so on, until $x_{1,k} = x_{2,k} = \cdots = x_{n,k}$. It is assumed that all 2^k combinations of each input vector can be received by the comparator.

Put inverters on bit lines of half input vectors as follows.

$$\bar{V}_2^* = (\bar{x}_{2,1}, \cdots, \bar{x}_{2,k})$$

$$\cdots \quad \cdots$$

$$\bar{V}_{2m}^* = (\bar{x}_{2m,1}, \cdots, \bar{x}_{2m,k})$$

where $m = [n/2]$, the minimum integer larger than or equal to $n/2$. Therefore,

$$V_1 = (V_1^*, \bar{V}_2^*)$$

$$= (x_{1,1}, \bar{x}_{2,1}, \cdots, x_{1,k}, \bar{x}_{2,k})$$

$$\cdots \quad \cdots$$

$$V_m = (V_{2m-1}^*, \bar{V}_{2m}^*)$$

$$= (x_{2m-1,1}, \bar{x}_{2m,1}, \cdots, x_{2m-1,k}, \bar{x}_{2m,k})$$

are two-rail encoded inputs. If n is even, $2m = n$. If n is odd, let $V_{2m}^* = V_1^*$.

Applying the design procedure given in Section III to design the K-unit TSCC as a comparator, we can get

$$z_1 = f(x_{1,1}, \cdots, x_{1,k})$$

$$z_2 = f_2(x_{3,1}, \cdots, x_{3,k})$$

$$\cdots \quad \cdots$$

$$z_m = f_m(x_{2m-1,1}, \cdots, x_{2m-1,k})$$

when $m \leq k$. It is easy to verify that B produces all combinations of $(z_1, \cdots, z_m)$. As a result of Theorem 3, we have the following theorem.

Theorem 4: The K-unit network implementing

$$F = f(f(x_{1,1}, \cdots, x_{1,k}), f_2(x_{3,1}, \cdots, x_{3,k}), \cdots, f_m(x_{2m-1,1}, \cdots, x_{2m-1,k}))$$

is a TSC comparator.

The comparison of the K-unit network to the multipattern comparator proposed in [10] shows that the advantage of the K-unit TSC comparator is not only the reduction of gate delays, but also the significant reduction of the number of logic gates needed.

V. Conclusions

Totally self-checking circuits have caught our attention in the area of testing and fault-tolerant computing. VLSI technology enables the reality of TSC circuits. TSC circuits using PLA's are very appropriate for VLSI implementation. This correspondence presents a general approach to designing a circuit using PLA's that achieves the TSC goal.

For the functional SFS PLA's, concurrent SFS PLA's with two-rail encoded outputs are suggested to simplify the design of the associated TSCC. A design of an SFS multiplexer illustrates the concurrent SFS PLA.

For the TSC checker, a general design procedure of TSCC is given to meet the requirement that a given codeword set sufficiently exercises the TSCC. A K-unit TSC comparator with an arbitrary number of inputs is a successful application of this kind of TSCC.

Acknowledgment

The authors would like to thank Dr. L. T. Wang and Z. Li for their encouragement and helpful suggestions.

References

[1] D. A. Anderson, "Design of self-checking digital networks using code techniques," TR R-527, Univ. Illinois, 1971.
[2] J. E. Smith and G. Metze, "Strongly fault secure logic networks," *IEEE Trans. Comput.*, vol. C-27, June 1978.
[3] M. Nicolaidis, I. Jansch, and B. Courtois, "Strongly code disjoint checkers," in *Proc. 14th Int. Symp. Fault-Tolerant Comput.*, Kissimmee, FL, 1984.
[4] J. E. Smith, "Design of totally self-checking combinational circuits," Ph.D. dissertation, Rep., R-737, Univ. Illinois, 1976.
[5] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The design of PLAs with concurrent error detection," in *Proc. 14th Int. Symp. Fault-Tolerant Comput.*, Kissimmee, FL, 1982, pp. 303-310.
[6] J. Raiski and V. K. Agarwal, "Testing properties and applications of inverter-free PLA's," in *Proc. 1985 Int. Test Conf.*, Philadelphia, PA, 1985, pp. 500-507.
[7] S. L. Wang and A. Avizienis, "The design of totally self-checking circuits using programmable logic arrays," in *Proc. Ninth Int. Symp. Fault-Tolerant Comput.*, WI, 1979.
[8] J. Khakbaz and E. J. McCluskey, "Concurrent error detection and testing for large PLA's," *IEEE Trans. Electron Devices*, vol. ED-29, no. 4, 1982.
[9] N. K. Jha and J. A. Abraham, "The design of totally self-checking embedded checkers," in *Proc. 14th Int. Symp. Fault-Tolerant Comput.*, Kissimmee, FL, 1984.
[10] J. A. Hughes, E. J. McCluskey, and D. Lu, "Design of totally self-checking comparators with an arbitrary number of inputs," *IEEE Trans. Comput.*, vol. C-33, no. 6, 1984.
[11] Y. Min and Z. Li, "A unified fault model for PLAs," in *Proc. Int. Conf. Circuits Syst.*, Beijing, China, 1985.
[12] Y. Min, "Generating a complete test set for PLAs," in *Proc. First Int. Conf. Comput. Applications*, Beijing, China, 1984.
[13] J. Khakbaz and E. J. McCluskey, "Self-testing embedded code checkers," *IEEE Trans. Comput.*, vol. C-33, Aug. 1984.

Topological Properties of Hypercubes

YOUCEF SAAD AND MARTIN H. SCHULTZ

***Abstract*—The n-dimensional hypercube is a highly concurrent loosely coupled multiprocessor based on the binary n-cube topology. Machines based on the hypercube topology have been advocated as ideal parallel architectures for their powerful interconnection features. In this paper, we examine the hypercube from the graph theory point of view and consider those features that make its connectivity so appealing. Among other things, we propose a theoretical characterization of the n-cube as a graph and show how to map various other topologies into a hypercube.**

***Index Terms*—Binary n-cube, characterization of hypercube graphs, hypercube imbeddings, hypercube networks, hypercube topology.**

I. Introduction

Hypercubes are loosely coupled parallel processors based on the binary n-cube network and introduced under different names (cosmic cube, n-cube, binary n-cube, Boolean n-cube, etc.). A few machines based on the hypercube topology have been experimented in several institutions, see [8] for references, and others are now being built. An

Manuscript received November 5, 1985; revised July 22, 1986. This work was supported in part by ONR Grant N00014-82-K-0184 and in part by a joint study with IBM/Kingston.

The authors are with the Department of Computer Science, Yale University, New Haven, CT 06520.

IEEE Log Number 8716257.

n-cube parallel processor consists of 2^n identical processors, each provided with its own sizable memory, and interconnected with n neighbors.

There are essentially two broad classes of MIMD parallel processor design with a large number of processors, presently competing against each other. The first type of architecture consists of a large number of identical processors interconnected to one another according to some convenient pattern. In this type of machine, there is no shared memory and no global synchronization. Moreover, intercommunication is achieved by message passing and computation is data driven (although some designs incorporate a global bus, this does not constitute the main way of intercommunication). By message passing, it is meant that data or possibly code are transferred from processor A to processor B by traveling across a sequence of nearest neighbor nodes starting with node A and ending with B. Synchronization is driven by data in the sense that computation in some node is performed only when its necessary data are available. Examples include grid networks such as the finite element machine [1], tree machines [4], the cosmic cube [8], and many others. At the border line of this class, one might also include the data flow machines which utilize the same concept of data-driven synchronization but adopt a more dynamic way of circulating data. The main advantage of such architectures, often referred to as ensemble architectures, is the simplicity of their design. The nodes are identical, or are of a few different kinds, and can therefore be fabricated at relatively low cost.

The second important class of parallel processors consists of a set of N identical processors interconnected via a large switching network to N memories. Thus, the memory can be viewed as split into N "banks," and shared between the N processors. Variations on this scheme are numerous, but the essential feature here is the switching network. Examples include the Ultracomputer developed at NYU [5] which uses an omega network. The main advantage of this second configuration is that it enables us to make the data access transparent to the user who may regard data as being held in a large memory which is readily accessible to any processor. This greatly facilitates the programming of the machine but memory conflicts can lead to degraded performance. Also, the network can simulate any of the intercommunication patterns of the first type of architecture. On the other hand, shared memory models cannot easily take advantage of proximity of data in problems where communication is local. Moreover, the switching network becomes exceedingly complex to build as the number of nodes increases. In fact, to connect N nodes, the Ultracomputer requires a total of $O(N \log_2 N)$ identical 2×2 switches. In particular, this raises the problem of reliability as the probability of failure increases proportionally with the number of components. The first models can easily be made fault tolerant by shutting down failing nodes: at the difference with the shared memory models, the decision of shutting down failing nodes and choosing alternate routes is a local one.

As is mentioned above, it is clear that one of the most important advantages of the first class of designs is the ability to exploit particular topologies of problems or algorithms in order to minimize communication costs. Thus, a two-dimensional grid network is perfectly suitable for solving discretized elliptic partial differential equations, e.g., by assigning each grid point to its counterpart in the array, because the iterative methods for solving the resulting linear systems require only nearest neighbor grid-point interaction. This means that if a general purpose ensemble architecture is to be designed, it must have powerful mapping capabilities, i.e., it must be capable of mapping easily many common geometries such as grids or linear arrays. The hypercube is a machine of the first class which has excellent mapping capabilities. This explains in part the growing interest that hypercube-based architectures are currently arousing.

It is the purpose of this paper to study the topological properties of the hypercube. We will first derive some simple properties of the hypercube regarded as a graph and will propose a theorem that will describe an n-cube by a few characteristic properties. Mapping other topologies is very important for designing efficient algorithms that map perfectly into those topologies. We will consider this problem in detail and show how to map rings, linear arrays, and multidimensional meshes into hypercubes.

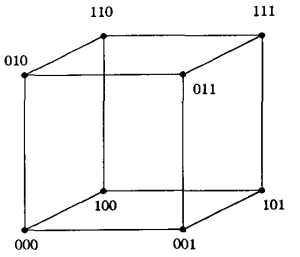

Fig. 1. 3-D view of the 3-cube.

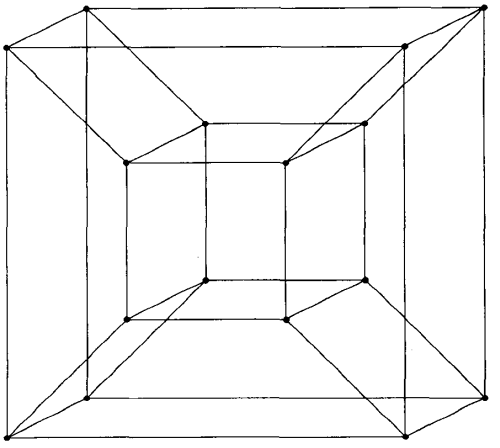

Fig. 2. 3-D view of the 4-cube.

II. The Hypercube Graph and its Basic Properties

In what follows, the hypercube is regarded as a graph and we will often use the terms vertices or nodes interchangeably for the processors they represent. A 3-cube can be represented as an ordinary cube in three dimensions where the vertices are the $8 = 2^3$ nodes of the 3-cube, see Fig. 1. More generally, one can construct an n-cube as follows. First, the 2^n nodes are labeled by the 2^n binary numbers from 0 to $2^n - 1$. Then a link between two nodes is drawn if and only if their binary numbers differ by one and only one bit. In this paper, we will refer to the hypercube graph, or hypercube, as the graph thus defined.

Definition 2.1: An n-cube graph is an undirected graph consisting of $k = 2^n$ vertices labeled from 0 to $2^n - 1$ and such that there is an edge between any two vertices if and only if the binary representations of their labels differ by one and only one bit.

The first important property of the n-cube is that it can be constructed recursively from lower dimensional cubes. More precisely, consider two identical $(n - 1)$-cubes whose vertices are numbered likewise from 0 to 2^{n-1}. By joining every vertex of the first $(n - 1)$-cube to the vertex of the second having the same number, one obtains an n-cube. Indeed, it suffices to renumber the nodes of the first cube as $0 \wedge a_i$ and those of the second by $1 \wedge a_i$ where a_i is a binary number representing the two similar nodes of the $(n - 1)$-cubes and where $\wedge$ denotes the concatenation of binary numbers. This is illustrated for $n = 4$ in Fig. 2, where a 4-cube is obtained by joining all corners of an inner 3-cube with the corresponding corners of an outer 3-cube. An interesting geometric property of the illustration is that it provides one way of constructing higher dimensional cubes from 3-cubes by simply repeating the above process with more enclosing cubes.

Separating an n-cube into the subgraph of all the nodes whose leading bit is 0 and the subgraph of all the nodes whose leading bit is 1, the two subgraphs are such that each node of the first is connected to one node of the second. If we remove the edges between these two graphs, we get two disjoint $(n-1)$-cubes. This operation of splitting the n-cube into two $(n-1)$-cubes so that the nodes of the two $(n-1)$-cubes are in a one-to-one correspondence will be referred to as tearing. The tearing suggested above gives privilege to the leading bit but there is no particular reason for this. More generally, for a given numbering, tearing simply amounts to separating the graph into two subgraphs obtained by considering all the nodes whose ith bit is 0 and those whose ith bit is 1. This will be referred to as tearing along the ith direction. Since there are n bits, there are also n directions. These simple properties are summarized in the following proposition.

Proposition 2.1: There are n different ways of tearing an n-cube, i.e., of splitting it into two $(n-1)$-subcubes so that their respective vertices are connected in a one-to-one way. Given the labeling of Definition 2.1, each different tearing corresponds to splitting the n-cube graph into two subgraphs: one whose node labels have a one in position i and one whose node labels have a zero in position i.

The following result tells us how many ways there for labeling an n-cube.

Proposition 2.2: There are $n!2^n$ different ways in which the 2^n nodes of an n-cube can be numbered so as to conform with Definition 2.1.

Proof: The proof is by induction. The result is trivial for $n = 0$. Assume it is true for $n-1$. To number the nodes of the n-cube, we will first choose their leading bits to be either zero or one. To do so, we will tear the n-cube into two $n-1$ cubes (there are n different ways to do it). Then we number the nodes of the first $(n-1)$-cube (in $(n-1)!2^{n-1}$ different ways) and add a one as leading bit, and the nodes of the second cube in the same way and then add a zero as leading bit. A second numbering is obtained by reversing the bits zero and one. We thus obtain a total of

$$n[(n-1)!2^{n-1}+(n-1)!2^{n-1}]=n!2^n$$

different numberings of the vertices of the n-cube. ■

Note that without the restriction that the numbering must conform to Definition 2.1, we would have a total of $(2^n)!$ different ways of numbering 2^n different vertices, a much larger number than that of Proposition 2.2.

Proposition 2.1 has the following important consequence.

Proposition 2.3: Any two adjacent nodes A and B of an n-cube are such that the nodes adjacent to A and those adjacent to B are connected in a one-to-one fashion.

Proof: Since the nodes considered are neighbors, their node numbers A and B differ by one bit, say the ith bit. Let us tear the n-cube along the ith direction. Then the neighbors of A and those of B can be put in a one-to-one correspondence by mapping a node whose label has a one in its ith position to the one whose label has a zero in its ith position. ■

We can define the parity of a node to be positive if the number of ones in its binary label is even and negative otherwise. It is clear that neighboring nodes have opposite parity. The following result follows from this simple property.

Proposition 2.4: There are no cycles of odd length in an n-cube.

Proof: Consider a cycle $A_1, A_2, \cdots, A_t$, with $A_1 = A_t$. As we travel from node A_i to node A_{i+1}, $1 \le i \le t-1$, the parity changes. Since $A_1 = A_t$, there must be an even number of changes, i.e., the length of the cycle is necessarily even. ■

Given two nodes of an n-cube, there is always a path between them. One way of reaching node B from node A is to modify the bits of A one at a time in order to transform the binary number A into B. Each time one bit is changed, this means that we have crossed one edge. This provides a simple way of constructing a path of length *at most* n between any two vertices of an n-cube. Therefore, recalling that the diameter of a graph is the maximum distance between any two nodes in the graph, we can state Proposition 2.5.

Proposition 2.5: The n-cube is a connected graph of diameter n.

The above propositions establish some basic properties of the n-cube as a graph. The important question we would like to answer next is how to recognize a hypercube in a simple way, i.e., how to characterize an n-cube by a few simple rules. As an example of application, looking at a four by four grid with nearest neighbor connection and wraparound at the edges (of the grid) one might ask whether the corresponding graph is an n-cube, i.e., whether its 16 nodes can be numbered according to the rule of Definition 2.1. It is clear that without the wraparound at the edges, the grid cannot be a cube since all the vertices of an n-cube have the same degree. The next result will answer this question.

Theorem 2.1: A graph $G = (V, E)$ is an n-cube if and only if

1) V has 2^n vertices;
2) every vertex has degree n;
3) G is connected;
4) any two adjacent nodes A and B are such that the nodes adjacent to A and those adjacent to B are linked in a one-to-one fashion.

Proof: Necessary Condition: Conditions 1–4 are clearly satisfied for an n-cube as a result of the definition and some of the previous propositions.

Sufficient Condition: The proof is by induction. It is clear that the property is true for $n = 1$. Assume that it is true for $n-1$, i.e., that any graph having 2^{n-1} nodes satisfying Properties 1–4 is an $(n-1)$-cube. The proof consists in separating the graph in two subgraphs each of which has the same properties for $n-1$.

Consider any two adjacent nodes R (for red) and B (for black) of the graph. According to Property 4, the neighbors of R and those of B are connected in a one-to-one fashion. We can, therefore, color the neighbors of the red nodes (except the one node which is already black) in red and the neighbors of the black node (except the one node which is already red) in black. This process can be continued until exhaustion of all links. We refer to two nodes of different colors that are linked by an edge as two opposite nodes. After this is done we have the following.

a) All the nodes have been colored either B or R. This is because the graph is connected and therefore there is a path between the original node R (or B) to any node.

b) Exactly half the nodes have the color red and the other half have the color black, because all the B nodes and the R nodes are linked in a one-to-one fashion.

c) It is clear that the R nodes constitute a connected graph, since, by construction, each node is connected to the original R node. The same property holds for the black nodes.

d) Consider the two subgraphs obtained by removing all red–black links. Thus, each node loses exactly one edge, i.e., its degree is $(n-1)$. (In the graph theory terminology, the set of R–B edges is called a cut set.) Then Property 4 is satisfied for the subgraph of the red nodes (resp., the black nodes).

e) Because of Property 4, and by construction, two red nodes are adjacent if and only their black opposites are adjacent.

By the induction hypothesis and by b), c), d), and e), the subgraph of the red nodes is an $(n-1)$-cube. Now label the red nodes according to the definition and use the same labeling for the black nodes opposite to them [we can use the same labeling thanks to e)]. Adding the bit zero in front of the red nodes and the bit one in front of the black nodes, we obtain a labeling of the nodes of the initial graph. ■

A consequence of the theorem is that the 4×4 grid with wraparound at the edges (often referred to as the torus) is a 4-cube. A generalization is the $4 \times 4 \times 4$ grid in three dimensions with again wraparound at the edges. From the theorem, this is nothing but a 6-cube. Thus, these mappings provide simple three-dimensional geometric representations of n-cubes when $n \le 6$.

III. Distances and Paths in Hypercubes

Any multiprocessor system should allow for its processors to exchange data between all of its nodes. Let A and B be any two nodes of the n-cube and consider the problem of sending data from node A to node B. The way in which this is achieved in ensemble

architectures is to move the data (ideally in packets) along a path from A to B crossing a (possibly small) number of processors. By definition, the length of a path between two nodes is simply the number of edges of the path. As was already mentioned in Section II, there exists a path of length at most n between any two nodes. To reach B from A, it suffices to cross successively the nodes whose labels are those obtained by modifying the bits of A one by one in order to transform A into B. Assuming that A and B differ only in i bits, i.e., that their Hamming distance is $H(A, B) = i$, the length of the path will be i. Clearly, there is no path of smaller length between the nodes A and B. This elementary result can be formalized as follows.

Proposition 3.1: The minimum distance between the nodes A and B is equal to the number of bits that differ between A and B, i.e., to the Hamming distance $H(A, B)$.

For future reference we would like to write down explicitly one of the paths suggested by the proof of the above proposition. This path corresponds to correcting the first different bit in A and B, then the second, and so on to the last bit different in A and B. Let $A \equiv a_1a_2 \cdots a_n$ and $B \equiv b_1b_2 \cdots b_n$ be the labels of A and B where a_i and b_i are the bits zero or one. For convenience we assume without loss of generality that A and B differ in their i leading bits. Then one path from A to B is the following

$$\begin{aligned} A = \text{node } 0 &= a_1a_2a_3 \cdots a_ia_{i+1} \cdots a_n; \\ \text{node } 1 &= b_1a_2a_3 \cdots a_ia_{i+1} \cdots a_n; \\ \text{node } 2 &= b_1b_2a_3 \cdots a_ia_{i+1} \cdots a_n; \\ &\cdots \cdots \\ B = \text{node } i &= b_1b_2 \cdots b_ia_{i+1} \cdots a_n. \end{aligned}$$

The generalization to the case where the different bits in A and B are not necessarily the leading ones is straightforward.

One important question we would like to address is whether there are different paths between A and B. The existence of such paths might be useful for speeding up transfers of large amounts of data between two nodes. It also provides a way of selecting alternative routes in case a given node in a path is failing [3]. In order for this to be possible, the paths must not cross each other, i.e., they must not have common nodes, except for nodes A and B. We will refer to such paths as node-disjoint paths or *parallel paths*. So the above question can be reformulated as follows: *how many parallel paths are there between any two nodes A and B*?

A simple look at the above path between A and B reveals that there is no reason why to start by correcting the first different bit. More generally, assuming again that the i bits different in A and B are in front, one might start correcting the jth bit, where $1 \le j \le i$, then the $(j+1)$st bit, and so forth until the ith bit is reached, after which we correct, in turn, bits $1, 2, \cdots (j-1)$. We can thus define i different paths and number them from $j = 1$ to $j = i$. It is easy to prove that any two such paths are parallel. Indeed, the label h, of the node X_h of any path $X_0, X_1, \cdots X_h, \cdots, X_i$, (with $X_0 \equiv A$) of the above i paths, differs from the label of A in exactly h bits. By construction, any two different paths starting the correction in positions j_0 and j_1, respectively, cannot reach the same node X_h in the same number of steps. Also, they cannot reach this same node in two different numbers of steps, otherwise one path would correct A into X_h in changing l_1 bits while the other will achieve the same result in changing l_2 bits with $l_1 \neq l_2$, which is a contradiction. Therefore, we can state the following proposition.

Proposition 3.2: Let A, B be any two nodes and assume that $H(A, B) < n$. Then there are $H(A, B)$ parallel paths of length $H(A, B)$ between the nodes A and B.

Note that the choice of a set of $i = H(A, B)$ parallel paths is by no means unique. Also observe that when $i = n$, the result is optimal in that we can use the maximum allowable number of paths leaving from node A, since the degree of any node is n. We would like now to improve the above result by showing that if we relax the restriction that the length must be i, then as many as n parallel paths can be found even for the case $i < n$. This is important as it will allow us to use the *full bandwidth of the multiprocessor* for data transfer operations between two given processors.

Proposition 3.3: Let A, B be any two nodes of an n-cube and assume that $H(A, B) < n$. Then there are n parallel paths between A and B. Moreover, the length of each path is at most $H(A, B) + 2$.

Proof: In addition to the i paths described prior to Proposition 3.2 and numbered from 1 to i, consider the paths which we will number from $j = i + 1$ to n obtained as follows. First modify the bit a_j into its complement $\bar{a}_j$. Thus, the additional paths start as their first node the node

$$\text{node } 1\text{: } \hat{a}^{(j)} = a_1a_2 \cdots a_ia_{i+1} \cdots \bar{a}_ja_{j+1} \cdots a_n.$$

Then correct bits 1 through i according to *one of the* i paths of the previous proposition to reach, after i steps, the node

$$\text{node } i+1\text{: } \hat{b}^{(j)} = b_1b_2 \cdots b_ib_{i+1} \cdots \bar{a}_ja_{j+1} \cdots a_n.$$

Finally, remodify the bit $\bar{a}_j$ into a_j to reach the final destination B. It is clear by construction that the additional paths thus defined will never cross each other and that they will not cross any of the initial i paths. Moreover, the length of each of the additional paths is $i + 2$. ■

Note that the constructive proof given above yields i paths of length i each and $n - i$ paths of length $i + 2$ each but there are generally more than just $n - i$ paths of length $i + 2$. What the proof indicates is that the first i paths do not use all possible tearings of the cube. The additional paths exploit the unused $(n - i)$-cubes corresponding to the bits in labels of nodes A and B which agree.

IV. Mapping Other Geometries into Hypercubes

In this section, we will be concerned with the problem of mapping other topologies (rings and meshes) into the hypercube. What is meant by mapping other geometries is the following. We are given some graph $G = (V, E)$ having no more than 2^n vertices and we would like to assign the vertices of the graph into the nodes of the n-cube so that every adjacent vertex of the graph belongs to neighboring nodes of the n-cube. There are mainly two different reasons why such mappings are important.

1) Some algorithm may be developed for another architecture for which it fits perfectly. Then one might wish to implement the same algorithm with little additional programming effort. If the original architecture can be mapped into the hypercube, this will be easy to achieve.

2) A given problem may have a well-defined structure which leads to a particular pattern of communication. Mapping the structure may result in substantial savings in communication time. The best example is that of mesh geometries that arise from the dicretization of elliptic partial differential equations in one, two, or three dimensions. Most iterative methods for solving elliptic PDE's require only local communication, i.e., communication between mesh points that are neighbors. If the mesh is perfectly mapped into the cube, then only local communication will be required between the nodes of the hypercube thus resulting in important savings in transfer times.

In this section, we consider mapping ring and grid structures into hypercubes.

A. Mapping Rings and Linear Arrays into Hypercubes

Given a ring-structured graph of 2^n vertices, consider the problem of assigning its vertices into the nodes of a hypercube in such a way as to preserve the proximity property, i.e., so that any two adjacent vertices belong to neighbor nodes. Another way of viewing this problem is that we are seeking a cycle of length $N = 2^n$ that crosses each node once and only once. In graph theory terminology, we are looking for a Hamiltonian circuit in a hypercube.

If we number the nodes of a hypercube according to Definition 2.1, i.e., so that two neighbor nodes differ by one and only one bit, a Hamiltonian circuit simply represents a sequence of n-bit binary numbers such that any two successive numbers have only one

different bit and so that all binary numbers having n bits are represented. Binary sequences with these properties are called Gray codes, and have been extensively studied in coding theory, see, e.g., [6].

There are many different ways in which Gray codes can be generated but the best known method leading to the so-called binary reflected Gray code is as follows. One starts with the sequence of the two 1-bit numbers 0 and 1. This is a 1-bit Gray code. To build a 2-bit Gray code, take the same sequence and insert a zero in front of each number, then take the sequence in *reverse order* and insert a one in front of each number. In other words, we get the sequence

$$G_2 = \{00, 01, 11, 10\}.$$

We can then repeat the process to build a 3-bit Gray code by taking the above sequence inserting a zero in front, then taking the reverse sequence and inserting a one in front:

$$G_3 = \{000, 001, 011, 010, 110, 111, 101, 100\}. \quad (4.1)$$

More generally, denoting by G_i^R the sequence obtained from G_i by reversing its order, and by $0G_i$ (resp., $1G_i$) the sequence obtained from G_i by prepending a zero (resp., a one) to each element of the sequence, then Gray codes of arbitrary order can be generated by the recursion

$$G_{n+1} = \{0G_n, 1G_n^R\}. \quad (4.2)$$

It is easy to verify that such sequences are Gray codes [6].

Gray codes allow us to map rings whose lengths are powers of two into hypercubes. Suppose now that we have a ring of arbitrary length l which we would like to map into a hypercube. First observe that the mapping is possible only when l is even since, according to Proposition 2.4, a hypercube does not admit odd cycles. Therefore, assume that $4 \leq l \leq 2^n$. The problem is to find a cycle of length l in the n-cube, where l is even.

Let $m = (l - 2)/2$ and denote by $G_{n-1}(m)$ the partial $(n - 1)$-bit Gray code consisting of the first m elements of G_{n-1}. Then using the above notation, a cycle having the desired property is the following:

$$\{0G_{n-1}(m), 1G_{n-1}(m)^R\}.$$

Observe that when $l = 2^n$, we obtain as a particular case the formula (4.2). We can, therefore, state the following.

Proposition 4.1: A ring of length l can be mapped into the n-cube when l is even and $4 \leq l \leq 2^n$.

Finally, we point out that there is no difficulty in embedding a linear array, instead of a ring, into the n-cube. It suffices to map the nodes of the linear array $P_0, P_1, \cdots, P_l$ of arbitrary length $l \leq 2^n - 1$ successively into the nodes $g_0, g_1, \cdots, g_l$. Given a linear array of arbitrary length l, the smallest dimension n-cube into which it can be mapped is clearly the cube of dimension $n = \lceil \log_2 (l + 1) \rceil$.

B. Mapping Grids into Hypercubes

One of the most attractive properties of the binary n-cube topology is that meshes of arbitrary dimensions can be imbedded in it. This is one of the main reasons for the success of hypercube architectures. Consider an $m_1 \times m_2 \cdots \times m_d$ mesh in the d-dimensional space R^d and assume that the mesh size in each direction is a power of 2, i.e., it is such that $m_i = 2^{p_i}$. Let $n = p_1 + p_2 + \cdots p_d$ and consider the problem of mapping the mesh points into the n-cube, one mesh point per node. Observe that we have just enough nodes to accommodate one mesh point per node.

What is meant by a mapping of the mesh into the cube is an assignment of the mesh points into the nodes of the cube so that *the proximity property is preserved*, i.e., so that two neighbor points of the mesh are assigned to neighbor nodes in the cube. In the case $d = 1$, the problem was solved in the previous section by using Gray codes. We show next how to extend the ideas of the previous section to more than one dimension.

Our argument is best illustrated by an example. Consider a two-dimensional 8×4 mesh, i.e., $d = 2, p_1 = 3, p_2 = 2, n = p_1 + p_2 = 5$. A binary number A of any node of the 5-cube can be regarded as consisting of two parts: its first 3 bits and its last 2 bits, which we write in the form

$$A = b_1 b_2 b_3 c_1 c_2$$

where b_i and c_j are bits zero or one. It is clear from the definition of an n-cube that when the last 2 bits are fixed, then the resulting 2^{p_1} nodes form a p_1-cube (with $p_1 = 3$). Likewise, whenever we fix the first 3 bits we obtain a p_2-cube. The mapping then becomes clear. Choosing a 3-bit Gray code for the x direction and a 2-bit Gray code for the y direction, the point (x_i, y_j) of the mesh is assigned to the node $b_1 b_2 b_3 c_1 c_2$ where $b_1 b_2 b_3$ is the 3-bit Gray code for x_i while $c_1 c_2$ is the 2-bit Gray code for y_j. This mapping is illustrated in Fig. 3 where the binary node number of any grid point is obtained by concatenating its binary x coordinate and its binary y coordinate.

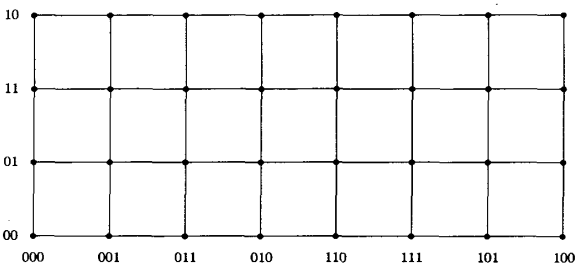

Fig. 3. Two-dimensional Gray code for an 8×4 grid.

Thus, if we call a Gray sequence any subsequence of a Gray code, we observe that any column of grid points forms a Gray sequence and any row of grid points forms a Gray sequence. Therefore, we will refer to the codes defined above as 2-D Gray codes.

Generalizations to higher dimensions are straightforward and one can state the following general theorem.

Theorem 4.1: Any $m_1 \times m_2 \cdots \times m_d$ mesh in the d-dimensional space R^d, where $m_i = 2^{p_i}$ can be mapped into an n-cube where $n = p_1 + p_2 + \cdots p_d$. The numbering of the grid points is any numbering such that its restriction to each ith variable is a Gray sequence.

Note that the assumption that all m_i's be powers of 2 is not essential and the theorem can be generalized by using the remark following Proposition 4.1 concerning mapping general one-dimensional meshes. In particular, it suffices to redefine p_i in the above theorem as $p_i \equiv \lceil \log_2 (m_i) \rceil$.

V. Conclusion

We have shown a few properties of hypercubes that put in light some of the reasons why hypercubes are attractive networks. For reasons of limited space, we have skipped one other important reason which is that trees can also be nicely mapped into hypercubes. These properties are now discussed elsewhere in the literature, for example, see [7], [2], [9]. These very useful mapping properties make the hypercube an ideal network and outweigh some of its inherent drawbacks such as the high cost and complexity of building large hypercubes.

References

[1] L. M. Adams, "Iterative algorithms for large sparse linear systems on parallel computers," Ph.D. dissertation, Univ. Virginia, *Appl. Math.*, 1982. Also available as NASA Contractor Rep. 166027.

[2] S. N. Bhatt and I. C. F. Ipsen, "How to imbed trees in hypercubes," Res. Rep. 443, Dep. Comput. Sci., Yale Univ., 1985.

[3] L. N. Bhuyan and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Trans. Comput.*, vol. C-33, pp. 323–333, 1984.

[4] S. A. Browning, "The tree machine: A highly concurrent computing

environment," Tech. Rep. TR-3760, Dep. Comput. Sci., California Instit. Technol., 1980.
[5] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer—Designing a MIMD shared memory parallel computer," *IEEE Trans. Comput.*, vol. C-32, pp. 175–189, 1983.
[6] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
[7] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," Res. Rep. 389, Dep. Comput. Science, Yale Univ., 1985.
[8] C. L. Seitz, "The cosmic cube," *Commun. ACM*, vol. 28, pp. 22–33, 1985.
[9] A. Y. Wu, "Embedding of tree networks into hypercubes," *J. Parallel Distributed Comput.*, vol. 2, pp. 238–249, 1985.

An Experimental Study to Determine Task Size for Rollback Recovery Systems

SHAMBHU J. UPADHYAYA AND KEWAL K. SALUJA

Abstract—**The effects of using a recovery cache to save the variables of a program are studied. A new optimization model for rollback is formulated to include the effects of a recovery cache in rollback systems. The parameters of the model proposed in this correspondence are the maximum recovery time, the cache size, and the save and load times associated with the task size. We also discuss the results of an experimental study conducted to estimate the parameters of the programs that are critical for arriving at a suitable task size or cache size to minimize the cost of recovery.**

Index Terms—**Program graph, recovery cache, recovery time, rollback recovery, task size.**

I. Introduction

Rollback recovery [1], [2] is an effective technique to recover from transient failures during a program execution. The variables of the program should be protected from any damage that may be caused by the transients in order to have successful recovery. The mechanism of protecting the recovery data, i.e., data pertinent to successful recovery from failures is often termed as state saving [2].

In some applications such as database systems, the state saving is done in secondary storage [1], [2]. Due to the inherent low speed, the use of a secondary store for state saving introduces unnecessarily long delay. Large size buffers may be required if any measures to reduce this delay are employed. Furthermore, loading the saved state back to the main memory from a secondary store during recovery may take large time. Postprocessing of the saved information (such as elimination of multiple copies) can be employed to reduce the time for loading back the saved state. But, this may require an independent processor since any such postprocessing should be done in real-time for rapid recovery.

Alternatively, a special hardware unit can be used for state saving [3], [4]. Lee, Ghani, and Heron [4] have proposed a recovery cache for the PDP-11, for use in recovery block schemes [5]. This concept was extended for application in rollback recovery schemes in [6] and [7]. However, introduction of a recovery cache in rollback systems influences the rollback point insertion strategy and needs to be investigated.

Rollback recovery can be implemented in two ways. In one method, rollback points are inserted at some regular intervals. Some analytic models for rollback recovery and determination of optimal checkpoint intervals are discussed in [2] and [8]. The second method is applicable to those real-time applications in which maximum permissible recovery time may be a critical parameter. For example, in a space-borne system, at some stage of operation a "launch window" may be specified. While the system is in the "launch window" state, it is important that recovery from any failure take place within a specified time because the penalty for nonrecovery can be prohibitively high. Chandy and Ramamoorthy [1] have studied the automatic rollback insertion under the constraint that at every point in the program, recovery should be possible within a specified time.

In the method employing an automatic rollback insertion strategy, a program is analyzed before hand and represented as a sequence of tasks. Then, based on certain parameters of the program, the rollback points are inserted in an automatic manner. For a quick recovery, in a real-time system described above, the task size cannot be made arbitrary. Furthermore, the finite size of the recovery cache also effects the rollback point insertion strategy as well as the task size. In this correspondence, we look into the rollback problem from both perspectives, the maximum recovery time and the maximum cache size. In Section II, we present some basic concepts and definitions. In Section III, the interdependency of task size, maximum recovery time, and cache size are brought out and the rollback problem is formulated. Section IV outlines an experimental solution of the problem and conclusions are given in Section V. The simulation technique used to obtain estimates of parameters required for the rollback recovery scheme is given in the Appendix.

II. Basic Concepts and Definitions

A program is represented as a sequence of tasks in a program graph, in order to automate and optimize the rollback point insertion. With each task i, a quantity t_i is associated which is defined as the maximum possible execution time for the task. We assume that if an error occurs within a task, it is detected before the completion of the task [1]. Thus, t_i is the expected time of completion for the task i, considering the longest path in the task i.

Definition 1: The *save time* S_i for a task i is defined as the time required to make a copy of the modified variables of the task i, during its execution.

Definition 2: The *load time* L_i for a task i is the time required to load the saved variables of the task i back into the main memory.

The load time obviously depends on the speed of the loading mechanism but is directly proportional to the amount of data saved. The constant of proportionality relating the load time and the amount of data saved will be denoted by k.

In Chandy and Ramamoorthy's algorithm [1], determination of the insertion of rollback points requires an interrogation at each edge of the program graph. The save time and the load time are associated with the edges of the program graph. In practice, however, saving of variables can be done during the execution of a task [4], [6] as opposed to storing the entire set of variables before the execution of a task. Let e_i denote the execution time for a task i when no rollback strategy is implemented and let r_i represent the execution time including the time for making a copy of the variables. Then the worst case value of S_i is given by $\max (r_i - e_i)$ taken over all possible paths in the task i. Note that this will be the path along which the maximum number of variables are changed.

Definition 3: Save space P_i for a task i is defined as the space required to store the variables of the task i modified during the time S_i.

The load time L_i and the save space P_i can be related as $L_i = kP_i$, where the constant k has the dimension of time/space. Note that a

Manuscript received September 12, 1985; revised December 19, 1986. This work was supported in part by ACRB and ARGS Australia and the National Science Foundation under Grant DCR-8509194.

S. J. Upadhyaya is with the Department of Electrical and Computer Engineering, State University of New York at Buffalo, Buffalo, NY 14260.
K. K. Saluja is with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8716364.