is known as *expansion*. In particular, an M-input splitter is said to have (α, β)-*expansion* if every set of $k \leq \alpha M$ inputs is connected to at least βk up outputs and βk down outputs, where $\alpha > 0$ and $\beta > 1$ are fixed constants. For example, see Figure 4.

A splitter network is said to have (α, β)-expansion if all of its splitters have (α, β)-expansion. More simply, a splitter or a splitter network is said to have *expansion* if it has (α, β)-expansion for some constants $\alpha > 0$ and $\beta > 1$. A splitter network with expansion is more commonly known as a *multibutterfly* [12].

Splitters with expansion are known to exist for any $d \geq 3$, and they can be constructed deterministically in polynomial time [2, 9, 12], but randomized wirings typically provide better expansion. A discussion of the tradeoffs between α and β in randomly-wired splitters, can be found in [7] and [12]. For the purposes of this paper, two facts are needed. First, for fixed d and sufficiently small α, the expansion, β, of a randomly-wired splitter will be close to $d-1$ with probability close to 1. Second, for fixed α and sufficiently large d, $\alpha\beta$ will be close $1/2$ (the best possible) with probability close to 1. It is not known if it is possible for β to be close to $d-1$ and for $\alpha\beta$ to be close to $1/2$ simultaneously.

A multibutterfly with (α, β)-expansion is good at routing because one must block βk splitter outputs in order to block k splitter inputs. In classical networks such as the butterfly, the reverse is true: it is possible to block $2k$ inputs by blocking only k outputs. When this effect is compounded over several levels, the effect is dramatic. In a butterfly, a single fault can block 2^l switches l levels back, whereas in a multibutterfly, it takes β^l faults to block a single switch l levels back.

3 Routing around faults

In this section, we present an $O(\log N)$ time on-line algorithm for reconfiguring a multibuttery network in the presence of faults. We begin in Section 3.1 by describing the fault model. In Section 3.2 we review the off-line algorithm of Leighton and Maggs. Next, in Section 3.3, we describe the on-line algorithm. To simplify the presentation of the algorithm, we augment the multibutterfly with some additional edges. These edges increase the size and VLSI layout area of the network by at most a constant factor. As it

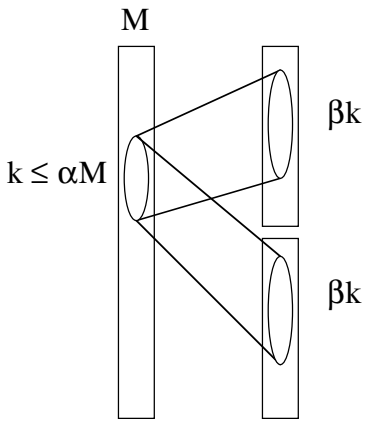

Figure 4: An M-input splitter with (α, β)-expansion.

turns out, this additional hardware is not really necessary. We conclude in Section 3.4 by explaining how to implement the algorithm without using these extra edges.

3.1 The fault model

The reconfiguration algorithm and the routing algorithms in [6] tolerate static, non-malicious faults in the switches. In the *static* fault model, some faulty switches may be produced by the manufacturing process but once the network has been manufactured, no working switch ever fails, and no faulty switch ever begins to work. We shall assume that failures are *non-malicious* in the sense that a working switch can query any one of its neighbors and determine if that neighbor is faulty in constant time.

3.2 The Leighton-Maggs algorithm

The Leighton-Maggs algorithm consists of two parts. The first part, called *erasure*, removes some of the outputs from the network. The second part, called *fault propagation*, removes some of the inputs. The goal of the reconfiguration algorithm is to leave intact a large working subnetwork in which every input can reach every output, and in which the splitters have (α, β')-expansion, where β' may be less than β, but must be greater than

one. The proof that a fault-free multibutterfly can route $\log N$ permutations in $O(\log N)$ time holds for any expansion β greater than one [6, 12]. By a similar argument, if $\beta' > 1$, the subnetwork also can route any $\log N$ permutations between its inputs and outputs in $O(\log N)$ time [6].

The erasure part of the algorithm consists of removing those splitters that contain too many faults. This step requires some off-line computation. Each splitter in the multibutterfly is examined, and if more than an ε fraction of its input switches are faulty, where $\varepsilon = 2\alpha(\beta' - 1)$ and $\beta' = \beta - \lfloor \frac{d}{2} \rfloor$, then the splitter is "erased" from the network as are all of the switches that can be reached from the splitter, including outputs on level $\log N$. In the next section, we will present an on-line algorithm for counting the number of faults in each splitter.

The second part of the algorithm, fault-propagation, is executed by the switches themselves. Working from level $\log N$ backwards, each switch checks if at least half of its upper output edges lead to faulty switches that have not been erased, or if at least half of its lower output edges lead to faulty switches that have not been erased. If so, then it declares itself to be faulty (but does not erase itself). Such a fault is called a *propagated fault.*

Finally, all of the remaining faulty switches are erased. Since every remaining input in every splitter is linked to at least $\lceil \frac{d}{2} \rceil$ working upper outputs (if the descendant multibutterfly outputs have not been erased) and $\lceil \frac{d}{2} \rceil$ working lower outputs (if the corresponding multibutterfly outputs have not been erased), the network has (α, β')-expansion.

The following pair of lemmas bounds the number of removed inputs and outputs in the case of worst-case faults and random faults, respectively.

Lemma 3.1 ([6]) *Suppose that there are f faults in the network. Then the erasure process removes at most $f/\varepsilon = O(f)$ outputs, and there will be at most $\frac{f}{\beta'-1} = O(f)$ propagated faults on any level.*

Lemma 3.2 ([6]) *There exist fixed constants $p > 0$, and $\lambda > 0$, such that if each switch fails independently with probability p, then with probability at least $1 - e^{-\Theta(N/\log^2 N)}$ the erasure and fault propagation processes leave behind at least λN inputs and λN outputs*

3.3 On-line reconfiguration

This section presents an algorithm for determining which switches to remove from the network in $O(\log N)$ time. The algorithm is *on-line* in the sense

that the computation is performed entirely by the switches, without the aid of any off-line computation. As in Section 3.2, the reconfiguration of the network consists of two parts. First, each splitter must determine if the number of faults in its input block exceeds an ε fraction, and, if so, then it must erase itself. This part is difficult because each splitter must count its own faults and distribute the count to its working switches, even though the splitter itself may contain many faults. Second, faults are propagated from the outputs of the network towards the inputs. A switch is declared faulty if more than $d/2$ of its upper or lower output edges lead to switches that are faulty, but not erased. After erasure and fault propagation, all of the remaining faulty switches are erased.

The erasure part consists of two tasks. First, we must identify those blocks that contain too many faults and must be erased. Then for each splitter, each input switch must be told whether either of the two output blocks in the splitter have been erased. To help with these tasks, we add some edges to the network. In particular, each switch is connected in a random fashion to d_2 other switches in the same block. These edges will be used solely for the purpose of counting, and not for routing. They increase the VLSI layout area of the network by at most a constant factor. For sufficiently small, but fixed, $\alpha_2 > 0$, with probability close to 1, every set S of $k \leq \alpha_2 M$ switches in a block of size M will have at least $\beta_2 k$ neighbors [7, 12]. We will choose α_2 to be small so that β_2 will be close to $d_2 - 1$.

3.3.1 Blocks with too many faults fall asleep

The erasure algorithm begins by repeating the following *basic step* δ times, where $\delta = \lceil \log N / \log \beta_2' \rceil$, and $\beta_2' = \beta_2 - \lfloor d_2/2 \rfloor$. Initially, every working switch in the network is *awake*. At each basic step, each switch that is awake examines its d_2 neighbors within the same block, and falls asleep if more than $\lfloor d_2/2 \rfloor$ of them are faulty or asleep. The following lemma shows that if there were not too many faults to begin with, then few working switches fall asleep.

Lemma 3.3 *Let t denote the number of faults in a block of size M, and let s denote the number of working switches that fall asleep. If $t < \varepsilon_2 M$, where $\varepsilon_2 = \alpha_2(\beta_2 - \lfloor d_2/2 \rfloor - 1)$, then $s < \alpha_2 M$ and $s < t/(\beta_2 - \lfloor d_2/2 \rfloor - 1)$.*

Proof: The proof is by induction on the number of basic steps. The base case is trivial, since initially no working switches are asleep. Now let U_i be

the set of working switches that are asleep at the end of step i. Suppose that $|U_i| \geq \alpha_2 M$. Then

$$|U_{i-1}| + t \geq \left(\beta_2 - \left\lfloor \frac{d_2}{2} \right\rfloor\right) \alpha_2 M,$$

since the switches in U_i have at least $\beta_2 \alpha_2 M$ neighbors, and each switch in U_i has at most $\lfloor d_2/2 \rfloor$ neighbors that are not asleep or faulty. Since $|U_{i-1}| < \alpha_2 M$ by induction, and $t < \varepsilon_2 M$, we have a contradiction. Thus $|U_i| < \alpha_2 M$. As a consequence, the switches in U_i have at least $\beta_2 |U_i|$ neighbors. Thus,

$$|U_{i-1}| + t \geq \left(\beta_2 - \left\lfloor \frac{d_2}{2} \right\rfloor\right) |U_i|.$$

Now suppose that $|U_i| \geq t/(\beta_2 - \lfloor d_2/2 \rfloor - 1)$. Since $|U_{i-1}| < t/(\beta_2 - \lfloor d_2/2 \rfloor - 1)$ by induction, we have a contradiction. □

The next lemma shows that if any working switch is awake after step δ, then it is connected to many nearby working switches.

Lemma 3.4 *If a switch r in a block of size M is awake after step δ, then at least $\alpha_2 \beta_2' M$ working switches can be reached from r along paths of length at most δ that pass through only working switches.*

Proof: If r is still awake at the end of step i, then r must have had at least $\beta_2' = \beta_2 - \lfloor d_2/2 \rfloor$ neighbors that were awake at the end of step $i-1$. In turn, these neighbors must have had at least $\beta_2'^2$ neighbors that were awake at the end of step $i-2$, provided that $\beta_2' \leq \alpha_2 M$. Otherwise, these switches had at least $\alpha_2 \beta_2' M$ neighbors that were awake. In general, r can reach a set of $\min\{\beta_2'^j, \alpha_2 \beta_2' M\}$ switches that were awake at the end of step $i-j$. Furthermore, there is a path of length j from r to any switch in this set that passes only through working switches. Choosing j such that $\min\{\beta_2'^j, \alpha_2 \beta_2' M\} = \alpha_2 \beta_2' M$, i.e.,

$$\begin{aligned} j &= \left\lceil \frac{\log \alpha_2 \beta_2' M}{\log \beta_2'} \right\rceil \\ &\leq \left\lceil \frac{\log N}{\log \beta_2'} \right\rceil \\ &= \delta, \end{aligned}$$

and choosing $i = \delta$ completes the proof. □

3.3.2 Input blocks count awake switches in output blocks

Now the erasure algorithm must determine, for each splitter, whether its output blocks contain too many faults, and it must inform each input switch if either of the two output blocks must be erased.

In order to count the number of faulty switches in the output blocks, the switches in each input block organize themselves into trees. Suppose that some switch r in a block of size M remains awake for δ steps. We call r a *ruler*. Each ruler attempts to form a depth-δ breadth-first spanning tree of the working switches that it can reach with itself as the root. If r were the only ruler, then the number of steps required to form the spanning tree would be at most δ. However, since each ruler simultaneously attempts to form a spanning tree, there will be conflicts when the trees overlap. To resolve these conflicts, we will assume that each switch in the block possesses a distinct label. Each time a switch is added to a spanning tree, it is given the label of the spanning tree's ruler. If several trees attempt to add the same switch, then the one with the smallest label succeeds, even if the switch must be removed from another tree. Since the growth of the spanning tree with the smallest label is unimpeded by the other spanning trees, after δ steps, it will contain at least $\alpha_2\beta_2' M$ switches.

Next, in δ steps, each ruler counts the number of switches in its spanning tree. If the total is at least $\alpha_2\beta_2' M$, then it broadcasts a message to the switches in the tree, telling them that they belong to a *large tree*.

Now each large tree makes an underestimate of the number of switches that are awake in the upper and lower output blocks. In order to perform this task, a third set of edges is added to the graph. For each input switch, d_3 edges are added to switches in both the upper and lower output blocks of outputs at the next level. The edges are inserted at random so that each set of $k \leq \alpha_3 M$ switches in a block of size M has at least $\beta_3 k$ neighbors in both the upper and lower blocks, where $\alpha_3\beta_3 < 1/2$. These edges increase the VLSI layout area of the network by at most a constant factor. We will choose $\alpha_3 = \alpha_2\beta_2'$ so that a tree of size $\alpha_2\beta_2' M$ will have at least $\alpha_2\beta_2'\beta_3 M$ neighbors in each output block, and we will choose d_3 to be large so that $\alpha_2\beta_2'\beta_3 > (1 - (\varepsilon_2 + \alpha_2))/2$. In δ steps, each large tree sums up the number of different switches in the upper output block that are awake and have a neighbor in the tree. It then does the same for the lower output block.

3.3.3 Output blocks that are not to be erased are marked

If any large tree in the input block counts more than $(1 - 2(\varepsilon_2 + \alpha_2))M/2$ switches that are awake in an output block of size $M/2$ then it *marks* all of those switches. As we shall see, a block will be erased unless it contains a marked switch, in which case it will not be erased. The following lemma bounds the number of network outputs that are erased.

Lemma 3.5 *Let f denote the number of faults in the entire network. Then the total number of erased network outputs is at most f/ε_2.*

Proof: If an output block of size $M/2$ has fewer than $\varepsilon_2 M/2$ faults, then by Lemma 3.3, after δ steps it will have at most $(\varepsilon_2 + \alpha_2)M/2$ faulty and asleep switches. Since the switches in each large tree have at least $(1-(\varepsilon_2 + \alpha_2))M/2$ neighbors in each output block, at least $(1 - 2(\varepsilon_2 + \alpha_2))M/2$ of those neighbors must be awake. These neighbors will all be marked and the block will not be erased. Thus, if an output block is erased, then it must have had at least an ε_2 fraction of faulty switches to begin with. □

3.3.4 Input switches are told whether output blocks are erased

After the large trees have marked switches in the output blocks that are not to be erased, the rest of the input switches that are awake must be informed. First, every working input switch (awake or asleep) queries its d_3 upper output neighbors to determine if they are marked. If any of them are, then the input switch *colors* itself. Then the following *coloring step* is repeated δ times. If any of an input's d_2 neighbors in the input block are colored, then the switch colors itself. (The same algorithm is then applied to the lower output block.) The following lemma shows that after δ steps, each input switch will know if the upper output block has been erased.

Lemma 3.6 *If any switch in an upper output block is marked, then every awake switch in the input block will be colored in δ coloring steps, provided that $\alpha_2\beta_2'\beta_3 > 1/4$.*

Proof: If any switch in an upper output block of size $M/2$ is marked, then at least $\alpha_3 M = \alpha_2\beta_2'\beta_3 M$ of them are marked, which for $\alpha_2\beta_2'\beta_3 > 1/4$ is more than half of the switches in the block. By Lemma 3.4, each input switch in a block of size M that is awake can reach at least $\alpha_2\beta_2' M$ other working switches via paths of length at most δ. These switches have at least

$\alpha_2\beta_2'\beta_3 M$ neighbors in the upper output block, which for $\alpha_2\beta_2'\beta_3 > 1/4$ is more than half of the switches in the block. If more than half of the switches in the upper output block are marked, and more than half of the switches are neighbors, then at least one neighbor is marked. □

The last step before fault propagation is to declare any switch that is asleep to be faulty. The following lemma shows that the blocks that are not erased contain at most a $2(\varepsilon_2 + \alpha_2)$ fraction of faulty and asleep switches.

Lemma 3.7 *If a block of size $M/2$ is not erased, then it has at most $(\varepsilon_2 + \alpha_2)M$ faulty or asleep switches.*

Proof: If an output block of size $M/2$ has more than $(\varepsilon_2 + \alpha_2)M$ faulty or asleep switches, then every large tree in the corresponding input block has at most $(1 - 2(\varepsilon_2 + \alpha_2))M/2$ neighbors in the output block that are awake, and none of those neighbors will be marked. □

The algorithm for propagating faults from the outputs to the inputs in $O(\log N)$ time is the same as the propagation algorithm from the Leighton-Maggs algorithm. It consists of $\log N$ stages, numbered 1 through $\log N$. At stage i, each switch on level $\log N - i$ counts the number of faulty neighbors it has on level $\log N - i + 1$. If more than half of its upper or lower outputs lead to faulty switches that have not been erased, then the switch declares itself to be faulty. Otherwise it does nothing. We will choose $2(\varepsilon_2 + \alpha_2) < \varepsilon$, so that each unerased block has at most an ε fraction of faulty switches. As a consequence, we can apply Lemmas 3.1 and 3.2 to bound the number of faults that propagate to the inputs.

3.3.5 A final look at the constants

At this point, it seems wise to verify that all of the constraints on the constants α, β, d, ε, α_2, β_2, d_2, ε_2, α_3, β_3, and d_3 can be satisfied. First, we must choose α small so that β is close to $d - 1$. Second, we choose α_2 small so that β_2 is close to $d_2 - 1$. This choice will make $\varepsilon_2 = \alpha_2(\beta_2 - \lfloor d_2/2 \rfloor - 1)$ small. Third, we set $\alpha_3 = \alpha_2\beta_2'$, which is small, and choose d_3 large so that $\alpha_3\beta_3$ is close to $1/2$. In particular, we need $\alpha_3\beta_3 \geq (1 - (\varepsilon_2 + \alpha_2))/2$ and $\alpha_3\beta_3 > 1/4$. Finally, we need $2(\varepsilon_2 + \alpha_2) \leq \varepsilon$.

3.4 Removing the additional edges

The algorithm of Section 3.3 augments the multibutterfly network with two types of edges. First d_2 edges are added from each switch to switches in the

same block. Then d_3 edges are added from each switch to both the upper and lower blocks at the next level. The second type of edges are easily removed. Their tasks can be performed by the d routing edges leading from each switch to the upper and lower blocks at the next level.

Removing the first type of edges is more problematic. The basic idea is to simulate them using the d routing edges. We begin by observing that a randomly-wired splitter is likely to have expansion both from the inputs to the outputs and from the outputs to the inputs [7, 12]. Let (α_4, β_4) be the expansion property from the output blocks (taken together) to their input block. Then $\alpha_4\beta_4 < 1$, and β_4 will be close to $2d - 1$, provided that α_4 is sufficiently small. A set of $k \leq \alpha M$ input switches in a block of size M has at least $2\beta k$ output neighbors (counting those in both the upper and lower blocks). These outputs in turn have at least $2\beta_4\beta k$ input neighbors, provided $2\beta k \leq \alpha_4 M$. Thus, as long as none of the output neighbors are faulty, they can be used to simulate expansion $2\beta_4\beta$ within the block. This expansion can be used in place of β_2 in the algorithm of Section 3.3. What if some of these output neighbors are faulty? This problem can be solved by declaring a switch to be faulty if *any* of its output neighbors are faulty (*without propagating any faults*) before the reconfiguration process begins. This trick may multiply the number of faults in the network by a factor of $2d$, but if a switch survives then all of its output neighbors were initially working.

4 Remarks

The techniques described in this paper can also be applied to other networks whose underlying structures are trees, and whose blocks are connected by expanding graphs. One example is a class of networks called *multi-fat-trees*, which are based on the fat-tree networks of Leiserson and Greenberg [1, 8].

References

[1] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.* Volume 5 of *Advances in Computing Research*, pages 345–374. JAI Press, Greenwich, CT, 1989.

[2] N. Kahale. Better expansion for Ramanujan graphs. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 398–404. IEEE Computer Society Press, October 1991.

[3] R. R. Koch. Increasing the size of a network by a constant factor can increase performance by more than a constant factor. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 221–230. IEEE Computer Society Press, October 1988.

[4] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, C–32(12):1091–1098, December 1983.

[5] C. P. Kruskal and M. Snir. A unified theory of interconnection network structure. *Theoretical Computer Science*, 48:75–94, 1986.

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

[7] T. Leighton, C. L. Leiserson, and M. Klugerman. Theory of parallel and VLSI computation. Research Seminar Series Report MIT/LCS/RSS 10, MIT Laboratory for Computer Science, May 1991.

[8] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[9] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

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

[11] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson. The monarch parallel processor hardware design. *Computer*, 23(4):18–30, April 1990.

[12] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 241–250, May 1989.

Andrew V. Goldberg received the S.B. degree in mathematics from the Massachusetts Institute of Technology in 1982, the S.M. degree in computer science from the University of California at Berkeley in 1983, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology in 1987. Currently he is an Associate Professor of Computer Science at Stanford University. His research interests include sequential and parallel algorithms and combinatorial optimization.

Bruce Maggs (M'92) received the S.B., S.M., and Ph.D., degrees in computer science from the Massachusetts Institute of Technology in 1985, 1986, and 1989, respectively. Currently he is a Research Scientist at the NEC Research Institute in Princeton, NJ. His research interests include parallel computing systems and parallel algorithms.

Serge Plotkin received the Ph.D. degree in computer science from the Massachusetts Institute of Technology in 1988. In 1988 he joined the Department of Computer Science at Stanford University. His main research interests include parallel and distributed computation, high-speed interconnection networks, and combinatorial optimization.