Efficient Determination of Maximum Incomplete Subcubes in Hypercubes with Faults

Nian-Feng Tzeng, *Senior Member, IEEE*,
and Guanghua Lin, *Member, IEEE Computer Society*

Abstract—After faults arise in a hypercube, it is often desirable to reconfigure the faulty hypercube in such a way as to retain as many fault-free nodes as possible, because system performance tends to be in proportion to the computational power, and a reconfigured hypercube with more nodes is likely to retain performance better. This inspires us to identify maximum *incomplete* subcubes in a faulty hypercube, as the subcube so reconfigured is often much larger than that reconfigured according to earlier schemes. Here we propose an efficient algorithm for determining maximum incomplete subcubes in faulty hypercubes. The basic idea is to construct a maximum incomplete subcube from a number of healthy complete subcubes of *distinct sizes*. To this end, an efficient procedure for finding all maximum fault-free complete subcubes in a faulty hypercube is introduced, and then an efficient algorithm for determining maximum incomplete subcubes is presented.

Index Terms—Fault patterns, hypercubes, incomplete subcubes, maximum subcubes, reconfiguration, time complexity.

1 Introduction

THE hypercube is a popular interconnection scheme for multiprocessor construction due to its attractive topological properties, simple routing, rich connection paths, and so on. For a large hypercube system, the probability of fault occurrence can be high, making it necessary to consider the fault-tolerant issue in system design. Fault-tolerant techniques suggested for the hypercube system fall into two categories, depending on whether or not redundant nodes/links are employed. If redundancy is added, the design goal is to keep the system size unchanged in the presence of operational failures after reconfiguration by replacing the failed components with spares [1], [2], [3], [4]. On the other hand, if no redundancy is involved in a hypercube, fault-tolerance is achieved either by utilizing the workable portion of the system to emulate the whole machine with certain slowdown [5], [6] or by reconfiguring the machine into a smaller sized system after faults occur [8], [9], [10].

All prior reconfiguration techniques attempt to identify *complete* fault-free subcubes with the maximum possible dimension in a faulty hypercube. These techniques are often unable to retain as many workable nodes as possible after reconfiguration due to the strong restriction on allowable system sizes. In this paper, we consider reconfiguring a faulty hypercube with an effort to keep as many fault-free nodes as possible. This often leads to an *incomplete hypercube* (to be defined formally in Section 2), whose size can be *arbitrary*. A reconfigured incomplete subcube is often much larger than a reconfigured complete subcube, because the former generally involves a copy of the latter *plus* some smaller sized subsystem(s). Katseff [11] has developed simple and deadlock-free algorithms for routing and broadcasting messages in incomplete hypercubes. Recent studies on the incomplete hypercube system suggest that no congestion point exists in such a system [12], [13]. As system performance tends to be in proportion to the computational power, a reconfigured incomplete subcube with more nodes is likely to retain performance better.

The basic idea of our algorithm for determining maximum incomplete subcubes in a faulty hypercube is to identify maximum collections of complete subcubes of distinct sizes such that elements in each collection together form an incomplete cube. It begins with finding all maximum fault-free complete subcubes in a faulty hypercube using an efficient procedure. To ensure that elements in each collection form an incomplete cube, search takes place within pairs of a maximum fault-free complete subcube and its "adjacent" faulty subcube of the same size (called a co-subcube) as follows: For each pair considered, the fault-free complete subcube is assigned to the collection currently in search, with its co-subcube treated as a faulty hypercube, in which a maximum incomplete cube is to be identified. This search process repeats until the current co-subcube contains no fault-free subcube. The total search space of the proposed algorithm is reduced significantly by utilizing some properties of faulty hypercubes.

It is worth noting that our reconfiguration scheme may drop certain fault-free nodes in order to preserve the incomplete hypercube topology. If a system attempts to maintain all the fault-free nodes without assuring the incomplete hypercube topology, it is impossible to have simple, deterministic routing and broadcasting algorithms. In fact, unlike an incomplete hypercube, this often requires complicated node-to-node routing and broadcasting on the basis of "sidetracking," "backtracking" [14], or duplicated transmission [15], which do not guarantee deadlock-freeness. Furthermore, traffic density over a link under this situation is not bounded by a small constant and a message is not necessarily routed through a shortest path (as in an incomplete hypercube), possibly causing excessive communication delay.

In the following, Section 2 introduces definitions and notations. Section 3 gives a procedure for finding all maximum complete subcubes in the presence of faulty nodes, followed by an efficient algorithm for identifying maximum incomplete subcubes in a hypercube involving faulty nodes. Section 4 extends the algorithm developed to deal with the system involving both faulty nodes and faulty links.

2 Definitions and Notations

An n-dimensional hypercube, H_n, is a graph of 2^n nodes, each with n links connected directly to its n immediate neighbors. A node in H_n is labeled by an n-bit binary string $b_{n-1}b_{n-2} \ldots b_1b_0$, and a link exists between two nodes whose labels differ in exactly one bit position. A k-dimensional subcube in H_n, S_k, is labeled by an n-tuple $\{0, 1, *\}^n$ such that there are exactly k ($k < n$) *don't care* (*) positions in its label. A subcube will be represented by its label, and the term "subcube" will be used to indicate a complete subcube unless stated otherwise.

DEFINITION 1. *An n-dimensional incomplete hypercube of size M, I_n^M, is defined recursively as follows:*

1) $I_n^M = H_{n-1} + I_k^{M-2^{n-1}} (k < n)$;
2) *A link exists between node x in H_{n-1} and node y in $I_k^{M-2^{n-1}}$ if and only if their labels differ in exactly one bit.*

The incomplete hypercubes under consideration are reconfigured from a faulty H_n. They can be of any size less than 2^n, depending on the fault patterns. For example, a three-dimensional

• *N.-F. Tzeng is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504. E-mail:tzeng@cacs.usl.edu.*
• *G. Lin was with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.*

Manuscript received May 6, 1994; accepted December 26, 1995.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96076.

incomplete subcube of size 7 reconfigured from a faulty H_4, I_3^7, is shown by bold lines in Fig. 1.

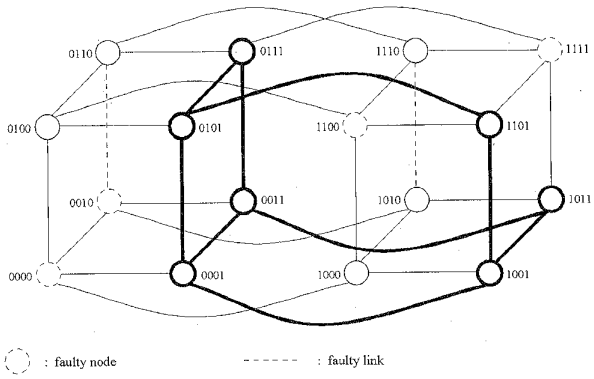

Fig. 1. A four-dimensional hypercube, in which an incomplete subcube with seven nodes is shown by bold lines.

DEFINITION 2. *Two subcubes of the same dimension are said to be lateral with parameter δ if the "*" positions in their labels are exactly the same and the remaining bits differ in δ and only δ positions. A subcube involving fault(s) is said to be a co-subcube (COS for short) of a fault-free subcube if they are lateral with $\delta = 1$.*

In Fig. 1, for example, 1**1 and 0**0 are lateral with $\delta = 2$; 1**1 is a *COS* of fault-free subcube 0**1. A *COS* contains at least one fault (either a faulty link or a faulty node), and a fault-free subcube may have more than one *COS*.

According to Definition 1, an incomplete hypercube is essentially a collection of fault-free subcubes of distinct dimensions. Such a collection can be defined as follows:

DEFINITION 3. *A collection corresponding to an n-dimensional incomplete hypercube, C_n, is defined recursively as:*

1) $C_n = H_{n-1} \cup C_k$ $(k < n)$;
2) *C_k is a collection resided in a COS of H_{n-1}.*

For example, a three-dimensional incomplete subcube shown by bold lines in Fig. 1, is a collection of {0**1, 10*1, 1101}, where the two smaller fault-free subcubes, 10*1 and 1101, reside in a *COS* of healthy subcube 0**1, namely, 1**1. Since C_p $(p \le n)$ comes from a pair of H_{p-1} and its *COS*, a maximum collection (i.e., a collection corresponding to a maximum incomplete subcube) in a faulty H_n is derived from such a pair that involves a maximum (fault-free) complete subcube (*MCS* for short) in H_n and a *COS* of the *MCS*. Such a pair, (*MCS*, *COS*), is of our interest and is referred to as an *M-C* pair.

A *maximum incomplete subcube* in a faulty hypercube is an incomplete (healthy) hypercube with the largest possible size. The incomplete subcube composed of bold lines in Fig. 1, I_3^7, is in fact a maximum incomplete subcube in H_4 under the given fault pattern. It is so because no other healthy node can be added without making it ineligible as an incomplete hypercube characterized by Definition 1. For example, node 1000, although healthy, cannot be added due to the existence of a zero-dimensional subcube (i.e., node 1101) in I_3^7; another reason is that node 1000 does not belong to the *COS* of constituent subcube 0**1. Our solution to the problem of finding maximum incomplete subcubes in a faulty H_n is equivalent to finding maximum collections of fault-free subcubes of distinct dimensions, and it builds on an algorithm for identifying all *MCS*s in H_n, as described below. For easy explanation, we first restrict our attention to hypercubes with faulty nodes only. The algorithm developed is then extended to deal with hypercubes involving both faulty nodes and faulty links.

3 Reconfiguring Hypercubes with Faulty Nodes

3.1 Find Maximum Complete Subcubes (MCSs)

There are $\binom{n}{n-k}2^{n-k}$ k-dimensional subcubes in H_n. This comes from the fact that for a general n-tuple $\{0, 1, *\}^n$, there are exactly $\binom{n}{n-k}$ different $(n-k)$-bit position combinations, each having 2^{n-k} combinations of $\{0, 1\}^{n-k}$, resulting in a total of $\binom{n}{n-k}2^{n-k}$ different labels (after assigning "*" to the remaining k positions), each corresponding to an S_k. In H_n, a set of faulty nodes, F, is said to cover r combinations of $\{0, 1\}^{n-k}$ at the given $(n-k)$ bit positions if F contains r distinct combinations of $\{0, 1\}^{n-k}$ at those $(n-k)$ bit positions, where $r \le 2^{n-k}$. Fig. 2 shows an example in which a set of faulty nodes F covers various combinations when different sets of bit positions are concerned. If a combination is covered by F at the given $(n-k)$ bit positions, the subcube with its label composed of the combination at those given $(n-k)$ bit positions and "*" at the remaining k positions, contains at least one faulty node. As an instance, in Fig. 2, the combination "01" is covered by F at bit positions 1 and 0, indicating that subcube **01 contains at least one faulty node (in this case, it actually involves two faulty nodes, 0001 and 1101). Similarly, subcube **11 is faulty since the combination "11" is covered by F at bit positions 1 and 0. It can be seen that if all the 2^{n-k} combinations at each set of $(n-k)$ bit positions in faulty H_n are exhaustively covered by faulty nodes, then every S_k contains at least one faulty node, and thus there is no fault-free subcube with dimension k or larger. On the other hand, if a combination at any $(n-k)$ bit positions is absent, i.e., not covered by the set of faulty nodes, a fault-free subcube can be identified by assigning the combination to those particular $(n-k)$ bit positions and "*" to the remaining k positions. Our solution for the problem of identifying all *MCS*s in the presence of a given set of faulty nodes is equivalent to finding all the combinations absent from each set of $(n-k)$ bit positions, starting with $k = n-1$, as shown in Algorithm 1, where m is the number of faulty nodes.

Algorithm 1: Finding all *MCS*'s in the presence of faulty nodes

Input: The addresses of m faulty nodes in H_n;
Output: All *MCS*s in H_n;

```
L = φ; done = 0;
k = n − 1;
repeat
    for each set of (n−k) bit positions do
        determine whether or not any absent combination exists
          according to the addresses of m faulty nodes;
        if any absent combination exists then
            identify all absent combinations;
            append the absent combinations to L;
            done = 1; /*no need to further examine any smaller k*/
        end if
    end for
    k = k − 1;
until done = 1.
```

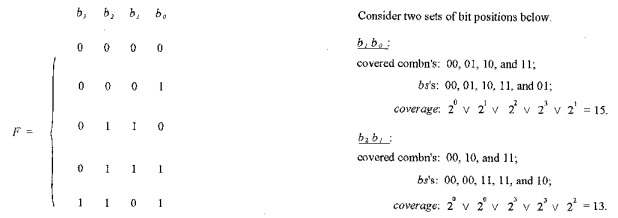

Fig. 2. An example in which F covers various combinations when different sets of bit positions are concerned.

To determine whether or not any absent combination exists for each set of $(n-k)$ bit positions, a value, *coverage*, is computed as follows: First, for each faulty node, determine $|bs|$, the value of an $(n-k)$-bit string bs composed of those particular $(n-k)$ bits in the node label, and evaluate $2^{|bs|}$. Next, apply the bitwise *OR* operation to the values evaluated, to get *coverage*. If *coverage* reaches $(2^{2^{n-k}}-1)$, then, there is no combination absent from that set of $(n-k)$ bit positions. Otherwise, absent combinations exist and can be derived directly from the resulting *coverage*. Consider the example given in Fig. 2, *coverage* is $2^0 \vee 2^1 \vee 2^2 \vee 2^3 \vee 2^1 = 15$ when bit positions 0 and 1 are considered. This indicates that there is no combination absent from bit positions 1 and 0. On the other hand, when bit positions 2 and 1 are concerned, *coverage* is $2^0 \vee 2^0 \vee 2^3 \vee 2^3 \vee 2^2 = 13 < 15$, suggesting that combination "01" is absent from bit positions 2 and 1. If those $|bs|$s used in the last search repetition are saved, the complexity of computing $|bs|$s in the current repetition can be reduced, because each $|bs|$ is then obtained simply by adding $b_j 2^{n-k-1}$ to the corresponding $|bs|$ saved, where j is the highest bit position at the current search. Initialized with an empty list, L contains all the absent combinations (which correspond to *MCS*s) in a faulty H_n when Algorithm 1 terminates.

3.1.1 Time Complexity

For each k in the outmost loop, there are $\binom{n}{n-k}$ different sets of $(n-k)$ bit positions. At the initial search repetition (i.e., $n-k=1$), each $|bs|$ can be computed in a constant time. In each subsequent search, a $|bs|$ can be obtained in a constant time as well, according to our prior statement. Let t_i be the time spent in computing *coverage* at the ith search repetition, and d be the dimension of *MCS* in a faulty H_n, then the total time spent in computing all the *coverages* during the entire search process can be expressed as

$$T_1(n) = \sum_{k=d}^{n-1} t_{n-k}\binom{n}{n-k} = \sum_{i=1}^{n-d} t_i \binom{n}{i}$$

(by letting $i = n-k$). All zero bit positions (each of which corresponds to one absent combination) in *coverage* can be identified by checking all the bits of *coverage*. For each set of $(n-k)$ bit positions examined, a *coverage* is computed and the *coverage* is of 2^{n-k} bits, which take 2^{n-k} steps (using the shift operation) to get every zero bit position. As a result, checking *coverage* to identify all absent combinations takes

$$T_2(n) = \sum_{k=d}^{n-1} 2^{n-k}\binom{n}{n-k} = \sum_{i=1}^{n-d} 2^i \binom{n}{i},$$

which is obviously less than

$$(n-d)2^{n-d}\binom{n}{n-d}$$

It has been shown in [7] that an upper bound for $(n-d)$, the number of dimensions destroyed by m faulty nodes, satisfies $(n-d-1) \le \log_2 m$, i.e., $2^{n-d-1} \le m$. If we assume a machine model in which the word consists of c bits, then 2^{n-d} bits can be kept in $\lceil 2m/c \rceil$ words. This indicates that a bitwise *OR* operation is done in time $O(m)$, and *coverage* in the $(n-d)$th repetition can be computed in time $O(m^2)$, because there are $(m-1)$ *OR* operations involved. (Note that a bitwise *OR* operation is assumed to take a constant time in [8], which seems inappropriate.) The worst case time complexity of Algorithm 1 is thus

$$T(n) = T_1(n) + T_2(n) < \sum_{i=1}^{n-d} t_i\binom{n}{i} + (n-d)2^{n-d}\binom{n}{n-d},$$

which is smaller than

$$\sum_{i=1}^{n-d} t_i\binom{n}{i} + O\left(\log_2 m \cdot m \cdot \binom{n}{n-d}\right)$$

by making use of the result of $2^{n-d-1} \le m$ [7]. As $t_1 < t_2 < \ldots < t_{n-d} = O(m^2)$, the preceding expression is less than

$$O\left(\sum_{i=1}^{n-d} m^2 \cdot \binom{n}{i}\right) + O\left(\log_2 m \cdot m \cdot \binom{n}{n-d}\right),$$

which is dominated by the first term, yielding the worst case time complexity of Algorithm 1

$$O\left(\sum_{i=1}^{n-d} m^2 \cdot \binom{n}{i}\right) < O\left(m^2 2^n\right) = O\left(m^2 N\right),$$

where $N (= 2^n)$ is the system size.

Work has been done in finding *MCS*s in a faulty H_n. In particular, Sridhar and Raghavendra [9] have shown an algorithm to find all the *MCS*s with time complexity $T(n) = O(nh^2)$, where h is the number of fault-free nodes. Since the number of fault-free nodes is generally of order $O(N)$, Algorithm 1 presented here compares favorably to that in [9] when the number of faults, m, is much smaller than N (say $m = O(n)$). Ozguner and Aykanat [8] have introduced an algorithm with time complexity in the worst case scenario being

$$O\left((n-d)2^m + m^2(n-d)\binom{n}{d}\right) \le O\left(n2^m + m^2 n\binom{n}{\frac{n}{2}}\right) = O\left(n2^m + m^2\sqrt{n}2^n\right).$$

to find the dimension and the number of *MCS*s, but not the locations of *MCS*s. Clearly, Algorithm 1 presented here is more efficient and useful.

While our Algorithm 1 is superior to the two prior algorithms [8], [9] in the worst case, it is interesting to see if this also holds true in the average case. To this end, we implemented the two earlier algorithms and Algorithm 1 using C and collected the simulation results under various fault patterns for different n and m values. The results demonstrate that in the average case, Algorithm 1 is more efficient than the one proposed in [9] when $m = O(n)$ and is always much faster than the algorithm considered in [8]. As an example, for $n = 8$ and $m = 15$, Algorithm 1 took about 106 *ms* (on SUN SPARCstation 2) to identify all the *MCS*s, whereas the algorithm given in [9] required more than 240 *ms* to finish and the algorithm proposed in [8] spent roughly 472 *ms*. A distributed procedure for finding fault-free subcubes was given in [10], and it was shown to have a polynomial time complexity with respect to n and m, i.e., $O(n^2m^2)$, at each (fault-free) node. Unfortunately, this procedure does not guarantee discovering *MCS*s at all times, and its accumulated time complexity of all nodes involved is higher than ours, as in general, it involves $O(2^n)$ nodes.

3.2 Identifying Maximum Incomplete Subcubes

Our approach is to examine *M-C* pairs and find the maximum collections. The subsequent properties of faulty hypercubes help to reduce significantly search space, or equivalently the total number of *M-C* pairs examined.

3.2.1 Properties of Faulty Cubes

An *M-C* pair may generate multiple collections, due to different ways of selecting fault-free subcubes of distinct dimensions. The maximum collection generated by an *M-C* pair is called a collection, for simplicity. Let d be the dimension of *MCS* in a faulty H_n. Several *MCS*s in H_n may share a common *COS*. It can be shown easily [16] that the total number of *MCS*s that share a common *COS* does not exceed $(n - d)$. On the other hand, a $(d + 1)$-dimensional subcube may involve several *M-C* pairs in H_n, and the total number of *M-C* pairs involved in a $(d + 1)$-dimensional subcube can be proved [16] to be no more than $(d + 1)$. If a $(d + 1)$-dimensional subcub involves multiple *M-C* pairs, the collections derived from the respective *M-C* pairs are of the same size, so it suffices to select only one of the *M-C* pairs for generating a collection. As a result, *only a portion* of all *M-C* pairs in a faulty H_n are selected for generating collections in order to reduce search space. The following theorem reveals a bound on the number of selected *M-C* pairs in H_n.

THEOREM. *Let d be the dimension of MCS in a faulty H_n. The maximum number of the selected M-C pairs is* $\frac{1}{n-d}\binom{n}{d+1}2^{n-d-1}$

PROOF. Since d is the dimension of *MCS*, a maximum incomplete subcube actually resides within a $(d + 1)$-dimensional subcube. To find maximum incomplete subcubes in the worst case, each one of the $\binom{n}{d+1}2^{n-d-1}$ $(d + 1)$-dimensional subcubes needs to be considered. Without loss of generality, let's consider an *arbitrary* *M-C* pair from an *arbitrary* $(d + 1)$-dimensional subcube. Besides the *MCS* of the *M-C* pair considered, there are $(n - d - 1)$-dimensional subcubes that are lateral to the *COS* of the *M-C* pair with $\delta = 1$. If all these subcubes are of *MCS*, then $(n - d)$ *MCS*s (including the *MCS* under consideration) share the same *COS*. Since it is sufficient to select only one *M-C* pair from each $(d + 1)$-dimensional subcube, we have considered $(n - d)$ $(d + 1)$-dimensional subcubes (corresponding, respectively, to the $(n - d)$ *M-C* pairs) simply by selecting an *M-C* pair. On the other hand, if some of these subcubes are not of *MCS*, then, the subcubes formed by the *COS* and those non-MCS subcubes cannot yield maximum incomplete subcubes. Therefore, it is concluded that the number of the *M-C* pairs selected for generating collections is at most $\frac{1}{n-d}\binom{n}{d+1}2^{n-d-1}$. □

3.2.2 Efficient Algorithm

As observed earlier, each maximum collection under consideration comes from an *M-C* pair in a faulty H_n. We first identify all *MCS*s in H_n using Algorithm 1. All possible *M-C* pairs (where an *MCS* may have more than one *COS*) are recognized, but only a subset of the *M-C* pairs are selected for further examination, based on the above properties. In the next step, for each one of the selected *M-C* pairs, the *MCS* itself is assigned to the collection currently under search, with the *COS* treated as a faulty subcube, in which Algorithm 1 is employed to identify *MCS*s *resided* in it. All possible *M-C* pairs resided in the *COS* are identified but only a subset of them are selected for further examination. For each chosen *M-C* pair, the *MCS* itself again is appended to the collection, with its *COS* being examined to determine *M-C* pairs *resided* in it. This process repeats until no further subcube can be added to the collection. When the current search terminates, the size of the generated collection is compared to the size of the maximum collections generated so far, and the bigger one is kept as the current maximum size. If the newly generated collection has the same size, it is kept along with the previously generated maximum collections. As a result, the search carried out for an *M-C* pair in H_n gives rise to one or multiple collections. The examination of other *M-C* pairs in H_n is carried out one by one until all candidate *M-C* pairs are exhausted. After completion of the entire search, the collections obtained are the maximum incomplete subcubes. Algorithm 2 gives the procedure for identifying maximum incomplete subcubes. While Algorithm 2 may identify multiple maximum incomplete subcubes, it is not guaranteed to discover all the maximum incomplete subcubes existing in a faulty hypercube.

Algorithm 2: Finding maximum incomplete subcubes in the presence of faulty nodes

Input: The addresses of m faulty nodes in a faulty H_n;
Output: Maximum incomplete subcubes;

```
find MCSs (using Algorithm 1);
recognize all M-C pairs and select only a subset of them
  based on the properties given in Sec. 4.1;
maxsize = 0;
for each selected M-C pair do
    assign the MCS to the collection currently in search;
    find all the MCSs within the COS (using Algorithm 1);
    recognize all M-C pairs and select only a subset of them Φ(i) (= a set
      of M-C pairs selected at current step i) based on the above properties;
    repeat the last three statements for each M-C pair ∈ Φ(i) until
      no MCS can be found;
    compute size, the size of the newly generated collection;
    if size > maxsize then
        maxsize = size;
        keep the generated collection as the maximum collection;
    else if size = maxsize then
        the generated collection is kept along with the existing maximum
          collections;
    end if
end for
```

As an example, Fig. 3 illustrates the search details of finding maximum incomplete subcubes after four faulty nodes arise in H_4, as shown in Fig. 1. The three-dimensional subcube 0*** involves two *M-C* pairs, (01**, 00**) and (0**1, 0**0), in the presence of the given faults. Only one of the two pairs is selected (e.g., (0**1, 0**0) is chosen in Fig. 3) to generate a collection. Similarly, ***1 involves three *M-C* pairs, (0**1, 1**1), (*0*1, *1*1), and (**01, **11), and only one pair will be selected (e.g., (*0*1, *1*1) is chosen in Fig. 3). Note that only those three-dimensional subcubes that involve multiple *M-C* pairs are given in Fig. 3. Two *MCS*s, 01** and 10**, share common *COS*s, 00** and 11**, yielding four *M-C* pairs, (01**, 00**), (10**, 00**), (01**, 11**), and (10**, 11**). Among them, the first two pairs may be ignored since the collections corresponding to these two pairs are of the same size as that corresponding to pair (0**1, 0**0), which has already been considered. For the remaining two pairs, an arbitrary one can be selected ((01**, 11**) is chosen in Fig. 3). In addition to the selected pairs, another two pairs at the two-dimensional level, (**01, **00) and (*0*1, *0*0), have to be selected to complete the search at this level, yielding a total of five pairs selected at this level. The selected pairs are shown by bold ellipses in Fig. 3.

Corresponding to the five selected pairs, five *COS*s are examined for resided *MCS*s. Among them, 11** contains two fault-free nodes, and either one of the two can contribute to the collection, yielding two collections of size 5. Similarly, **00 involves two fault-free nodes and results in two collections of size 5 as well. 0**0 contains a single *MCS*, 01*0, whereas *0*0 contains an *MCS*, 10*0. These two *MCS*s share a common *COS*, 00*0, which contains no *MCS*, resulting in a collection of size 6. Finally, *1*1 involves two *M-C* pairs, (*101, *111) and (01*1, 11*1), where both *111 and 11*1 contain one fault-free node each, yielding a collection of size 7, {*0*1, *101, 0111}, which is the maximum one. There are totally two *M-C* pairs selected at the one-dimensional level.

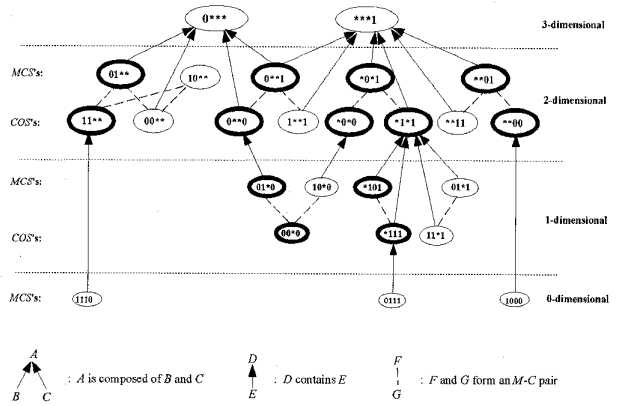

Fig. 3. Illustration of the search procedure for the example given in Fig. 1.

3.2.3 Time Complexity

The entire search time is equal to the sum of the time taken at each step of the search procedure. At each step, no more than $\frac{1}{n-k_i}\binom{n}{k_i+1}2^{n-k_i-1}$ M-C pairs are considered according to the preceding theorem, and thus at most $\frac{1}{n-k_i}\binom{n}{k_i+1}2^{n-k_i-1}$ COSs are examined, each requiring time at most $m_i^2 2^{k_i}$ for examination (using Algorithm 1), where k_i is the dimension of the MCS found at the ith search step and m_i is the maximum number of faulty nodes involved in a corresponding COS. Therefore, we have the following worst case time complexity

$$\begin{aligned}
T(n) &= m^2 2^n + \frac{1}{n-k_1}\binom{n}{k_1+1}2^{n-k_1-1}\cdot m_1^2 2^{k_1} \\
&\quad + \frac{1}{n-k_2}\binom{n}{k_2+1}2^{n-k_2-1}\cdot m_2^2 2^{k_2} + \cdots \\
&< m^2 2^n + \frac{1}{2(n-k_1)} m_1^2 2^n \left\{\binom{n}{k_i+1} + \binom{n}{k_2+1} + \cdots\right\} \\
&< m^2 2^n + \frac{1}{2n} m^2 2^n \sum_{i=0}^{n}\binom{n}{i} \quad (\text{as } n > k_1 > k_2 > \cdots) \\
&= m^2 2^n + \frac{1}{2n} m^2 2^{2n}.
\end{aligned}$$

Since the second term dominates the above expression, the worst case time complexity is bounded above by

$$O\left(\frac{m^2}{n}2^{2n}\right) = O\left(\frac{m^2 N^2}{n}\right),$$

where N $(= 2^n)$ is the system size.

The search for a collection in Algorithm 2 terminates when no further resided MCS can be found. Actually the search process may terminate when the number of faulty nodes involved in the current COS is no greater than 3, because the collection currently in search is then determined. This idea helps to reduce the time complexity of Algorithm 2 by a constant factor, and it can be generalized as follows (a proof of this generalization can be found in [16]): Let m be the number of faulty nodes in H_n and S_k be the smallest subcube that contains all the faulty nodes. If $1 \le m \le 3$, the maximum incomplete subcube is determined directly, with its size expressed as

$$maxsize = \begin{cases} 2^n - 1, & \text{if } k = 0; \\ 2^n - 2^k + 2^{k-1} - 1, & \text{if } 1 \le k \le n. \end{cases}$$

4 Extension to Including Faulty Links

This section explains how Algorithm 2 is extended to deal with the system involving both faulty nodes and faulty links.

4.1 Finding Maximum Complete Subcubes

A link is labeled by an n-tuple $\{0, 1, *\}^n$, which contains exactly one "*" at a coordinate position. Two end nodes of the link can be uniquely determined by assigning "0" and "1" to the "*" coordinate. It is observed that a faulty link which does not reside *inside* a subcube has no damaging impact on the subcube. This suggests a modification to Algorithm 1 to deal with the case where fault set F involves faulty nodes and links, as follows: While searching for combination(s) absent from a set of bit positions, faulty links which contain "*" at that set of bit positions are ignored. If a combination is covered by F at a given $(n-k)$ bit positions, the subcube with its label composed of the combination at those given $(n-k)$ bit positions and "*" at the remaining k positions, contains at least one fault (either a faulty link or a faulty node). On the other hand, if a combination at any $(n-k)$ bit positions is absent, a fault-free subcube is identified by assigning the combination to those particular $(n-k)$ bit positions and "*" to the remaining positions.

4.1.1 Time Complexity of Modified Algorithm

Let m' be the total number of faulty nodes and faulty links in a system. It is indicated in [7] that the minimum number of faulty links necessary to destroy $(n-d)$ dimensions is no less than that of faulty nodes necessary to destroy the same number of dimensions. The upper bound used in Section 3.1 thus remains valid for m', i.e., $2^{n-d-1} \le m'$, leading to the worst case time complexity of the modified algorithm equal to $O((m')^2 N)$, where N $(= 2^n)$ is the system size.

4.2 Identifying Maximum Incomplete Subcubes

As mentioned before, the algorithm for finding maximum incomplete subcubes in a faulty H_n builds on the algorithm for identifying all MCSs in H_n. Hence, Algorithm 2 can be extended by replacing Algorithm 1 with the modified algorithm described in the previous subsection, to handle the system involving both faulty nodes and faulty links. In addition, special attention is required for each faulty link with one of its two end nodes involved in the MCS of a selected M-C pair, and the other in the COS of the pair. To prevent such a faulty link from being included in any candidate incomplete subcube, the end node in the COS should be regarded as a faulty node (in order to reflect the effect of the faulty link). This minor modification to Algorithm 2 does not affect the overall time complexity of the modified algorithm. Thus, time complexity of the modified algorithm remains to be $O((m')^2 N^2)$, where N $(= 2^n)$ is the system size.

5 Concluding Remarks

In this paper, we have shown how to reconfigure a faulty hypercube into maximum incomplete subcubes. An algorithm (Algorithm 1) for identifying all the maximum complete subcubes efficiently in the presence of faulty nodes has been presented and compared to prior known schemes [8], [9], [10]. Based on Algorithm 1, an algorithm (Algorithm 2) for determining maximum incomplete subcubes in a hypercube with faulty nodes is developed. The algorithm is made efficient by taking advantage of some properties of faulty hypercubes to reduce search complexity significantly. Both Algorithm 1 and Algorithm 2 are extended to deal with the system involving both faulty nodes and faulty links.

Extensive simulation was carried out to compare the size differences of reconfigured complete subcubes and reconfigured maximum incomplete subcubes in given faulty hypercubes. The simulation results demonstrate that a maximum incomplete subcube ob-

tained indeed is considerably larger than its maximum complete counterpart. In fact, the maximum incomplete subcubes always have the average size of roughly two times as large as that of their complete counterparts [16], clearly exhibiting the advantage of reconfiguring a faulty hypercube into an incomplete system. While our algorithms for finding maximum incomplete subcubes could arrive at multiple maximum incomplete subcubes, it is not guaranteed to identify all the maximum incomplete subcubes present in a faulty hypercube.

Once a maximum incomplete subcube is determined in a faulty hypercube, an initialization procedure is invoked to relabel nodes in the subcube as a part of reconfiguration. The same routing and broadcasting algorithms can be used after nodes are relabeled in the reconfigured subcube, if the router at each node is equipped with a crossbar for switching. However, if the router follows a fixed dimensional order in selecting ports, the dimension numbers assigned to ports at each node have to alter after node relabeling to keep routing and broadcasting unchanged.

REFERENCES

[1] S.-C. Chau and A.L. Liestman, "A Proposal for a Fault-Tolerant Binary Hypercube Architecture," *Proc. 19th Int'l Symp. Fault-Tolerant Computing*, pp. 323-330, June 1989.

[2] T.C. Lee and J.P. Hayes, "Design of Gracefully Degradable Hypercube-Connected Systems," *J. Parallel and Distributed Computing*, vol. 14, pp. 390-401, 1992.

[3] J. Bruck, R. Cypher, and C.-T. Ho, "Wildcard Dimensions, Coding Theory and Fault-Tolerant Meshes and Hypercubes," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing*, pp. 260-267, June 1993.

[4] J. Bruck, R. Cypher, and C.-T. Ho, "Fault-Tolerant Meshes and Hypercubes with Minimal Numbers of Spares," *IEEE Trans. Computers*, vol. 42, no. 9, pp. 1,089-1,104, Sept. 1993.

[5] B. Aiello and T. Leighton, "Coding Theory, Hypercube Embeddings, and Fault Tolerance," *Proc. Third Ann. ACM Symp. Parallel Algorithms and Architectures*, pp. 125-136, July 1991.

[6] J. Bruck, R. Cypher, and D. Soroker, "Tolerating Faults in Hypercubes Using Subcube Partitioning," *IEEE Trans. Computers*, vol. 41, no. 5, pp. 599-605, May 1992.

[7] B. Becker and H.-U. Simon, "How Robust Is the n-Cube?" *Proc. IEEE 27th Symp. Foundations of Computer Science*, pp. 283-291, Oct. 1986.

[8] F. Ozguner and C. Aykanat, "A Reconfiguration Algorithm for Fault Tolerance in a Hypercube Multiprocessor," *Information Processing Letters*, vol. 29, pp. 247-254, Nov. 1988.

[9] M.A. Sridhar and C.S. Raghavendra, "On Finding Maximal Subcubes in Residual Hypercubes," *Proc. Second IEEE Symp. Parallel and Distributed Processing*, pp. 870-873, Dec. 1990.

[10] S. Latifi, "Distributed Subcube Identification Algorithms for Reliable Hypercubes," *Information Processing Letters*, vol. 38, pp. 315-321, June 1991.

[11] H.P. Katseff, "Incomplete Hypercubes," *IEEE Trans. Computers*, vol. 37, no. 5, pp. 604-608, May 1988.

[12] N.-F. Tzeng, "Empirical Evaluation of Incomplete Hypercube Systems," *Proc. 22nd Int'l Conf. Parallel Processing*, vol. I, pp. 96-99, Aug. 1993.

[13] N.-F. Tzeng and H.-L. Chen, "Structural and Tree Embedding Aspects of Incomplete Hypercubes," *IEEE Trans. Computers*, vol. 43, no. 12, pp. 1,434-1,439, Dec. 1994.

[14] M.-S. Chen and K.G. Shin, "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 39, no. 12, pp. 1,406-1,416, Dec. 1990.

[15] P. Ramanathan and K.G. Shin, "Reliable Broadcast in Hypercube Multicomputers," *IEEE Trans. Computers*, vol. 37, no. 12, pp. 1,654-1,657, Dec. 1988.

[16] N-F. Tzeng and G. Lin, "Identifying Maximal Incomplete Subcubes in Faulty Hypercubes," *Proc. Seventh Int'l Conf. Parallel and Distributed Computing Systems*, pp. 186-193, Oct. 1994.

A New Synchronizer Design

Jacqueline Walker, *Student Member, IEEE*, and

Antonio Cantoni, *Senior Member, IEEE*

Abstract—A new synchronizer design is presented. Current synchronizer designs have certain disadvantages, both in characterization and in the tradeoff between settling time and sampling rate, which are overcome in the new design. Two possible implementations of the synchronizer are discussed.

Index Terms—Metastability, synchronizer, asynchronous, synchronization, synchronizer design, flip-flop, synchronous digital systems.

1 INTRODUCTION

IN digital system design, the need to introduce an asynchronous signal into a synchronous system arises quite frequently. A common occurrence is in a digital system that samples a digital signal generated by an external autonomous device [8], [12]. An asynchronous input to a synchronous system may come from another synchronous system [3] or arise in a globally asynchronous, locally synchronous system when synchronous subsystems need to communicate [1]. An equivalent problem is the need to arbitrate between competing requests for a shared resource [2], [5] or to choose between competing commands which arise from independent sources [7].

It has become well known that asynchronous inputs in synchronous systems can lead to system failures due to bistable devices entering metastable states. Metastable states may occur in bistable devices when the timing constraints of the device are not observed [4], [6], [9], as can be the case with an asynchronous input which can change at any time with respect to the clock edges of the synchronous system. In the metastable state, the output of the device does not reach either of the valid logic levels but hovers between the two for a time that is long compared with the normal timing delays of the device [9] or may even oscillate [5], [10]. System failure occurs because of the inconsistent response of other devices within the system to the metastable output [6].

To overcome the problem of metastable failure within systems, synchronizers have been adopted as an interface between the asynchronous input and the synchronous system. The synchronizer samples the asynchronous input at the rate of the system clock. The output of the synchronizer is thus synchronous with the rest of the system. However, the use of a synchronizer does not eliminate the possibility of metastable failure, it is only possible to limit its occurrence to within the synchronizer and thereby minimize its effect on the system.

Modeling of metastability in bistable devices has shown that the probability of metastable failure, when the timing of data edge transitions is modeled by a Poisson process, decreases exponentially with the length of time allowed for the output to settle [6], [11], [13]. As a result the performance of a synchronizer can be substantially improved by increasing the delay before the output is accessed [6]. Thus when designing synchronizers the aim should be to provide an adequate settling time.

• *The authors are with the Australian Telecommunications Research Institute, Curtin University of Technology, Building 314, Bentley, WA, 6102, Australia. E-mail: walker@atri.curtin.edu.au.*

Manuscript received Sept. 19, 1994; revised Aug. 28, 1995.
For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C96040.