

AD-A189 682

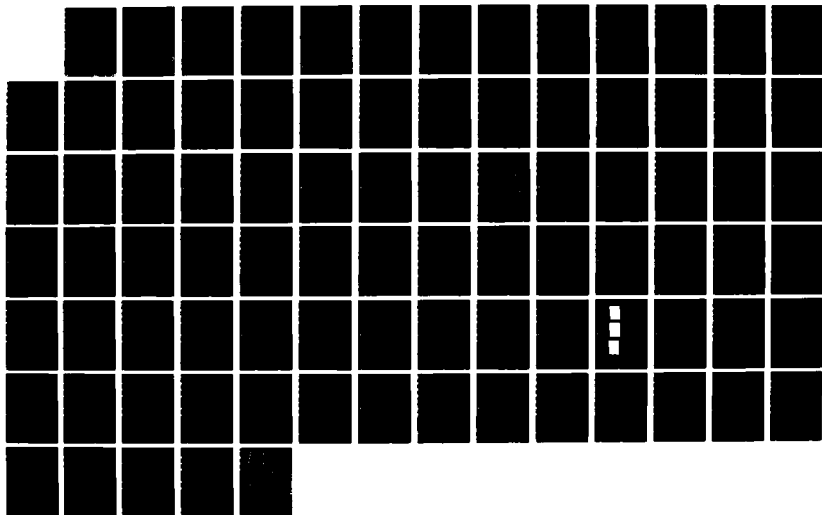
SIMULATION OF FAULT TOLERANCE IN A HYPERCUBE
ARRANGEMENT OF DISCRETE PROCESSORS(U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI
G ZILBERSTEIN DEC 87 AFIT/GSO/ENG/87D-1

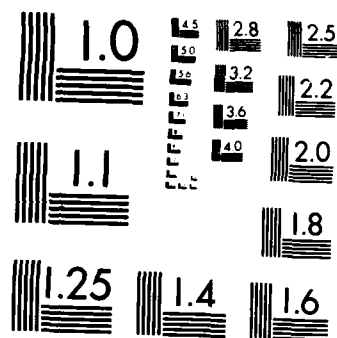
1/1

UNCLASSIFIED

F/G 12/5

ML

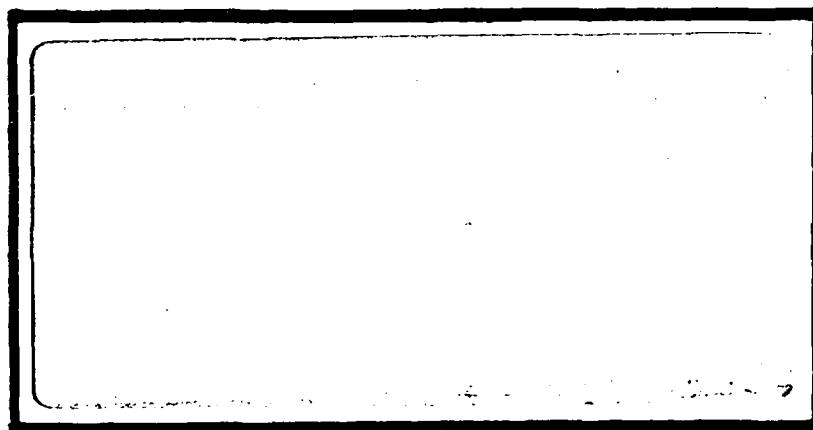




MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY

AD-A189 682



DTIC
ELECTE
MAR 07 1988
S H D

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

88 3 01 1 2 3

AFIT/GSO/ENG/87D-1

SIMULATION OF FAULT TOLERANCE
IN A HYPERCUBE ARRANGEMENT
OF DISCRETE PROCESSORS
THESIS

Gil Zilberstein
Captain, USAF

AFIT/GSO/ENG/87D-1

DTIC
ELECTE
MAR 07 1988
S H

Approved for public release; distribution unlimited

AFIT/GSO/ENG/87D-1

SIMULATION OF FAULT TOLERANCE
IN A HYPERCUBE ARRANGEMENT
OF DISCRETE PROCESSORS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of
Master of Science in Space Operations

Gil Zilberstein, B.A.

Captain, USAF

December 1987

Approved for public release; distribution unlimited

Preface

The purpose of this study was to investigate how parallel programming architectures can be used to provide redundancy. Specifically, it is desirable that computers continue to function in the presence of a hardware failure, with a minimum of duplicated components.

This report is limited in scope to one particular concept, and its implementation on a specific parallel computer which is commercially available. The areas investigated were the adaptation of the theoretical framework to the actual operating system available, and the resulting performance of the system.

It is difficult to find words to convey my gratitude to Dr. M. Kabrisky, my thesis advisor, as well as my appreciative thanks to Lt Col G. Parnell, my reader. Their encouragement, guidance and patience were essential to completing this project. I must also thank Capt B. Hodges, a fellow AFIT student, for the time he took from his own thesis work to help me. I also wish to express my deep appreciation to Mr. Richard Norris of the parallel processing laboratory for his patience and help when the computer became recalcitrant, or when the thesis student got himself hopelessly entangled in the computer's operating system. Finally, I wish to thank, For all of my family and friends who saw and heard very little of me for their patience during these difficult months.

Gil Zilberstein

Availability Codes	
Avail and, or	Special
Dist	Special
A-1	

Table of Contents

	Page
Preface	ii
List of Figures	v
List of Tables	v
Key Definitions	vi
Abstract	viii
I. Introduction	1
Background	1
Research Problem	4
Scope & Limitations	4
Assumptions	4
General Approach and Sequence of Presentation	5
II. Review of the Literature	6
Conceptual Background	6
Self-Organizing Techniques	7
Transfer of Concepts from the Biological to the Electronic Realm	9
Cube-Connected Cycles and their Potential Applications	12
Conclusion	17
III. The Intel "Hypercube" Parallel Computer	19
Basic Concepts	19
Parallel Computers	19
Hypercubes	20
Geometric Properties	22
Binary Properties	26
Intel Hypercube Hardware Arrangement	28
IV. Cube-Connected Cycles	31
Background	31
Basic Concepts	32
Geometric Properties of the CCC	35
CCC Redundancy	38

V.	Re-Configurable Cube-Connected Cycles	40
	Global Redundancy and Reconfiguration ...	40
	Local Redundancy and Reconfiguration	42
VI.	Fault Tolerance Simulation on the Hypercube .	45
	Optimal Methods	45
	Alternative Methods	47
	Node Based Failure Simulation	48
	Cube Manager Based	
	Failure Simulation	49
VII.	Methodology	50
	Analysis and Problem Formulation	50
	Model Building	51
VIII.	Failure and Switch-Over Simulation	53
IX.	Conclusions and Recommendations	58
	Appendix: Computer Programs	62
	Bibliography	71
	Vita	74

List of Figures

	Page
Figure 1: Hypercubes of Different Dimensions	21
Figure 2: Hypercube Properties	24
Figure 3: The Four-Dimensional Hypercube	25
Figure 4: An Example of a Cube-Connected Cycle (CCC) .	33
Figure 5: Node Light Indications on the Hypercube	55

List of Tables

Table 1: Hypercube Parameters Versus Dimensions	23
---	----

Key Definitions

Pattern Recognition. Pattern Recognition is a subset of computer technology whose aim is to enable computers to sense their outside environment in a manner similar to the way living creatures accomplish that task.

Neurons. As used here, a neuron is a computational unit at the lowest level (i.e. computationally indivisible), which performs the following computation: it takes the sum of positive and negative inputs, compares the sum to a threshold, and emits an output if the threshold is exceeded.

Neural Network. A neural network is a structured arrangement of interconnected neurons, which is capable of performing some computation at a higher level than that of individual neurons.

Parallel Computation. The performance of independent computations simultaneously, by independent computational devices, rather than sequentially over time by a single computational device (which is serial computation).

Redundancy. As used here, redundancy is the ability of a computing device to continue functioning in spite of the failure of a physical component, by the use of an identical component which, previous to the failure, was surplus to the minimum set of components required to perform the computation.

Node. As used here, a node is an independent computational device in a cube-type parallel computation architecture.

N-Dimensional Cube. An orthogonal geometric structure, real or imaginary, which resembles a cube in its geometric properties. By way of example: a line is an n-cube where "n" is 1; a square is an n-cube where "n" is 2; a cube is an n-cube where "n" is 3; and so on.

Hypercube. A hypercube is an n-dimensional cube whose dimension is greater than 3, (which is the dimension of Euclidean or "normal" space).

Cube-Type Parallel Architectures. A method of connecting together the independent computational devices of a parallel computation machine, in which the independent computational devices (nodes) are connected to each other as if they were the "corners" of an n-dimensional cube.

Cube-Connected Cycles. A cube-connected cycle (CCC) is a variation of the cube-type parallel architecture, in which instead of nodes at the n-dimensional cube's corners, there are "loop" or "ring" type arrangements of several nodes, which form the corners.

VLSI. The initials stand for Very Large Scale Integrated. A VLSI circuit is one that contains a very large number of electronic components, and their associated connections, formed on a single piece of material of extremely small dimensions (such that the electronic components themselves are of microscopic size).

Abstract

The purpose of this study was to implement a technique for fault-tolerant parallel computation on the Intel Corporation's Hypercube computer. This work was motivated by the recent progress in parallel computation and neural network techniques.

This study focuses on the implementation of one particular type of parallel processing architecture on the Intel Hypercube. The architecture in question is known as the cube-connected cycle (CCC). This architecture is used as a basis for a reconfiguration scheme known as reconfigurable cube-connected cycles. The aim of this reconfiguration is to build a parallel computing system with fault tolerance capability.

Implementation of this technique on the Intel Hypercube was by simulation. The loading of only part of the hypercube's available nodes, holding the remaining nodes in reserve was accomplished, followed by a simulation of the replacement of a deactivated node with a spare node.

Conclusions are reached regarding the suitability of the Intel machine for fault tolerance experiments versus the rapid computation for which it was designed. Recommendations are made regarding the next logical steps in continuation of the work presented in this study.

IMPLEMENTING CUBE-CONNECTED CYCLES ON THE INTEL HYPERCUBE FOR FUNCTIONAL REDUNDANCY

I. Chapter 1: Introduction

Background

The concepts behind this report grew out of the study of the computing problem known as pattern recognition. Some current attempts to solve the pattern recognition problem are now focusing on the use of neural networks (Kabrisky, 1987). Neural networks attempt to implement, in modern electronic computing technology, some concepts deduced from the study of biological brains. Such networks are in an early stage of development, but already they show promise in two main problems in computing. First, they offer the possibility of dramatic increases in speed through the use of a parallel architecture. Second, they offer redundancy so that the failure of a computing element does not compromise the system's ability to perform its function.

This paper explores a method of achieving these same goals of parallel computation and redundancy using more conventional technology which is available today. By improving the effectiveness of computers built on more traditional lines, we may achieve improvements of the type promised by neural networks, while bridging the time gap while neural network technology is maturing.

Such parallel, redundant computing capabilities will probably be needed in the design of orbital systems to solve certain national defense problems, particularly the Strategic Defense Initiative (SDI). General Robert T. Herres, presently Vice Chairman of the Joint Chiefs of Staff, has said that the country's national security "depends on the high-tech edge of our space systems." (Ulsamer, 1985: 92). Such systems must be able to quickly find and identify certain objects or phenomena in their environment.

With regard to Teal Ruby, which is a space-based infrared "staring" (as opposed to "scanning") array, the following has been said.

A technical challenge associated with such advanced staring infrared detection systems is the computational capacity to deal with all the information that is being produced by the million-plus elements of the system (Ulsamer, 1985: 95).

Since existing techniques for solving such problems are slow when compared to the time available to solve the problem, there is a need for faster computation (Evans and Gajewski, February 1985: 74, 77). Parallel computing architectures, whether implemented with traditional technology or with neural networks have the potential of providing the necessary increase in computing speed.

The fragility of computer hardware (and, implicitly, the resident software) in orbiting vehicles is another area which requires improvement. Space vehicles are known to experience radiation and EMP (Electro-Magnetic Pulse) induced problems.

As micro-electronics become smaller, they offer increased capabilities, but they also become more susceptible to transient faults and total dose degradation from the effects of the natural space radiation environment (Gjermundsen, 1984: 28).

The results of radiation-induced damage to hardware, or changes in software, can be disastrous both to the vehicle's immediate mission, and to the vehicle itself (Evans and Gajewski, February 1985: 74, 77). Furthermore, the memory loss involved need not be extensive to cause serious effects. It need only affect a physically tiny part of the computer's circuit, which in turn controls a critical part of the computer's software. As an article in Defense Systems Review states: "Random spacecraft faults must be detected, isolated and corrected." (Gjermundsen, 1984: 27).

There is a need for solutions to such problems which can be implemented as soon as possible. As Gjermundsen says, "New design and manufacturing techniques must be developed to decrease the susceptibility of microelectronics to radiation."; and, "The military command now recognizes the importance of autonomy to the survivability of space assets." (Gjermundsen, 1984: 28). Redundancy, and the automatic reconfiguration capability to use it, are a promising means of achieving that space asset autonomy. It is increasingly apparent that the benefits of this concept outweigh the cost.

Spacecraft configuration tradeoffs are performed initially at subsystems level to achieve a balanced level of protection that satisfies reliability and capability requirements. Further tradeoffs can then be applied to optimize redundancy (Behmann and Nawar, 1985: 191).

The cube-connected cycle concept discussed in this paper offers a way to provide parallelism, and reconfigurable redundancy, with at a reasonable, incremental step forward from current technology.

Research Problem

This is a report on work to investigate fault tolerance, by means of the functional redundancy provided by reconfigurable cube-connected cycles, using the Intel Hypercube computer.

Scope & Limitations

The cube which can be used for this project is limited by the number of nodes available on the Hypercube. The Hypercube was set-up for 32 nodes; The redundancy required by the reconfigurable cube-connected cycles limited the cube dimension available to perform computations.

Failures of circuit components were simulated, since the Hypercube's operating system cannot handle actual failure of computer hardware.

ASSUMPTIONS

It is assumed that the problems to be solved can be effectively parsed, or divided, into parts which are amenable to solution by parallel processing. The problem of how to set up computing problems for effective computation in parallel will not be addressed here.

It is also assumed that the failures which are being simulated are not degenerate in the sense that they will not

nullify the redundancy scheme which is proposed. In other words, if there are two mutually redundant components, it is assumed unlikely that the first two failures will affect those two components, thereby eliminating the redundancy.

General Approach and Sequence of Presentation

A review of the literature will be presented first, covering the concepts of parallel computing and redundancy as they are derived from work on neural networks, and their application to proposed cube-connected cycle topologies.

This will be followed by a detailed explanation of the cube-connected cycle topology, the Hypercube's structure and operating principles, as distinct from those of a conventional computer, and the possibilities of re-programming integrated circuits in the manner simulated in this project.

Following the above, the actual simulation, results, and conclusions drawn will be presented. The actual computer programs used will be covered in an appendix in order to retain the flow of the main text.

II. Chapter 2: Review of the Literature

Conceptual Background

The inspiration for this project came from the study of pattern recognition, and in particular neural networks. For this reason, this review of the literature will start with a review of neural network related material.

The topic of self-organizing techniques is relevant, since the proposed parallel computation technique in this report is able to independently detect, and compensate for failures in its component elements. The self-organization referred to in the neural network literature is, admittedly, of a qualitatively different character. The purpose is to show the potential of independent machine action. What is proposed in this report is simply a rudimentary starting point.

The literature concerning the transfer of concepts from biology to electronics is also covered. The biological computing model, insofar as it is understood today, has served as an inspiration, if not an actual model for what can be achieved with parallel computing architectures. In particular speed and redundancy seem to be key characteristics of parallel machines.

It is in light of this background that the following review of the literature is presented.

Self-Organizing Techniques

No discussion of neural networks would be complete without a reference to Dr. Frank Rosenblatt's Perceptron (Butz, 1959: 60-71; Hopfield and Tank, 1986: 625-633; Fukushima and Miyake, 1982, 455-469). The Perceptron was introduced 30 years ago (Butz, 1959: 60-71). As described by Butz and Hopfield & Tank, the Perceptron was a computing device using two-state (on/off) neuron elements in several layers (3 layers according to Fukushima & Miyake) which was reported to be capable of self-teaching and pattern recognition.

The key idea was that the machine learned with an absolute minimum of programming. Random connections, together with the characteristics of the neurons were deemed sufficient to enable learning and pattern recognition to take place (Kabrisky, 1987). According to Fukushima & Miyake, Rich, and Kabrisky, the Perceptron was a disappointment (Fukushima and Miyake, 1982:455; Rich, 1983: 363; Kabrisky, 1987). Hopfield & Tank, as well as Fukushima & Miyake and Kabrisky state that upon further study the Perceptron was found to be a peculiar, and limited type of two-state value manipulator (known as a Boolean operator). According to Rich, techniques using random networks as starting points have failed because a low-level threshold of knowledge is a prerequisite for learning to take place (Rich, 1983:363).

What is interesting is that the basic concepts behind the Perceptron have not died.

Fukushima & Miyake have been working on a "cognitron" and a "neocognitron" (Fukushima & Miyake, 1982: 455-456). They work on the premise that the problems with the Perceptron were in its details of implementation, not in its basic concepts. Specifically, they propose more layers of neurons in the computing hierarchy and more widespread reinforcement of excited neurons throughout the network. They report these improvements have resulted in computations that are both useful for pattern recognition, and result in behavior similar to living beings.

Some of the same general concepts have been used by Ralph Linsker, but with a stronger connection to the biological realm (Linsker, 1986a: 7508; 1986b: 8390; 1986c:8779). Linsker is seeking to understand spatial orientation in certain primates. The common ground with Perceptrons is the use of a self-adaptive network of neurons (Linsker calls them cells) with minimal pre-programming. In Linsker's own words: "the detailed connections and strengths are unspecified and allowed to emerge during the development of the system" (Linsker, 1986a: 7508). What fascinates Linsker is that the results are "biologically interesting network structures" (Linsker, 1986a: 7508). Furthermore, he sees a parallel to the observed phenomenon that in some primates, cells in the visual cortex can self-organize in the absence of visual stimulation (Linsker, 1986c: 8779). It is this apparent similarity to observed biological behavior that interests Linsker.

The preceding research is groping with the task of devising computing architectures that are capable of self-organizing, i.e. learning, in a manner similar to living creatures. While this work is progressing, it makes sense to develop simple, architectures such as cube-connected cycles as a means of obtaining an immediate improvement in performance based on the idea of an independent, adaptive computer architecture.

Transfer of Concepts from the Biological to the Electronic Realm

In 1959, D.G. Willis of the Lockheed Aircraft Corporation's Missile and Space Division borrowed concepts from biology to define neurons with properties that would be of interest to electronics (Willis, 1959: 1,2-3,5). Willis concluded that modeling a neuron as a device that has two states, on or off, is insufficient to explain the observed behavior of the human brain. It also appeared to him that memory does not exist in individual neurons (Willis, 1959: 5).

Matthew Kabrisky of the Air Force Institute of Technology (AFIT) also realized the significance of investigating the operation of the human brain with a view to modeling at least parts of its functions (Kabrisky, 1966: 4,5). Given the speed of electronic computers (even those available in 1966), Kabrisky could foresee benefits to be gained from coupling electronic speed with the organizational and operational principles of the human brain.

There is however, one caveat Kabrisky mentions (Kabrisky, 1966: 4,5). Not all features of the human brain's construction are necessarily useful in designing computers. For example, birds can teach us much about aerodynamics. However, the fact that birds use feathers and flap their wings is irrelevant to the design of airplanes (Kabrisky, 1966: 4, 5).

Attempting to understand how the brain computes from its anatomy can be problematic. As Hopfield & Tank point out (Hopfield and Tank, 1986:625), the basic elements of a neural circuit have been well known for some time: neurons, synaptic connections, and circuit dynamics. But a detailed study of the components, or a sampling their activity does not shed much light on what is happening in a natural neural network.

One lesson learned from the study of these model circuits is that a detailed description of synaptic connectivity or a random sampling of neural activity is generally insufficient to determine how the circuit computes and what it is computing (Hopfield and Tank, 1986: 625).

D.G. Bounds expresses a similar thought a bit differently (Bounds, 1986: 11,12). He says that although the brain has been taken apart and its components examined, the algorithms it executes are still not known. But Bounds says that 3 key features have emerged: first, the brain is a highly parallel computer; next, inter-element connectivity among neurons is orders of magnitude higher among neurons than among miniaturized electronic components; lastly, biological systems are not binary. Bounds also cites Hopfield's work as showing

that fast computations can be obtained from parallel interactions between neurons.

R. Messner and H. Szu also studied the brain's operation, but they picked out a more narrow area of study (Messner and Szu, 1985: 50,51). They focused on the interface between the retina and the brain. Since the structure of this interface cannot be easily examined while it operates in a living being, computer simulation can be used.

According to Messner and Szu, the "spatial distribution of the cone photoreceptors" (Messner and Szu, 1985: 51) will provide the necessary knowledge for image processing simulations. They also suggest that the interface from the retina to the brain is a logarithmic mapping.

John Maddox, in addition to reinforcing some of the preceding concepts, points out that part of the problem is the difficulty of asking the right questions and the most useful questions when doing research in this area (Maddox, 1987: 11). He states that the proper questions sometimes show up in unexpected places, such as in neuron simulations. Thus, he says that even simulations based on two-state, on/off neurons can yield useful information about living systems. Maddox cites Hopfield when he concludes that memory is a property of a network, not of single elements within a network. This, Maddox says, may mean that memory recall can be invoked by stimulating a fragment of a network.

In his article "Distributed Memory", Leon Cooper reinforces some previous ideas, as well as adding some of his own (Cooper, 1985: 1,2,3-6, 8-9). He agrees with previous authors that the brain processes information in parallel, an idea that Cooper thinks is evident from the structure of the retina. Likewise, he also believes that it is the simultaneous activity of many neurons that is important in a neural network. Cooper also states that, "Large networks of neurons connected to other neurons via modifiable synaptic junctions provide the physiological substrate for the distributed parallel systems discussed here." (Cooper, 1985: 1).

All of the preceding points to parallel computing techniques as being a promising direction to take in improving performance. The studies of biology and neural networks point up the complexity levels of highly capable parallel "computers". While such complexity is beyond the scope of our technology, there is a need to maximize the capabilities of the comparatively simple parallel architectures which we do have.

Cube-Connected Cycles and their Potential Applications

The idea of using cube-connected cycles as a means of interconnecting a parallel computer was first proposed in a paper by Preparata and Vuillemin (Preparata and Vuillemin, 1981: 300-309). They approached the problem from the point of existing electronics technology (specifically VLSI, or very-large-scale-integrated circuits), rather than from the

point of biological studies, as many neural network researchers do.

Preparata and Vuillemin point out two ends of a spectrum in the implementation of parallel processing connections on a VLSI circuit (Preparata and Vuillemin, 1981: 300). One extreme is to have each pair of processing elements connected to each other. This scheme comes quite close to the way processing elements (neurons) in biological computers (brains) appear to be connected, and thereby close to the work being done in neural networks. The implementation of such connection schemes on VLSI circuits however, is impractical given the technological state of the art.

At the opposite end of the spectrum, Preparata and Vuillemin say, some researchers limit themselves to "planar links between topologically neighboring cells (arrays or meshes)" (Preparata and Vuillemin, 1981: 300). The advantage of these is that they are well within the limits of the current state of the art in VLSI technology. The disadvantage however, is that they cannot efficiently implement certain kinds of algorithms, particularly sorting and convolution problems. The point at which they are at a disadvantage is in moving data between points that are topologically far apart in a 2-dimensional (planar) array.

One answer to the problem of implementing algorithms such as discrete Fourier transforms, convolution, sorting, etc. has been the n-dimensional cube (Preparata and

Vuillemin, 1981: 300). Implementing an n -dimensional cube with VLSI technology is, however, infeasible for large values of " n " dimensions.

This has led to alternative layouts. One proposed layout is the perfect-shuffle-exchange (Stone, 1971: 153; Preparata and Vuillemin, 1981: 300). By simulating the perfect shuffle of two halves of a deck of cards (after the shuffle, cards from the first and second halves of the original deck will alternate), processors which were originally topologically far apart are brought closer to each other. The shuffle connection scheme has the advantage that it can be implemented in current VLSI technology.

Preparata and Vuillemin proposed the cube-connected cycle as a means of achieving the best of both worlds, cube-connection advantages and ease of implementation within existing VLSI constraints. It is a modified cube type of network, but has a compact and regular VLSI layout. Additionally, Preparata and Vuillemin say it is suited to "a wide class of problems" (Preparata and Vuillemin, 1981: 300).

A similar idea was expressed by Seitz in his article "The Cosmic Cube" (Seitz, 1985: 22).

The Cosmic Cube nodes were designed as a hardware simulation of what we expect to be able to integrate onto one or two chips in about five years. (Seitz, 1985: 22)

Preparata and Vuillemin's work forms the basis for work by Banerjee, Kuo and Fuchs (Banerjee et al, 1986: 286-291). This newer work adds the idea of reconfiguration capability to

the previous work done on cube-connected cycles. Banerjee, Kuo and Fuchs agree with Preparata and Vuillemin that cube-connected cycles offer an efficient way of performing algorithms such as discrete Fourier transforms and sorting. They also agree that n-dimensional cubes are "not directly usable in VLSI" (Banerjee, et al, 1986: 286), and that the cube-connected cycle "is suitable for realization in VLSI".

To the preceding work Banerjee Kuo and Fuchs add the idea of "designing reconfigurable CCC networks which are capable of tolerating classes of multiple failures." (Banerjee, et al, 1986: 286). They point out that current trends permit the implementation of parallel architectures on VLSI chips or wafers, using large numbers of interconnected processing elements. But this necessitates the consideration of reliability issues.

However, as the number of active devices on dies and wafers increases, the probability of single or multiple physical failures becomes unacceptably large. Consequently, redundancy has to be included in these systems to increase yield when there are production defects and to increase reliability when there are operational failures (Banerjee, et al, 1986: 286).

Banerjee, Kuo and Fuchs propose "cube-connected cycles (CCC) architectures capable of tolerating multiple failures." (Banerjee, et al, 1986: 286). They further state that cube-connected cycles are superior to hypercubes (multi-dimensional networks with a cube-geometry) because

the CCC complies with the basic requirements of VLSI technology: modularity, ease of layout, simplicity of communication among the PEs and simplicity in timing and control (Banerjee, et al, 1986: 286).

Two approaches to achieving the desired redundancy are presented in the above referenced paper, 1) global, and 2) local, redundancy and reconfiguration. Both techniques can retain the original CCC structure, while invoking a reasonable cost in spare or duplicated components, and extra circuit area on the integrated circuit chip.

The hardware side of the circuit reconfiguration issue is discussed in a paper by Greene and Gamal (Greene and Gamal, 1984: 694-717). Their main focus is on forming workable circuits at economic rates in the presence of production defects. They do, however, mention performance of reconfiguration using "electrically reprogrammable switches" (Greene and Gamal, 1984: 694, 716) which are integrated into the circuit, as a means of attaining "dynamic fault tolerance, in which elements fail during use" (Greene and Gamal, 1984: 716).

The paper also mentions that "regular layouts and interchangeable elements" (Greene and Gamal, 1984: 694) are necessary in order to apply reconfiguration techniques. This is similar to the regularity of VLSI circuitry which was mentioned as being desirable in the previous papers on cube-connected cycles.

Specifically, Greene and Gamal say that reprogrammable switches or, in the case of production defects, lasers, can be used to circumvent faulty processing elements in the circuit, and connect together the functioning processing

elements. In this way, any desired circuit within the limits of the chip's structure can be implemented. This benefit, of course, comes at a price. The cost of obtaining this flexibility is increased overhead area (due to switches, extra interconnections and extra processing elements, and an "increase in signal propagation delay." (Greene and Gamal, 1984: 694, 695). In the next sentence, they also mention another cost: "Also, determining the switch settings may entail a nontrivial computation." But this is a problem where ideas such as those of Banerjee, Kuo and Fuchs, about the reconfiguration possibilities of cube-connected cycles can be of help. The common goal is to attain, or maintain a functional circuit (specifically, a parallel circuit) in the presence of failed processing elements.

Conclusion

The literature in pattern recognition implies that a technological revolution is in the making. Attempts to link biological studies of living brains with the design of computers are progressing steadily. However, direct, practical applications are not at hand, and in the meantime modifications of conventional techniques may be more immediately useful.

Many researchers have analyzed the anatomy and behavior of brains and neurons in attempts to find underlying principles. Several conclusions have been reached by several researchers independently. First of all, it is apparent that

the brain processes large amounts of data in parallel (i.e. simultaneously). Likewise, brain functions, particularly memory are distributed throughout the entire neural network of a brain, in such a manner that no single component is indispensable.

It is these two basic concepts, parallelism, and the redundancy of components, that are used in the proposed cube-connected cycles architecture. In this way it is possible to reap some of the benefits of biologically inspired advances, and use the existing state-of-the-art in electronics to implement them. The reconfigurable cube-connected cycle architecture has the potential to provide a significant improvement in capabilities using technology that is close at hand.

III. Chapter 3: The Intel "Hypercube" Parallel Computer

Basic Concepts

Parallel Computers. The Intel Hypercube is a computer with a parallel architecture. A parallel computer makes use of the fact that many computational problems are composed of independent sub-problems. Conventional computers have only one processing device, usually called a Central Processing Unit (CPU). Conventional computers must therefore work on a problem in sequence over time (using the single CPU), in spite of any sub-problem structure inherent in the overall problem.

On the other hand, parallel computers have a number of independent processing devices. Thus, each processing device in a parallel computer can work independently on a sub-problem which is an independent component of an overall problem. If the sub-problems do not depend on results from previous calculations, their calculations can be performed simultaneously as well as independently. It is this capability to solve a computational problem quickly by simultaneous work on independent parts of a problem that gives parallel computation its main appeal.

The difficulty in designing parallel computing architectures is in the management of the independent processing elements. The main computational problem's component sub-problems must be handled in an orderly,

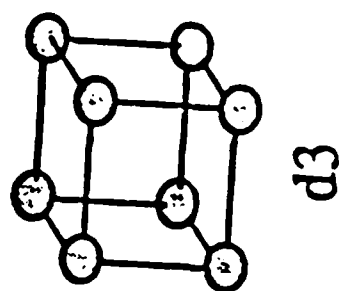
structured way. Furthermore, there must be a master, or overseer in the system which collects, or collates the completed sub-problems and synthesizes them into the solution to the main problem.

Hypercubes. The Intel Hypercube offers one solution to the problem of how to structure parallel computer architecture. It is based on geometric symmetry and geometric properties that are amenable to certain problems suited to parallel computation.

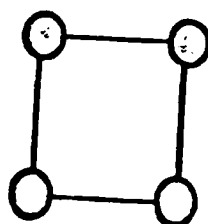
The hypercube uses the concept of the n -dimensional cube. Expanding on the definition already given, an n -dimensional cube is not really a cube in the ordinary sense of the word. A "normal" cube is the 3-dimensional member of the n -dimensional cube family. Calling such structures cubes is simply a convenient way to think about them and visualize them.

An n -dimensional cube consists of nodes, or corners, which are connected to each other in an orthogonal, geometric arrangement. The cases of "cubes" of dimensions "0" and "1" are trivial cases (see Fig. 1). The 0-dimensional "cube" is merely a single node or point. The 1-dimensional "cube" is simply two nodes, or points connected by a single line, (link or channel) (Notes, 1986).

It is at $n = 2$ and higher that the issue becomes interesting. The 2-dimensional "cube" is commonly known as a square. For the 2-dimensional and higher "cubes", the number of connections (links or channels) per node (corner) is equal to the dimension of the "cube" (Notes, 1986). See



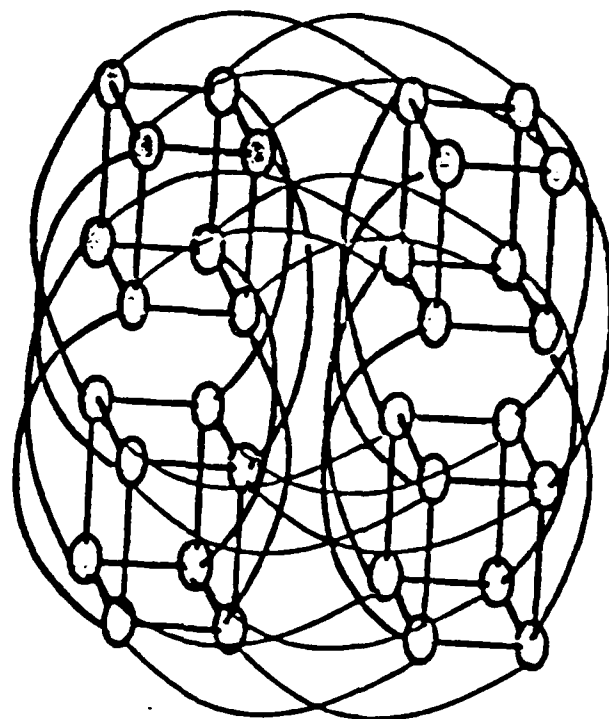
d2



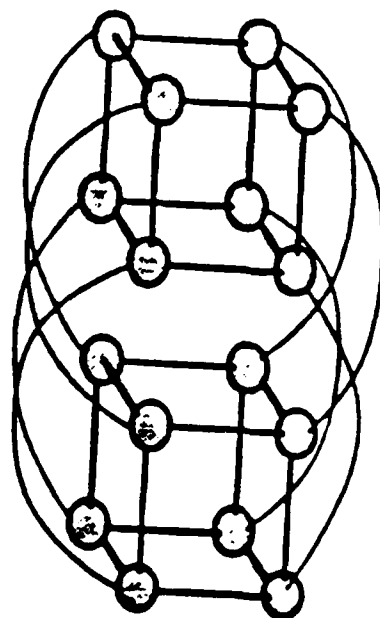
d1



d0



d5



d4

Figure 1: Hypercubes of Different Dimensions
(Notes, 1986)

Table 1 and Fig. 2 for a summary of these cube properties.

Furthermore, it will be noted that sets of parallel structures exist as subcomponents of "cubes", at one dimension less than that of the "cube". In other words, a square (dimension = 2), has 2 pairs of parallel lines (dimension = 1). A "normal" cube (dimension = 3), has 3 sets of parallel planes (dimension = 2).

The difficulty of visualizing the properties of "cubes" of dimensions beyond 3 can be demonstrated by considering a "cube" of dimension = 4. By extension from the preceding discussion, one would expect it to have 4 pairs of parallel "normal" cubes of dimension = 3. But such a geometry is difficult to visualize. Even a rendition of the 4-dimensional cube such as Figure 3 is of limited help.

Such cubes, of dimension greater than 3, are called hypercubes. It is apparent from the difficulty of visualizing the properties of such cubes as described above, why they have earned the special nomenclature. An attempt to depict cubes through dimension = 5 is shown in Figure 1. In order to effectively manipulate higher-dimensional cubes (hypercubes), it is useful to describe their properties in generic terms.

Geometric Properties. As seen in Table 1, "n" will designate the dimension of the cube. The first parameter of interest is the number of nodes (corners) that an n-dimensional cube has. This is of primary interest because when implementing such a cube on a parallel computer, the nodes will be the

Hypercube Dimensions

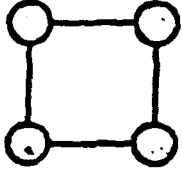
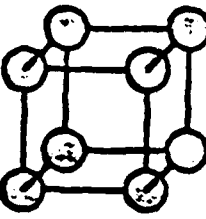
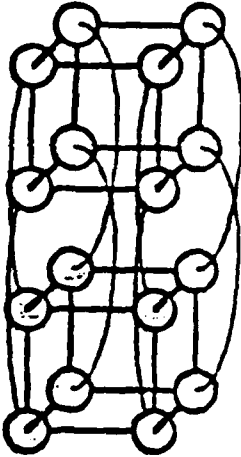
Dimension	Nodes	Chan/Node	Channels	Topology
0 D	1	0	0	
1 D	2	1	1	
2 D	4	2	4	
3 D	8	3	12	
4 D	16	4	32	

Table 1: Hypercube Parameters Versus Dimensions
(Notes, 1986)

- o Total Number of Nodes: $T = 2^n$, n = dimension
- o Number of Nearest Neighbors per Node = n
- o Maximum Distance Between Two Nodes = n
- o Average Distance Between Two (random) Nodes = $n/2$
- o Number of Binary Digits in a Node Label is Equal to the Dimension (n) of the Cube
- o Labels of Nearest Neighbor Nodes Differ by One Bit
- o The Minimum Distance Between any Two Nodes is Equal to the Number of Bits that are Different in their Addresses
- o The Number of Minimum-Distance Parallel Paths Between any Two Nodes is Equal to the Number of Bits that are Different in their Addresses
- o The Number of Parallel Paths Between any Two Nodes is Equal to the Dimension of the Cube
- o Internal Communications Bandwidth = $(T/2) \log_2 T$

Figure 2: Hypercube Properties

(Notes, 1986)

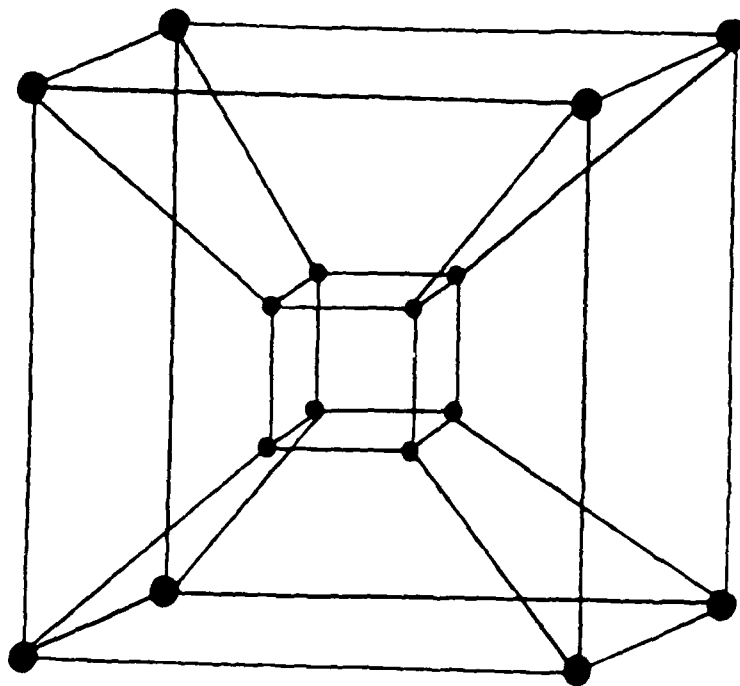


Figure 3: The Four-Dimensional Hypercube

independent computational processors. The number of nodes for an n-cube (i.e. an n-dimensional cube) is equal to 2^n . Therefore, for a computer having this type of architecture, the number of independent processors has to be a power of 2 (Notes, 1986).

The number of channels (links, or lines in the figures) per node is equal to "n", the dimension of the cube. The total number of actual channels however, is more difficult to describe, mathematically it is:

$$n * 2^{n-1} \quad (1)$$

where n is the dimension of the cube (Notes, 1986).

If the n-cube is considered to be a unit cube, that is, all of its "edges" or "sides" are equal to one, it will have the following properties. The maximum distance between a pair of nodes will be equal to "n", the dimension of the cube. (The preceding will hold true for nodes which can be thought of as being at diagonally opposite ends of the cube.) The average distance between any 2 randomly chosen nodes, on the other hand, will be equal to $n/2$ (Notes, 1986).

Binary Properties. In addition to the preceding properties of hypercubes, which are geometric in nature, there is a set of interesting and useful properties which arise from assigning binary numbers to the nodes of the hypercube as addresses. The binary address numbering system starts with the first address (number) being all zeroes.

The addresses progress from there to the maximum number of nodes in a hypercube minus one. The minus one arises because the first node is numbered zero, instead of being numbered one. This property is non-trivial, since it permits a highly ordered and efficient binary addressing scheme to be implemented. The first property of this binary addressing scheme is that the number of digits in the address for a node is equal to the dimension of the hypercube (Notes, 1986).

Note that this would be impossible if the first node were to be numbered one instead of zero. This is analogous to counting ten items by starting with zero and ending with nine, rather than the conventional way of starting with one and ending with ten. By going from 0-9, one digit in the "address" is saved, since the two-digit 10 doesn't have to be used. An analogous situation exists with binary numbers. One digit in a binary address is saved by starting the count with zero. This is no small consideration when designing a machine which will have to use this address in large numbers of components, many times over as it operates.

In addition to the above property of having the dimension of the cube be the same as the number of digits in its address, there is an address relationship between nodes. The simplest such property is that nearest neighbor nodes differ in their addresses by only one digit of their addresses (Notes, 1986). By nearest neighbor is meant nodes

which are only a unit "length" apart. (In a "normal" square or cube, these would be corners separated by a common edge.)

A property which, with further consideration, arises from the preceding, is that the minimum distance between any two nodes is equal to the number of digits that are different in their addresses (Notes, 1986). This property can be seen by considering the intermediary nearest neighbor pairs as stepping-stones between any desired pair of nodes.

Continuing with the less obvious properties, the number of minimum-distance parallel paths which are possible between any pair of nodes is equal to the number of bits that are different in their addresses (Notes, 1986). This is a property of the cube's symmetry, and can be seen without too much difficulty in the 2-dimensional and 3-dimensional cubes.

A feature which arises from the earlier property that each node has "n" channels, and the property mentioned in the previous paragraph, is that the number of parallel paths possible between any two nodes is equal to the dimension of the cube (Notes, 1986). This can be seen by noting that the originating node has "n" channels going "out", and the destination node has "n" channels coming "in".

Intel Hypercube Hardware Arrangement

The Intel Corporation's Hypercube parallel computer uses the available state of the art in electronics to implement the above cube-connection concepts. The nodes (corners) of the cube are, in actuality, microcomputers. These node

microcomputers are connected together by dedicated high-speed communication channels to form the "cube".

The cube, however, cannot function as a stand-alone unit when implemented in hardware. Besides the usual computer input/output devices (keyboards, terminals, printers, etc.), the cube needs a system manager. This management function is provided by a "Cube Manager" (iPSC User's Guide, 1986). The cube manager is also, like the nodes, a microcomputer, but with much more memory, and running a different program.

Each node's microcomputer has some memory associated with it. Specifically, each node uses an Intel 80286 Central Processing Unit, an 80287 Numerical Processing Unit, 512K of Dynamic Random Access Memory (RAM, i.e. usable memory) and 64K of ROM (Read-Only Memory) (iPSC User's Guide, 1986). The nodes also have three communications channels with their associated communications processors and controllers. These are: the inter-node channels; the global channel to the cube-manager; and the diagnostic channel to the cube-manager. Additionally, the nodes are provided with red and green LEDs which are visible at the front of the cube's cabinet. These LEDs illuminate to provide diagnostic information (iPSC User's Guide, 1986).

The cube-manager is an Intel System 310 microcomputer, and uses the same Central Processing Unit and Numerical Processing Unit as the nodes, but with more memory. The cube-manager has a 140 Meg. hard disk drive, a 360K floppy

disk drive, a 45 Meg. cartridge tape and 2 Meg. of Random Access Memory (RAM) (iPSC User's Guide, 1986).

The cube-manager's software capabilities cover several different functional areas. First of all, communication with the cube is made through the cube-manager using a VT-100 compatible terminal. The operating system is Xenix, which is a derivative of Unix. The system can be programmed in C or Fortran, and in addition has software to perform communications management within the cube, and diagnostics.

The communications management function is necessary because the nodes communicate with each other and with the cube-manager by "message passing" (iPSC User's Guide, 1986). The communications software must manage bit packaging, as well as message protocol, queueing, addressing and collision avoidance, among other functions. The system also has diagnostics permanently residing in ROM for: initialization, confidence tests, the initial addressing of the nodes, and primitives (simple permanent programs) which enable nodes to receive their initial programs (iPSC User's Guide, 1986).

The nodes and the cube-manager each have their own versions of a software library of commands and functions unique to the hypercube environment. In their roles as components of a parallel computer, the nodes can independently execute their own programs on their own data after downloading them from the cube-manager.

IV. Chapter 4: Cube-Connected Cycles

Background

The use of cube-connected cycles (CCC) as a means of interconnecting parallel processors was proposed in 1981 by Preparata and Vuillemin (Preparata and Vuillemin, 1981). Their main motivation was the conflict between the parallel processing topologies which were ideal from a computational standpoint, and those which were ideal from the standpoint of implementation on VLSI chips.

The topology best suited for VLSI implementation is a planar layout, which can be an array or mesh (Preparata and Vuillemin, 1981: 300). While such a layout is commendably simple, it limits the processors' communication to "planar links between topologically neighboring cells" (Preparata and Vuillemin, 1981: 300). While such a layout is suitable for some types of problems, it is not suitable for problems where data must be moved "between processors that are topologically far apart." (Preparata and Vuillemin, 1981: 300).

A topology which does enable the latter types of problems to be solved efficiently is the n -dimensional cube. As an example: sorting, permutations and fast Fourier transforms "are algorithms whose data exchange pattern corresponds to the links of the binary multidimensional cube." (Preparata and Vuillemin, 1981: 300; Banerjee et al, 1986: 286). But, while the n -dimensional cube accommodates the data exchange

patterns necessary for certain algorithms, "it is not directly usable in VLSI" (Banerjee et al, 1986: 286). The reason for this is that "each of the 2^k processors in the system is connected to k other processors." (in Preparata and Vuillemin, k=n), (Preparata and Vuillemin, 1981: 300). Such a connection scheme is impractical in current VLSI technology.

The cube-connected cycle (CCC) has therefore been proposed as a practical alternative to an n-dimensional cube connection network (Preparata and Vuillemin, 1981: 300), (Banerjee et al, 1986: 286). The CCC is readily adaptable to VLSI technology due to its structural regularity (Banerjee et al, 1986: 286). Furthermore, processing time for algorithms suited to n-cube implementation "is not significantly increased" (Preparata and Vuillemin, 1981: 300) in comparison to an n-cube.

Basic Concepts

The cube-connected cycle (CCC) as presented by Preparata and Vuillemin, and utilized by Banerjee et al, is based on a 3-dimensional cube (see Fig. 4). What makes this 3-dimensional cube unique is the structure of the corners. In a conventional n-dimensional cube layout, each corner is a node, or processing element. In the CCC, each corner is a ring-type connection of several processing elements (see Fig. 4). These ring-connection structures are called "cycles" (Preparata and Vuillemin, 1981: 303), hence the name of the connection scheme.

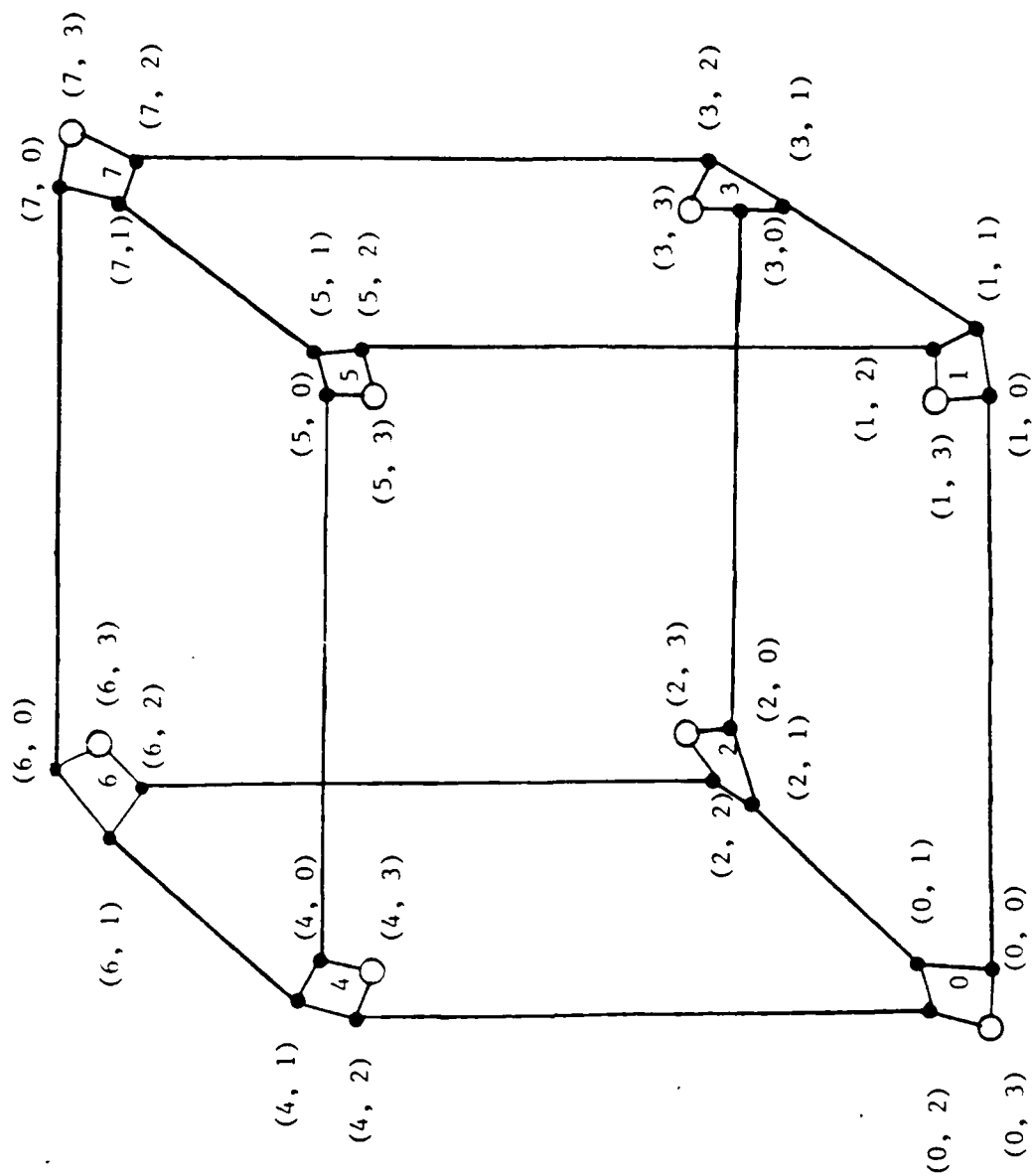


Figure 4: An Example of a Cube-Connected Cycle (CCC)

Since the overall cube is 3-dimensional, each cycle has 3 connections to adjacent cycles (formerly corners in an n-cube, see Fig. 4). Furthermore, each processing element within the cycle has a maximum of 3 connections to other processors. The processors which link-up to adjacent cycles have the two links to neighbors in their cycle, plus one more link to the adjacent cycle (see Fig. 4). Processors in a cycle which do not link-up to adjacent cycles simply have the two connections to their neighbors in their cycles. It is this limit to the number of connections or links required by the processing elements that makes the CCC a practical proposition for implementation in VLSI. The cube's advantage of communication to distant nodes is retained, but with a limit to the number of communication links each processor is required to have.

In the examples used in Preparata and Vuillemin, and in Banerjee et al, the cycles have four processing elements each, as illustrated in Figure 4. Thus, three of the four processors in a cycle are linked to adjacent cycles. Additionally, the total number of processors is 32 (4 per cycle or corner, multiplied by 8 corners in a 3-dimensional cube). If this number of processors were to be connected in the conventional n-dimensional cube scheme, the cube would be a 5-dimensional cube ($2^5=32$), and the number of connections per "corner" would be 5, thus complicating things greatly (see Table 1).

Geometric Properties of the CCC

To begin with, all processing elements are assumed to be identical, which permits the free assignment of sub-problems of the main problem to any processor. As stated previously, each processing element has either 2 or 3 ports with which to communicate to other processors, depending on whether or not it communicates with adjacent cycles. The total number of processors required for a given CCC is derived from the following equation:

$$T = h * 2^n \quad (2)$$

where

T = number of processing elements

h = number of processing elements per cycle

n = dimension of the cube

2^n = number of cycles (analogous to corners of the cube)

$*$ = multiplication

$h \geq n$

(Banerjee et al, 1986: 286)

In words, the above equation states, "The processing elements are grouped into 2^n cycles, each cycle consisting of h processing elements." (Banerjee et al, 1986: 286).

Each processing element has an address in the form of a pair of integers, (c, p) . In this addressing scheme, " c " is the address of the cycle (corner) to which the processing

element belongs, while "p" is the address of the particular processing element within the cycle denoted by "c". The limits of the "c" address parameter are: $0 \leq c \leq 2^{n-1}$, due to the requirements of the digital numbering system described in Chapter III under "Binary Properties". The limits of the "p" address parameter are $0 \leq p \leq h-1$, again the count does not go all the way to "h" due to the binary numbering requirements previously described.

The preceding description of the addressing limits will be helpful in understanding which processing elements have 3 communications ports, and which have 2, in the proposal by Banerjee et al. The following will apply to the processing elements within any given cycle of address "c". The processing elements whose "p" addresses are within the limits: $0 \leq p \leq n-1$ will have 3 communications ports, because the cycles which form the corners of the cube need to be linked up to the dimension of the cube, which is $n-1$. Processing elements with addresses beyond the above, i.e. with addresses: $0 \leq p \leq n \dots h-1$, have 2 communications ports, because the remainder of the processing elements beyond the cube dimension (i.e. $p > n-1$), need to link only with their adjacent neighbors within the cycle (corner).

The 3 possible types of communications ports are classified as: F (forward), B (backward), L (lateral) (Banerjee et al, 1986: 286), (Preparata and Vuillemin, 1981: 303). The processing elements forming any particular cycle

are connected into a cycle (analogous to a circle) by the F and B (forward and backward) ports. The first n processing elements in each cycle (numbered $p=0 \rightarrow n-1$), are interconnected into an n -cube by using the L (lateral) ports (in addition to the F and B ports which form the cycle). The remaining processing elements (numbered $p>n-1$), have only the F and B ports (Banerjee et al, 1986: 287). There is an interesting subtlety about the role of the p parameter in the processor's (c, p) address. When processors in different cycles (cube corners) which are connected together via their L (lateral) ports, these connections form the edges of the cube. Furthermore, there are sets of four parallel edges in a 3-dimensional cube which have a common axis. These edges, with a common parallel axis (one of the 3 axes of 3-space), are denoted by a common value of p in their (c, p) addresses (see Fig. 4).

It is now possible to express, in a generalized form, the connections between any pair of processing elements. In the following, F, B and L as well as (c, p) are the same as referred to in the preceding text.

$F(c, p)$ connects to $B(c, (p + 1) \text{ modulo } h)$

$B(c, p)$ connects to $F(c, (p - 1) \text{ modulo } h)$

$L(c, p)$ connects to $L(c + e * 2^p, p)$

where

$$e = 1 - 2 * \text{bit}_p(c)$$

$$\text{bit}_p(c) = \text{the } p^{\text{th}} \text{ bit of } c$$

Expanding on the preceding: bit, as used above, means a digit in a binary number. The significance of this is that the expression $\text{bit}_p(c)$ will be either a 0 or a 1, since these are binary numbers. In turn, this means that e will have a value of either $+1$ or -1 . Therefore the purpose of e is to serve as a positive/negative decision value. This positive/negative decision is necessary to define the forward/backward direction of the connection between ports within a cycle, and is similar to the symmetry between the F port using $(p + 1) \text{ modulo } h$, and the B port using $(p - 1) \text{ modulo } h$. The L port connects the cube's corners, as shown in Figure 4.

CCC Redundancy

The CCC structure is utilized by Banerjee et al as a starting point for devising a "parallel network architecture" "capable of tolerating multiple failures." (Banerjee et al, 1986: 286). This technique provides an improvement in reliability by the efficient use of a limited degree of redundancy. The CCC does not degrade in performance upon the failure of processing elements because the redundancy is used to "maintain the complete cube-connected cycles structure after successful reconfiguration" (Banerjee et al, 1986: 286).

In order to achieve the desired redundancy and reconfiguration capability, all processing elements will be provided with 3 communications ports (F, B and L) to enable

them to function at any point in the CCC network depicted in Figure 4. A spare cycle is made available to each of the original cycles, which form the corners of the cube in Figure 4, so that upon failure of an original cycle's processing element, a spare cycle can be brought into operation and enable the program to continue to execute as if there had been no failure. Graphically, the spare cycle would be located inside of the cube, accessible to all corner cycles, and would require extra processors in addition to the ones shown in Figure 4. Since the spare cycle is commonly available to all the cycles in the CCC, it offers a reasonable compromise between minimizing the use of VLSI resources and the number of redundant processing elements required, while retaining the cube's parallel processing capabilities.

Two methods of accessing the spare cycle have been devised (Banerjee et al, 1986: 286-291). According to Banerjee et al, "Reliability analysis shows both schemes to provide for significant reliability improvement over classical CCC networks."

V. Chapter 5: Re-Configurable Cube-Connected Cycles

The information in this chapter is based on work by Banerjee, Kuo and Fuchs (Banerjee et al, 1986: 287-289). The basic concept is to enhance the reconfiguration options of a cube-connected cycle (CCC) in the most efficient, economical means possible. Specifically, the methods proposed enable the retention of a CCC structure by the reconfiguration of the network upon failure of a processing element. The proposed methods make this possible using a minimum of additional, redundant processors and inter-processor communications links.

Global Redundancy and Reconfiguration

This reconfiguration scheme adds processors and inter-processor communications links in the following manner. Recalling the discussion headed "Geometric Properties of the CCC" in Chapter 4, we start with a structure of an n-cube (that is, a cube of dimension = n). Each corner of the n-cube has a ring-connected cycle of 'h' processors. These corner cycles are inter-connected to form the n-cube.

In the global reconfiguration scheme, a redundant processor is added for each dimension of the cube, and $(2^{n-1} + 1)$ redundant lateral links which form a ring connection (Banerjee et al, 1986: 287). Additionally, $(h - n)$ redundant processors are added to "form a redundant

cycle of 'h' processors using F and B (forward and backward) links." (Banerjee et al, 1986: 287).

The following benefits are gained by the modified CCC network:

- 1) Each of the processors with a lateral communications port (see discussion in Chapter 4) has gained an extra port due to the addition of a lateral link.
- 2) The number of cycles available in the CCC has increased by one to $(2^n + 1)$.

The net result is that a spare cycle has become available to replace a faulty cycle, without increasing the dimension of the CCC.

The total number of processors required for a globally reconfigurable CCC is given in the following equation:

$$T = h * (2^n + 1) \quad (3)$$

where

T = number of processing elements

h = number of processing elements per cycle

n = dimension of the cube

$2^n + 1$ = number of cycles (analogous to corners of the cube)

* = multiplication

$h \geq n$

(Banerjee et al, 1986: 287)

As previously described in Chapter 4, each processing element has an address of the form (l, p). Once again, 'l'

is the address of the cycle (cube corner) to which the processor belongs, and 'p' is the address of the processing element within its cycle.

In this case, however, the processors having lateral links to other cycles (cube corners) have four ports, not three as in the basic CCC. This is because of the extra connection to the spare cycle in the reconfigurable network. In the following, the connections are expressed in a generalized form, with L_2 denoting the extra lateral connection to the spare cycle.

$F(l, p)$ connects to $B(l, (p + 1) \text{ modulo } h)$

$B(l, p)$ connects to $F(l, (p - 1) \text{ modulo } h)$

$L_1(l, p)$ connects to $L_1(l + e * 2^p, p)$

$L_2(l + 2^p, p)$ connects to $L_2(l + e * 2^{p+1}, p)$

(the above is performed modulo $(2^n + 1)$)

where

$$e = 1 - 2 * \text{bit}_p(l)$$

$\text{bit}_p(l)$ = the p^{th} bit of l

This concludes the description of global redundancy and reconfiguration.

Local Redundancy and Reconfiguration

In this reconfiguration scheme, spare processors and inter-processor communications links are added to each cycle (cube corner) of the CCC.

The utility of this method is based on two assumptions:
(Banerjee et al, 1986: 288)

- 1) Each operational node processor can either process data or let the data pass through it.
- 2) A deactivated or faulty node processor will still allow data to pass through its ports.

In the case of the Intel Hypercube, the first assumption above is, in fact, a basic concept in its processing and message passing system. In the case of the second assumption, whether or not it is met depends on the particular method used to simulate faulty node processors on the Intel machine.

In their work, Banerjee, Kuo and Fuchs state that the first assumption is met by "a classical CCC network." (Banerjee et al, 1986: 288). Insofar as the second assumption is concerned, they propose a "simple network of switches" (Banerjee et al, 1986: 288) to route messages around a failed processor. Although they don't state this explicitly, Banerjee et al imply that these switches must be controllable in real time from the program's software.

The reason for this technique being referred to as local redundancy and reconfiguration is that these functions are provided for and exercised locally at each cycle (cube corner) in the network. A defective processor in any cycle is replaced by a redundant processor in the same cycle, this

being implemented by a reconfiguration of the switch network mentioned earlier. A redundant processor would be depicted in Figure 4 as an open circle, such as the 2-port processors shown. Finally, the reconfiguration is implemented in a manner that preserves the original network structure. In the words of Banerjee, Kuo and Fuchs:

Reconfiguration is performed such that the relative ordering and interconnections of nodes is the same structure as that before reconfiguration.
(Banerjee et al, 1986: 289)

VI. Chapter 6: Fault Tolerance Simulation on the Hypercube

Optimal Methods

The original goal of fault tolerance was to enable a parallel computer to continue its operation in spite of the failure of one of its processors to execute the portion of the problem allotted to it. This failure to execute would render that processor totally inoperative as far as the overall problem solution algorithm was concerned.

The ideal method of simulating such a situation would also involve total failure of a processor, or node in the hypercube structure. The more physical the nature of this failure, the more realistic would be the simulation. An actual failure in an operational circuit would involve some sort of physical change, deterioration or destruction of a circuit component. A simulation using a controlled, duplicatable physical failure would come closest to actual operational conditions.

An example of such a simulated failure would be to disrupt power to a processing node. The node would suddenly cease to respond to the master program (the cube manager in the case of the Intel Hypercube) in any manner whatsoever. The program would then have to be capable of switching in a spare node and continuing to execute the master program without interruption or loss of accuracy.

The most appropriate way to accomplish such a power disruption would be through a special switch or relay built into the hypercube's physical power connectors for its node's electronics boards. Such a device would have to be accessed by the software in order to control the simulation. Additionally, such a device would have to be capable of switching the node processor back on in an appropriate manner, including initialization and self-test functions in order that the node be able to participate in future calculations.

The preceding method, although it would be extremely realistic, is not practical under the circumstances of this project, since it entails fundamental physical and electronic modifications to the Intel Hypercube. This would be an electrical engineering design project requiring close coordination with the manufacturer.

A more simple, and in a sense more realistic way of achieving the same goal would be to physically withdraw a node board while it is participating in a computation. This would be a dramatic demonstration of a fault tolerant system's capabilities. The effects of such an action on a system which was not designed to handle such a disruption are, however, unpredictable. The consequences could be damage to the node requiring down-time and repair.

Because of the above mentioned reasons, the fault simulation methods which are optimal from the point of view

of the realism of the simulation, are impractical in terms of the limitations of the equipment available.

Alternative Methods

An alternative way of simulating fault tolerance without the complications and risks referred to in the preceding section is to perform the simulation solely using software. While this method does not simulate the physical aspect of nodes becoming inoperative, it does provide a safe way, to accurately control when the nodes will come into and leave the parallel computation.

The ring demonstration program provided by the Intel Corporation (Notes, 1986) provides a basis for building such a program. The ring demonstration program is a learning aid for programmers using the Intel Hypercube which passes a message around the nodes. The program prompts the user for a message size, and a number of times to pass the message around the ring.

The ring itself can be considered to be a rudimentary cube-connected cycle (CCC). It is a CCC of dimension $n = 0$, with one node (see Fig. 1). The processors in this single-node ring can be considered to be simply a cycle at that single node with a very large number of members in the cycle. In Formula 2, Chapter 4, this would be a large value of 'h', with $n = 0$.

Taking the above view of the ring structure into consideration, one can build a cube-connected cycle of the

type discussed previously by increasing the dimension 'n' of the CCC, and decreasing the value of 'h' , the number of processors per cycle.

Once such a structure is implemented using software, one has a CCC on which one can perform simulated failures of processing elements.

Node Based Failure Simulation. The failure of a node processor can be simulated in software at the node in question itself. The concept is to cause the node to generate an error which would stop its computation. On the Intel Hypercube, errors at the node processors generate error messages to the cube manager in order to notify the user that an error has occurred, and to aid in debugging.

The action taken by the cube manager upon receipt of an error message from a node processor is variable. Normally, a standardized, 'built-in' exception handler in the cube's software takes appropriate action, depending on the error involved.

It is possible for the user to write his own exception handler, if the action desired upon receipt of an error message is different from what the manufacturer's software provides. These user written exception handlers must, however, be written in assembly language, and therefore demand a high degree of expertise from the user.

A simple way to make use of the preceding capability would be to load the node which is to be deactivated with

data which would generate a computational error, such as division-by-zero. These data would be loaded into that particular node by the cube manager. A user written exception handler to that particular error would then switch in a spare processor, and keep the overall program running.

Cube Manager Based Failure Simulation. Another point at which a node processor's failure can be simulated in software is at the cube manager. This method uses node control commands provided by Intel for the Hypercube's operating system.

The failures can be performed in the operating system by using the 'kill' function built into the software libraries (iPSC User's Guide, 1986). The node in question can be deactivated by using a 'kill' command from the cube manager selectively on that node. Thus, nodes can be deactivated at will from the program running on the cube manager, in a manner that does not disrupt or change the standard operating system.

VII. Chapter 7: Methodology

Analysis and Problem Formulation

As mentioned early in this report, there are several emergent technologies in computer design which hold promise for solving problems related to satellite-borne data processors. Parallel computing is particularly attractive for space applications.

The specific problem areas where parallel computation holds the greatest promise are speed of computation and redundancy. In this report, the focus was on the problem of redundancy. Space-borne systems are vulnerable to all of the failures to which a device of equivalent quality would be subject in ground operations. Additionally, there are characteristics unique to the space environment, specifically, radiation damage, operation in a vacuum, and operation under extreme temperatures. To the preceding must be added the inaccessibility of the device once it is in orbit.

The study required the selection of an approach to the study of solutions to the satellite reliability problem, using information in the published literature as a starting point. The most salient characteristics of a system which have a bearing on reliability are: 1) the probability of failure of a given component; 2) the availability of redundant components in case a component does actually fail.

The probability of failure of computing system components is rooted in their detail design. This kind of analysis was excluded from the scope of this report in favor of the issue of redundancy. This was necessary to narrow the scope of the problem which was addressed.

The focus of the study was narrowed to one proposed system instead of a variety of alternative systems. This enabled the use of unique computing facilities at the U.S. Air Force Institute of Technology (AFIT). It also permitted this system to be examined to a greater degree of depth than would have been possible with a comparative report.

The method discussed in this report, that of reconfigurable cube-connected cycles (CCC), exists only in the literature at the time of writing. It does, however, promise feasibility in 3 important factors: 1) technical feasibility; 2) economic feasibility; 3) operational feasibility (Markland, 1983: 9). This study deals with the technical and operational feasibilities of reconfigurable CCCs.

Model Building

There are two classes of models which apply to the problem solving methodology. At the early stages of a project, the models used are of the initial type. These models are qualitative in the way they treat the problem. Subsequently, the models evolve into quantitative models, as

more information becomes available, and the problem can be defined rigorously in quantitative terms. The model can then be manipulated by means of controllable quantitative variables.

This study deals with an initial, qualitative model. The reconfigurable CCC is a conceptual model for one approach to solving the reliability problem in computational machines. The reconfigurable CCC is a subset of the use of parallel computer architectures to achieve reliability through redundancy.

Further methodological development would require the testing of a completed system, preferably with the involvement of a potential user. Feedback from a potential 'real-world' user would refine the model, and test its applicability to an environment outside of the laboratory. Once a system is found to be applicable to a user's environment, and acceptable to the user's method of operation, the solution can be implemented. The implementation plan should accommodate a method for following the system's performance in the field to ensure that it continues to meet operational requirements.

VIII. Chapter 8: Failure and Switch-Over Simulation

Proceeding from the previous chapter, the method chosen for simulation of node failure, and node switch-over was by control from the cube manager. The commands used were from those available in the cube manager's software library provided by the Intel Corp. The program was based on the ring demonstration program provided by the Intel Corporation (iPSC Program Development Guide, 1986).

The Intel ring demo program is intended to demonstrate the message passing system of the Intel Hypercube. The nodes are arranged, as the name suggests, into a ring network. The nodes then pass a message of given size, a given number of times in a loop around the ring. The program interacts with the user at a terminal, querying the user as to the size (in bytes) of the message to be passed, and the number of times to loop around the ring. The program also sends information to the terminal screen to keep the user informed of the program's progress.

The first set of modifications necessary to the ring demo program are to the cube manager's program, which is called the host program (program host in the Appendix). The main purpose of these modifications is to control the node loading function. The original program loads all of the available nodes (determined by the cube's dimension) simultaneously using the 'call load' function. This function

must be modified to enable the load to load only part of the total number of available nodes, in order to keep some nodes as spares.

The loading was accomplished using the 'call load' command, but with the individual nodes specified (see Appendix). A specified number of nodes, in this case the first 30 out of 32, were loaded using the 'call load' command in a loop. In addition to permitting two nodes to be held in reserve, loading the nodes by means of a loop allowed the process to be easily observed by watching the node lights provided on the node boards to monitor node status (see Fig. 5).

The next modification enabled the user to interactively select which node would be simulated to fail. Terminal screen prompts tell the user to pick a node from 1 to 29 for a simulated failure (see Appendix). The prompt explains that node 0 cannot be used because it is the one that communicates with the cube manager. The user is also told what to do if the program's execution hangs-up.

An operational modification follows the above prompts. A variable previously defined as 'kaputtnode' is used to denote which node the user has selected for a simulated failure. The variable 'kaputtnode' is then used in a selective 'call kill' command from the cube manager, to deactivate the selected node. A 'call load' command is then issued from the cube manager to load in one of the spare

nodes (see Appendix). In this program, node 30 is selected as the node to be substituted for the failed node.

This change-over of the nodes can be monitored by viewing LEDs (light emitting diodes) installed on the node boards to monitor node status. The photographs in Figure 5 show how the LEDs reflect the nodes' activities during the change-over. Figure 5 shows nodes numbers 30 and 31 (the 31st and 32nd nodes) in stand-by status after loading the first 30 nodes. Figure 5 shows node number 29 after deactivation, with node number 30 taking its place. Likewise, Figure 5 shows node number 23 after being deactivated, and again, node number 30 taking its place.

The final modification to the host program on the cube manager is the enlargement of the message buffer from 2 to 3, and the definition of the new buffer thus created, see the Appendix. The purpose of this is to allow the value of the inoperative node's address to be transferred from the host program to the node programs. As seen in the Appendix, the third message buffer (msgbuff (3)) is used to transfer the value of 'kaputtnode', the inoperative node, to the node programs.

The modifications to the node programs concerned the addressing system for message passing in the ring. Specifically, the method by which any given node calculates the next node to which the ring's message should be sent needs modification to take into account a deactivated node in

the ring. Firstly, as seen in the Appendix, the modulo function in the 'nextnode' statement needs to be modified to 30, the number of nodes in use.

Finally, modifications are required to route messages around the deactivated node via the substitute node (in this case, node number 30). This routing consists of two 'if' statements. The first 'if' statement takes effect if the next node in the message passing sequence is the deactivated node. This statement then passes the message to the replacement node (in this case, number 30). The second 'if' statement's purpose is to return the message back to the ring network at the point where it would have been had there been no deactivated node.

This last part of the program is, at the time of writing, not functional. When an attempt is made to run the ring demo program, the program stops without returning a ring count. The success of the node switch-over, which simulates replacing a failed node processor, can be seen by viewing the node board's LEDs, as described previously. This part of the program is functional.

The entire program, written in Fortran, compiled with no errors or warnings. Therefore, the remaining problem appears to concern the message passing system. Most likely, a message was misrouted, leaving the nodes in the network waiting indefinitely for the misrouted message. The next program development step would be to debug the message passing system.

IX. Chapter 9: Conclusions and Recommendations

The main conclusions that can be drawn from this study concern the implementation of concepts found in the parallel processing literature on the Intel Hypercube and its operating system.

The Intel Hypercube is one of a recent generation of parallel computers using cube architectures. The main purpose of these computers, according to Seitz, is to increase the speed with which problems can be solved. He states that:

highly concurrent systems of this type are an effective means of achieving faster and less expensive computing in the near future. (Seitz, 1985: 22)

The experience during the course of this project indicates that such is the case with the Intel Hypercube. This creates difficulties when attempting to use the Intel Hypercube to achieve an objective other than increased computational speed.

The purpose of this project was to utilize the built in redundancies of a parallel architecture to achieve fault tolerance. The operating system of the Intel Hypercube is designed with efficiency and speed as the primary goals. Fault tolerance was apparently not considered as a key design parameter.

The result is that a great deal of expertise and programming at low levels in the software, including assembly language, is necessary in order to implement the reconfigurable CCC which is the subject of this report. A particular problem is with the error notification system employed by the Intel Hypercube. The error system is designed to stop execution and notify the user of the problem. The purpose of this is to avoid as much as possible, the presentation to the user of fallacious or misleading computational results. Since most users are expected to adapt existing programs to the hypercube, the focus in the Intel system is on errors in parallel processing which would not have occurred in the serial processing implementation.

The error system is therefore not designed to deal with a user who wishes to continue a computation after encountering an error. The user for which the system is designed wishes to stop computation upon encountering an error. In the fault tolerant application, the whole point is precisely to continue execution in the presence of an error.

The limited time available made extensive software modifications infeasible, given the level of expertise available. However, the modifications which have been implemented on the Intel ring demonstration program show that it is indeed possible to simulate fault tolerance on the Intel Hypercube.

While it would certainly be convenient to use a machine specifically designed for redundancy, one cannot rely on such a machine being available when needed. If it is assumed that the parallel processing industry's focus on speed continues, then further work on reconfigurable cube-connected cycles will have to work around the existing equipment's limitations as was the case here.

Given the importance of reliability in space operations, work on redundancy using parallel architectures should continue in spite of equipment limitations. In continuation of this study, the next step would be to debug the inability of the existing program to continue message passing after reconfiguration.

Once this has been accomplished, the next logical step would be implementation of either the global or the local reconfiguration schemes for the CCC.

The global reconfiguration method would appear to be capable of implementation in the hypercube's addressing system between the node processors. The local reconfiguration method requires switches. Assuming it is undesirable to interpret switches to mean a hardware modification for reasons cited in Chapter 6, the local reconfiguration's switching requirements would most likely be met through software in the cube manager.

This last point, concerning the cube manager relates to one other recommendation. The error system on the hypercube

would have to be modified to enable the cube manager to take the appropriate corrective action, without stopping the execution of the entire program. As stated in Chapter 6, this would entail the writing, by the programmer, of a user-written exception handler.

In summary, progress on this problem has been made, but usable results will require more detailed software design work to make use of the equipment which is available. In more general terms, this study encountered a gap between the inherent redundancy of parallel computers, and the actual capability to make use of the redundancy. There is a need to join the concepts of parallel computing with the technology of fault sensing and corrective action in a manner that will not interrupt a computer's operation.

Appendix

(iPSC Program Development Guide, 1986)

```
% more host.for
C .....
C
C      program host
C .....
C
C      This is the Host code (in Fortran) for the Ring demo.
C
C      It prompts the user for:
C      a) the length of a message to send around a
C         RING in the cube
C      b) the number of times the message is to go around
C         the RING.
C
C      It outputs:
C      a) a ring "count" each time the ring message goes past
C         node 0, and
C      b) the time it took the message to go around the ring
C         the specified number of times.
C .....
C .....
C
C      DECLARATIONS:
C
C      Declare & initialize CONSTANTS:
C
C      integer*4 NODEPID
C      integer*4 HOSTPID
C      integer*4 ALLNODES
C      integer*4 INITTYPE
C      integer*4 TIMEMSGSIZE
C      integer*4 CNTMSGSIZE
C      integer*4 INITMSGSIZE
C      integer*4 N
C
C
C      Declare iPSC System Functions used:
C
C      integer*4 copen
C
```

```

C      Declare program variables:
C
integer*4 c1,type,cnt,frnode,frpid
integer*4 msglen
integer*4 ringcount
integer*4 msgbuff(3)
integer*4 kaputtnode

C      Declare time variable:
C
real*4    ringtime

DATA NODEPID          /1/
DATA HOSTPID          /1/
DATA ALLNODES         /-1/
DATA INITTYPE         /10/
DATA TIMMSGSIZE       /4/
DATA CNTMSGSIZE /4/
DATA INITMSGSIZE      /8/
DATA N                /0/

C
C .....
C .....
C
C      MAIN CODE:
C
C
C      write (6,51)
51      format(' LOADING RING INTO CUBE ...')
C
C      load the cube:
C
do 55 N=0,29,1
call load('node', N, NODEPID)
continue
55      write(6,160)
160      format(' ')
161      write(6,161)
161      format(' PICK THE NODE YOU WANT TO PUT OUT OF COMMISSION ')
162      write (6,162)
162      format(' BUT DO NOT PICK 0 !; PICK FROM 1 TO 29 !! ')
166      write (6,166)
166      format(' -- THE REASON YOU CANNOT PICK 0 IS THAT IT ')
167      write (6,167)
167      format(' COMMUNICATES WITH THE CUBE MANAGER . . . ')
181      write(6,181)
181      format(' ')
182      write(6,182)
182      format('if the PROGRAM HANGS-UP - press SHIFT/BACK-SPACE ')
183      write(6,183)
183      format('then RESET THE CUBE with a load -c COMMAND')
163      read(5,163) kaputtnode
163      format (13)
call lkill(kaputtnode,NODEPID)
call load('node',30,NODEPID)

```

```

c      Open a channel for the host-to-node-0 communications.
      c1 = copen(HOSTPID)

c*****
c
c      BEGIN MAIN PROGRAM LOOP:
c
10      write (6,100)
100     format(' ***** READY *****
1*****')

c      get the number of times to go around the ring:
      write(6,101)
101     format(' Number of times to go around the ring (neg.
1value quits): ')
      read(5,102) ringcount
102     format(i7)
c      ringcount = 1

c      If ringcount is negative exit HOST program:
c      if (ringcount .lt. 0) goto 600

c      Include ringcount in the message to the RING:
      msgbuff(1) = ringcount

c      get the message length:
      write (6,201)
201     format(' Length of Ring message in bytes (0-16384): ')
      read (5, 202) msglen
202     format(i5)
c      msglen = 2

c      Include msglen in the message to the RING:
      msgbuff(2) = msglen
      msgbuff(3) = kaputtnode

```

```

c      ship the message buffer off to node 0:
      call sendmsg(c1, INITTYPE, msgbuff, INITMSGSIZE, 0,
>      NODEPID)

c      Get the current ring count from node 0 and report
c      to user:

      do 400 i=1, ringcount

      call recvmsg(c1, type, msgbuff, CNTMSGSIZE, cnt, frnode,
>      frpid)

      write (6,310) msgbuff(1)
      format('Ring count: ',15)
310
400      continue

c      Get the RING time from node 0 & report to user:

      call recvmsg(c1, type, msgbuff, TIMEMSGSIZE, cnt, frnode,
>      frpid)

      ringtime = real(msgbuff(1))/1000.00

      write (6,306) ringtime
306      format(/,' Ring time :',F9.2,' secs.')

      goto 10

c
c      END OF MAIN PROGRAM LOOP.
c
c.....

c.....
c
c      CLEAN UP TIME!
c
600      write (6,601)
601      format(' CLEARING THE CUBE ...')

c      Kill RING processes in cube

      call lkill(-1,-1)
      call lwaitall(-1,-1)

      write (6,701)
701      format(' ***** DONE *****
1*****')

      end

c
c.....

```

more node.for

program node

This is the NODE part of the RING demo Program.
Node 0 will play the role of "controller" node.
It waits for a message from the host telling it:
a) the number of times to go around the RING, and
b) the length of the message to send around.
It then sends a message of the desired length to node 1 and
"controls" how many times the message goes around the RING.
At the end, Node 0 reports back to the Host the time it took
the message to go around the RING.
All the other nodes wait for a message and then
pass it on to the next node in the RING.

DECLARATIONS:

Program CONSTANTS:

integer*4 HOSTNID
integer*4 HOSTPID

integer*4 INITTYPE
integer*4 NODETYPE
integer*4 TIMETYPE
integer*4 COUNTTYPE

integer*4 INITSIZE
integer*4 TIMESIZE
integer*4 COUNTSIZE

```

integer*4 MAXMSGSIZE

integer*4 NOTBUSY
integer*4 NUMNODES

c      IPSC System Calls used:

integer*4 copen, status, mynode, mypid, cubedim
integer*4 clock

c      Program variables:

integer*4 hostchan, nodechan
integer*4 i, count, ringcount
integer*4 msglen
integer*4 nextnode, nextpid
integer*4 msgbuff(4096)
integer*4 ownnode, ownpid
integer*4 rtype, rcnt, rnode, rpid

c      Timing variables:

integer*4 starttime, ringtime

data HOSTNID      /-32768/
data HOSTPID      /1/

data INITTYPE     /10/
data NODETYPE     /20/
data TIMETYPE     /30/
data COUNTTYPE    /40/

data INITSIZE     /12/
data TIMESIZE     /4/
data COUNTSIZE    /4/
data MAXMSGSIZE   /16384/

data NOTBUSY      /0/
data NUMNODES     /30/

```

```

.....
*
*      MAIN CODE:

```

```

*
*
c      Each process identifies the node its running on and its pid:
      ownnode = mynode()
      ownpid  = mypid()

c      Each process determines the node id & and the pid of the next
c      node in the RING:
      nextnode = mod(ownnode + 1, 30)
      nextpid  = ownpid

      if(ownnode.eq.0) then
.....
*
*      BEGIN NODE 0 CODE:
*
c      Open channels for communicating with both the next node in
c      the RING (node 1) & the host:
      nodechan = copen(ownpid)
      hostchan = copen(ownpid)
.....
*
*      NODE 0 MAIN LOOP:
*
10  > call recvw(hostchan, INITTYPE, msgbuff, INITSIZE, rcnt,
      rnode, rpid)

      ringcount = msgbuff(1)
      msglen    = msgbuff(2)
      kaputtnode = msgbuff(3)

      starttime = clock()

      do 400 i=1,ringcount
      nextnode = mod(ownnode + 1, 30)
      if (nextnode.eq.kaputtnode) nextnode=30
      if (ownnode.eq.30) nextnode = mod(kaputtnode + 1, 30)
      call sendw(nodechan, NODETYPE, msgbuff, msglen,

```

```

>         nextnode, nextpid)
>         call recv(nodechan, NODETYPE, msgbuff, msglen, rcnt,
>             rnode, rpid)
c         As soon as the host channel is not busy report the
c         current count to the HOST:
200         if (status(hostchan).eq.NOTBUSY) goto 300
>             call flick()
>             goto 200
300         continue
>         count = 1
>         call send (hostchan, COUNTTYPE, count, COUNTSIZE,
>             HOSTNID, HOSTPID)
400         continue
>         ringtime = clock() - starttime
>         call sendw(hostchan, TIMETYPE, ringtime, TIMESIZE,
>             HOSTNID, HOSTPID)
>         goto 10
*
*         END NODE 0 MAIN LOOP.
*
* .....
*         END OF NODE 0 CODE.
*
* .....
>         else
>
> .....
>
>         BEGIN OTHER NODES' CODE:
c         All other nodes wait for a value from their left hand
c         neighbor, and pass it to their right hand neighbor.
c

```

```

c      They only have to open one channel for communication:
      nodechan = copen(ownpid)
.....
*
*      BEGIN OTHER NODES' MAIN LOOP:
*
20      call recvw(nodechan, NODETYPE, msgbuff, MAXMSGSIZE, rcnt,
>         rnode, rpid)
>      call sendw(nodechan, NODETYPE, msgbuff, rcnt, nextnode,
>         nextpid)
      goto 20
*
*      END OTHERS' MAIN LOOP.
*
.....
*
*      END OTHERS' CODE.
*
.....
      endif
      end
*
*      ... OF PROGRAM CODE.
*
.....

```

Bibliography

- Banerjee, Prithviraj et al. "Reconfigurable Cube-Connected Cycles Architectures," Digest of Papers of the 16th Annual International Symposium on Fault-Tolerant Computing Systems. 286-291. Washington, D.C.: IEEE Computer Society Press, 1986.
- Behmann, Francois F. and George Y. Nawar, TELESAT CANADA, Ottawa. "Effective Space Systems with Optimized Protection," 1985 Proceedings, Annual Reliability and Maintainability Symposium, IEEE. 191-196. 1985.
- Bounds, D. G. Physics for Travelling Salesmen: Some New Approaches to Combinatorial Optimization. March, 1986. Royal Signals and Radar Establishment, Malvern, England. (Memorandum No. 3945). (AD-A169547). Copyright controller, HMSO, London, 1986.
- Butz, J. S. Jr. "Electronic Device Simulates Processes of Human Brain," Aviation Week including Space Technology, Vol. 69 No. 1: 60-71 (July 7, 1958).
- Cooper, Leon N. Distributed Memory. Center for Neural Science and Physics Department, Brown University, Providence, Rhode Island. (Contract No. DAAG29-84-K-0202) (AD-A153364) (March 13, 1985).
- Evans, David D. and Maj Ralph R. Gajewski. "Expanding Role for Autonomy in Military Space," Aerospace America: 74,77 (February 1985).
- Fukushima, Kunihiro and Sei Miyake. "Neocognitron: A New Algorithm for Pattern Recognition Tolerant of Deformations and Shifts in Position," Pattern Recognition, Vol. 15 No. 6: 455-469 (1982).
- Gjermundsen, Lt Edward I. "Satellite Autonomy Enhances Space System Survivability," Defense Systems Review: 27-31 (February, 1984).
- Greene, Jonathan W. and Abbas El Gamal. "Configuration of VLSI Arrays in the Presence of Defects," Journal of the Association for Computing Machinery, Vol. 31 No. 4: 694-717 (October, 1984).
- Hopfield, John J. and David W. Tank. "Computing with Neural Circuits: A Model," Science, Vol. 233: 625-633 (8 August, 1986).

iPSC Program Development Guide. Order No. 310611-001. Intel Scientific Computers, Intel Corporation, Beaverton, OR, November, 1986.

iPSC User's Guide. Order No. 175455-004. Intel Scientific Computers, Intel Corporation, Beaverton, OR, April 1986.

Kabriskey, Matthew. A Proposed Model for Visual Information Processing in the Brain. Urbana, Illinois: University of Illinois Press, 1966.

----- . Personal Interviews. Air Force Institute of Technology, Wright-Patterson Air Force Base, Dayton, Ohio, 1 April through 30 April 1987.

Linsker, Ralph. "From Basic Network Principles to Neural Architecture: Emergence of Spatial-Opponent Cells," Proc. Nat'l. Acad. Sci. U.S.A., Vol. 83: 7508-7512 (October 1986).

----- . "From Basic Network Principles to Neural Architecture: Emergence of Orientation-Selective Cells," Proc. Nat'l. Acad. Sci. U.S.A., Vol. 83: 8390-8394 (November 1986).

----- . "From Basic Network Principles to Neural Architecture: Emergence of Orientation Columns," Proc. Nat'l. Acad. Sci. U.S.A., Vol. 83: 8779-8783 (November 1986).

Maddox, John. "Simulating Memory by Numbers," Nature, Vol. 325: 11 (1 January, 1987).

Markland, Robert E. Topics in Management Science. New York, John Wiley & Sons, Inc., 1983.

Messner, Richard A. and Harold H. Szu. "An Image Processing Architecture for Real Time Generation of Scale and Rotation Invariant Patterns," Computer Vision, Graphics, and Image Processing, 31: 50-66 (1985).

Notes. iPSC Concurrent Programming Workshop. Intel Scientific Computers, Intel Corporation, Beaverton, OR, 1-3 October 1986.

Preparata, Franco P. and Jean Vuillemin. "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," Communications of the ACM, Vol. 24 No. 5. 300-309. New York: Association for Computing Machinery, May, 1981.

Rich, Elaine. Artificial Intelligence. New York, McGraw-Hill Book Company, 1983.

Seitz, Charles L. "The Cosmic Cube," Communications of the ACM. Vol. 28 No. 1. 22-33. New York: Association for Computing Machinery, January, 1985.

Stone, Harold S. "Parallel Processing with the Perfect Shuffle," IEEE Transactions on Computers, Vol. C-20 No. 2. 153-161. New York: IEEE, February, 1971.

Ulsamer, Edgar. "The Military Imperatives in Space," Air Force Magazine: 92-98 (January, 1985).

Willis, D. G. Plastic Neurons as Memory Elements. Work carried out under Lockheed General Research Program, March 1959, Lockheed Missiles and Space Division, Lockheed Aircraft Company (LMSD-48432) (AD-217265).

VITA

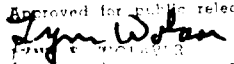
Captain Gil Zilberstein was born on 29 November 1955 in Havana, Cuba. He graduated from high school in Brooklyn, New York in 1972 and attended the City College of the City University of New York, from which he received the degree of Bachelor of Arts in Urban Geography in June, 1976. Upon graduation from Officer Training School in December, 1981, he commenced active duty, training to be the Deputy Missile Combat Crew Commander of a Titan II missile launch crew. In October, 1982 he was certified to perform Combat Crew Alert duties in the 532nd Strategic Missile Squadron, McConnell AFB, Kansas. Beginning in January, 1983, he attended classes in Aeronautical Engineering during his off-duty time, at Wichita State University, Wichita, Kansas. He continued attending these classes and performing missile alert crew duties in the 532nd Strategic Missile Squadron until entering the School of Engineering, Air Force Institute of Technology, in May, 1986.

Permanent address: 140-20 Benchley Pl.

Bronx, New York 10475

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GSO/ENG/87D-1			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
			WORK UNIT ACCESSION NO.		
11. TITLE (Include Security Classification) SIMULATION OF FAULT TOLERANCE IN A HYPERCUBE ARRANGEMENT OF DISCRETE PROCESSORS					
12. PERSONAL AUTHOR(S) Gil Zilberstein, Capt, USAF					
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1987 December	
15. PAGE COUNT 83					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Parallel Processing, Redundancy, Redundant Components Processing		
12	09				
12	01				
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Thesis Advisor: Dr. M. Kabrisky					
Approved for public release: 1AW AFR 190-17.  Lynn W. Woban Dean for Education and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. M. Kabrisky, Professor of Engineering			22b. TELEPHONE (Include Area Code) (513) 255-3430		22c. OFFICE SYMBOL AFIT/ENG

Abstract:

The purpose of this study was to implement a technique for fault-tolerant parallel computation on the Intel Corporation's Hypercube computer. This work was motivated by the recent progress in parallel computation and neural network techniques.

This study focuses on the implementation of one particular type of parallel processing architecture on the Intel Hypercube. The architecture in question is known as the cube-connected cycle (CCC). This architecture is used as a basis for a reconfiguration scheme known as reconfigurable cube-connected cycles. The aim of this reconfiguration is to build a parallel computing system with fault tolerance capability.

Implementation of this technique on the Intel Hypercube was by simulation. The loading of only part of the hypercube's available nodes, holding the remaining nodes in reserve was accomplished, followed by a simulation of the replacement of a deactivated node with a spare node.

Conclusions are reached regarding the suitability of the Intel machine for fault tolerance experiments versus the rapid computation for which it was designed. Recommendations are made regarding the next logical steps in continuation of the work presented in this study.

END
DATE
FILM
4-88
DTIC