Fault Tolerance for VLSI Multicomputers

by

Yuval Tamir

Ph.D. Dissertation

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

August 1985

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
AUG 1985		**00-00-1985 to 00-00-1985**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Fault Tolerance for VLSI Multicomputers	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The performance requirements of future high-end computers will only be met by systems that facilitate the exploitation of the parallelism inherent in the algorithms that they execute. One such system is a multicomputer that consists of hundreds or thousands of VLSI computation nodes interconnected by dedicated links. Some important applications of high-end computers, such as weather forecasting, require continuous correct operation for many hours. This requirement can only be met if the system is fault-tolerant, i.e., can continue to operate correctly despite the failure of some of its components. This dissertation investigates the use of fault tolerance techniques to increase the reliability of VLSI multicomputers. Different techniques are evaluated in the context of the entire system, its implementation technology, and intended applications. A proposed fault tolerance scheme combines hardware that performs error detection and system-level protocols for error recovery and fault treatment. Practical design and implementation tradeoffs are discussed. A fault-tolerant system must identify erroneous information produced by faulty hardware. It is shown that a high probability of error detection can be achieved with self-checking nodes implemented using duplication and comparison. The requirements for detecting errors caused by hardware faults are: (1) the comparator is fault-free, and (2) the functional modules never produce identical incorrect outputs. Requirement (1) is fulfilled with a self-testing comparator that signals its own faults during normal operation. An implementation of such a comparator using MOS PLAs is discussed. Requirement (2) is fulfilled with two modules that are implemented differently so that, although they perform identical functions, they have a low probability of failing simultaneously in exactly the same way. Low-cost techniques for implementing such modules are presented. The detection of an error implies that the state of the system has been corrupted. In order to recover from the error and resume correct operation, a valid system state must be restored. A low-overhead, application-transparent error recovery scheme for multicomputers is presented. It involves periodic checkpointing of the entire system state, using protocols that ensure that the saved states of all the nodes are consistent, and rolling back to the last checkpoint when an error is detected.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**152**	

Fault Tolerance for VLSI Multicomputers

Fault Tolerance for VLSI Multicomputers

Yuval Tamir

Abstract

The performance requirements of future high-end computers will only be met by systems that facilitate the exploitation of the parallelism inherent in the algorithms that they execute. One such system is a *multicomputer* that consists of hundreds or thousands of VLSI computation nodes interconnected by dedicated links. Some important applications of high-end computers, such as weather forecasting, require continuous correct operation for many hours. This requirement can only be met if the system is *fault-tolerant*, i.e., can continue to operate correctly despite the failure of some of its components. This dissertation investigates the use of *fault tolerance* techniques to increase the reliability of VLSI multicomputers. Different techniques are evaluated in the context of the entire system, its implementation technology, and intended applications. A proposed fault tolerance scheme combines hardware that performs error detection and system-level protocols for error recovery and fault treatment. Practical design and implementation tradeoffs are discussed.

A fault-tolerant system must identify erroneous information produced by faulty hardware. It is shown that a high probability of error detection can be achieved with self-checking nodes implemented using duplication and comparison. The requirements for detecting errors caused by hardware faults are: (1) the comparator is fault-free, and (2) the functional modules never produce identical incorrect outputs. Requirement (1) is fulfilled with a *self-testing* comparator that signals its own faults during normal operation. An implementation of such a comparator using MOS PLAs is discussed. Requirement (2) is fulfilled with two modules that are implemented differently so that, although they perform identical functions, they have a low probability of failing simultaneously in exactly the same way. Low-cost techniques for implementing such modules are presented.

The detection of an error implies that the state of the system has been corrupted. In order to recover from the error and resume correct operation, a valid system state must be restored. A low-overhead, application-transparent error recovery scheme for multicomputers is presented. It involves periodic checkpointing of the entire system state, using protocols that ensure that the saved states of all the nodes are consistent, and rolling back to the last checkpoint when an error is detected.

Carlo H. Séquin Aug 20, 1985

Carlo H. Séquin
(Committee Chairman)

Dedicated to my parents

Pinchas Tamir and Ruth Tamir

for raising me to be a nonconformist and

for their unconditional support.

Acknowledgements

On my trek towards a Ph.D. I was not alone. The advice, encouragement, and friendship of my research advisor, Carlo Séquin, were not only essential for my dissertation research, but also *the* major factor in making my six years at Berkeley both productive and enjoyable. Many of the ideas presented in this dissertation were the result of long discussions with Carlo. The quality of the dissertation itself was greatly improved by his comments that followed meticulous reading of every page. Working with Carlo has been a pleasure due to his openness to new ideas, willingness to consider different approaches to solving problems, and interest in a wide variety of areas. I particularly appreciate his unique ability to be a demanding advisor, yet treat his graduate students as valued colleagues rather than subordinates. I am grateful for the freedom he gave me to pursue my own interests and approaches to solving problems, as well as for the time he spent working with me to ensure that I would be proud of the final results.

I would like to thank Chittor Ramamoorthy and Kjell Doksum for serving on my Thesis Committee. Chittor Ramamoorthy provided useful comments on the dissertation and was a valued advisor on technical and professional matters throughout my years at Berkeley.

The research that culminated with this dissertation was not a large group project; only Carlo Séquin and myself were involved. However, in our work we were inspired by the exciting atmosphere at the U.C. Berkeley Computer Science Division where there are always numerous ongoing creative and productive research projects. I would like to thank all the members of the division for contributing to this environment.

Sudhakar Reddy introduced me to the field of fault-tolerant computing when I was an undergraduate student at the University of Iowa. Without his influence, I would never have chosen this field for my Ph.D. research.

This work was supported by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3803, monitored by Naval Electronic System Command under Contract No. N00039-81-K-0251. During my first year of graduate studies I was supported by a University of California Graduate Fellowship. In 1983 I was supported, in part, by a State of California MICRO Fellowship.

Table of Contents

Chapter One
Introduction

Since the development of the first electronic computer, advances in technology have lead to many orders of magnitude improvements in the available processing speeds. As higher processing speeds became available, users continued to discover new applications demanding yet faster processors and motivating the development of ever more powerful systems. At each point in time, important applications seem to require processors that are an order of magnitude faster than the fastest available systems [Fuss84].

As a result of the constant demand for high-speed processing, each generation of computers includes a group of large expensive systems, called "supercomputers," in which technology is pushed to its limits in order to implement computers that are only "one generation behind the computational needs of certain key industries" [Linc82]. At the present time, the technology used for implementing supercomputers has reached a point where significant improvements in its raw speed will not be possible due to fundamental physical constraints (such as the speed of light). Significant enhancements in the computational power of high-end computers will only be possible if the parallelism inherent in the algorithms that are executed on these computers is exploited.

Some of the computations performed on supercomputers, such as large circuit simulations, weather forecasting, or aeronautical design, may require continuous operation of the system for many hours (or even days) [Fern84]. In order to have high confidence in the validity of the results obtained by such computations, the system must be highly reliable. Thus, supercomputers require high reliability as well as high performance.

Unfortunately, the reliability of a system is inversely proportional to its size and complexity. In order to achieve the maximum possible performance, supercomputers are large complex systems with many thousands of components [Russ78]. The probability that one component out of many thousands will fail is relatively high, even if each component by itself is very reliable. If any single component failure can cause the system to produce incorrect results, high system reliability cannot be achieved. Thus, supercomputer systems must be able to continue correct operation despite the failure of individual components, *i.e.*, they must be *fault-tolerant*.

Due to advances in VLSI technology, a general purpose computing system composed of thousands of microcomputers is now economically feasible. In such a system, high performance may be achieved for computational tasks which consist of many subtasks that can run simultaneously on different microcomputers. High reliability may be achieved using fault tolerance techniques by exploiting the fact that the components of the system (the microcomputers) are "intelligent" and can adapt their "behavior" to changes in the system status caused by "faults."

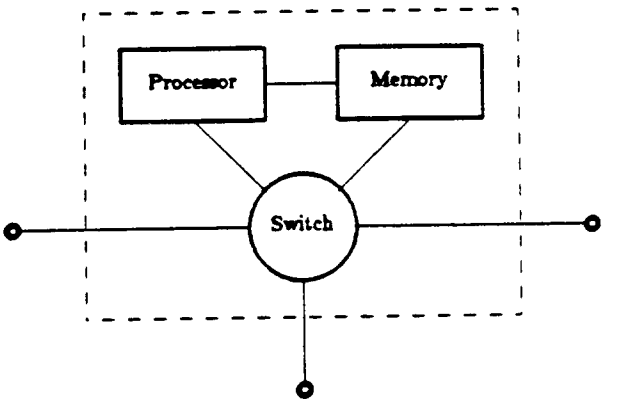


Fig. 1.1: A Computing Element (Node)

A possible building block for future supercomputer systems is a *computing element* composed of a processor, memory, and switching circuitry [Desp78, Barr83]. A system that consists of a large number of such computing elements interconnected by high-speed dedicated links is called a *multicomputer* [Fuji82, Séqu83]. In a multicomputer there is no single hardware component that is used by all (or a large number of) the computing elements and which can become a performance bottleneck and/or a critical resource whose failure results in system failure. Hence, a multicomputer is especially well suited for reliability enhancements using fault tolerance techniques [Tami83].

This thesis deals with the implementation of fault tolerance in supercomputers implemented as VLSI multicomputers. Since the main goal of a supercomputer is high-performance, special emphasis is placed on minimizing the performance degradation caused by the fault tolerance techniques. In order to continue correct operation despite a failed component, the resulting errors must be detected, a valid system state must be recovered from the system state corrupted by the fault, and the system must be reconfigured to avoid using the faulty component. This thesis focuses on the error detection and error recovery phases.

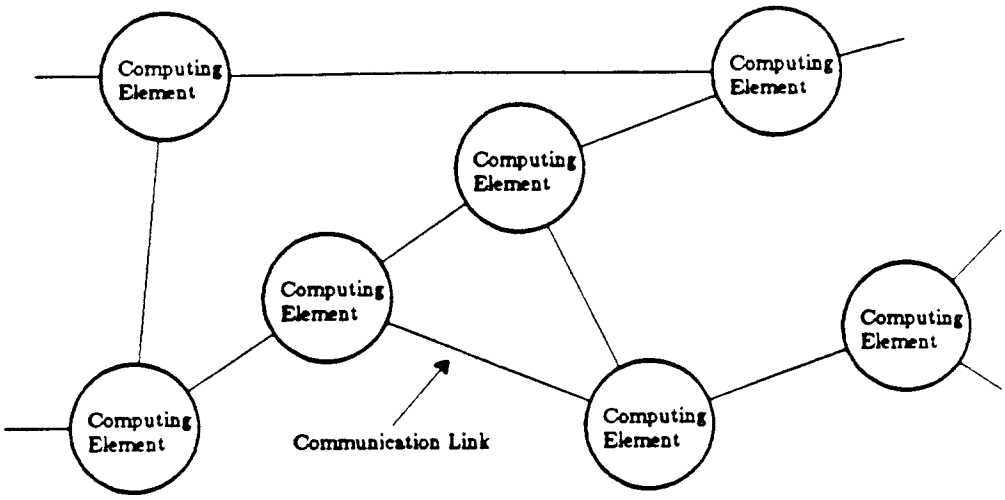


Fig. 1.2: A Multicomputer

1.1. Achieving High Performance Using Parallelism

Over the past thirty-five years the execution time of simple instructions on electronic computers has decreased by four or five orders of magnitude. This decrease has been due mainly to improvements in technology. Currently, the clock cycle time on the fastest supercomputers is less than ten nanoseconds [Hwan84]. In order to achieve the required high switching speed, the circuits used in these systems are characterized by high power dissipation. For example, in the CYBER 205 each one of the 1760 printed circuit boards dissipates 750W [Kozd80, Linc82], and the total power required by the CRAY-1 is more than 100KW [Russ78].

Two major factors limit further reductions in processor clock speeds: (1) signal propagation delays and (2) power dissipation. In vacuum, light travels approximately one foot per nanosecond. Since the propagation speed of signals on wires or inside chips is lower, severe timing problems can occur in synchronous circuits operating with clock cycles of a few nanoseconds due to related signals traveling through paths of different lengths with different delays. Removal of the heat dissipated by high-speed circuits requires expensive, complex cooling technologies. Even today, cooling is considered by some to be one of the most difficult problem in supercomputer system design [Cray74]. Unless there is a breakthrough in technology, the problem can be expected to get worse in future supercomputers that will use faster circuits.

Based on the above considerations, it seems unlikely that another order of magnitude increase in computer processing speed will be achievable by simply enhancing the raw

speeds of circuits. Instead, the greatest potential for achieving significantly higher processing speeds is in techniques that exploit parallelism. The basic tradeoff here is simple: since it is getting more and more difficult to implement circuits that operate faster, the amount of information that is processed by an individual gate cannot be significantly increased. On the other hand, since it is becoming cheaper to implement more circuitry (gates) in a system, the amount of information processed by the *system* can be increased if different parts of the information can be processed by different gates simultaneously.

Parallelism has always been used in computer systems for achieving high performance. A simple example is a "store" instruction that moves all the bits of a register to memory simultaneously rather than one bit at a time. At a higher level, pipelining techniques are currently used in all computers. With pipelining, at each point in time the processor contains several instructions that are at different stages of their execution. As a result, while the next instruction is being fetched, parts of the processor that are not used for instruction fetch (such as the ALU) are still doing useful work on some previous instruction.

Most of todays supercomputers (e.g. the CRAY-1 [Russ78]) are "vector machines" where parallelism is exploited at an even higher level. Single instructions operate on vectors of numbers rather than on individual elements, thereby achieving speedups proportional to the size of the vectors. The vector instructions are, of course, always used for vector operations specified by the programmer. In addition, all vector machines use sophisticated "vectorizing compilers" that can convert code segments which are not specified as simple vector operations into the vector operations supported by the hardware [Kuck84].

The main limitation of vector (or array) machines is that all the processing elements always perform the same operation. In many cases execution can be speeded up if the different processing elements are able to execute different code segments simultaneously. In order to exploit this potential for high performance, supercomputers of the future will consist of multiple independent processing elements.

Conceptually, there are two major types of systems with multiple independent processing elements: ***multiprocessors*** and ***multicomputers***. In a multiprocessor all the processing elements are connected to a shared memory which they can use to

communicate with each other (Fig. 1.3). A multicomputer consists of a number of *computing elements*, each of which is a complete computer that contains local memory as well as a processor. The computing elements do not share memory and communicate by sending messages through an interconnection network (Fig. 1.4).

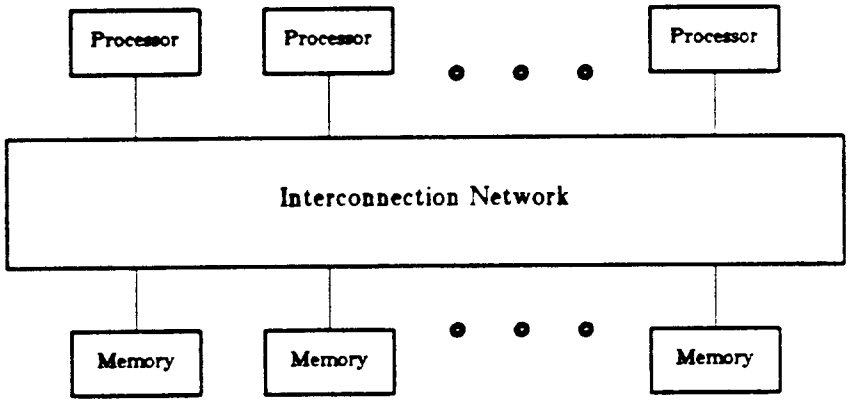


Fig. 1.3: A Multiprocessor

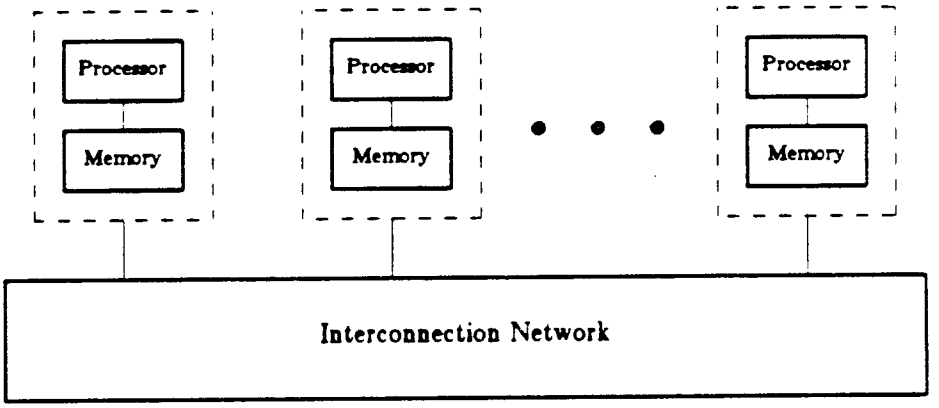


Fig. 1.4: A Multicomputer

The simplest way to connect multiple processors to a shared memory or to interconnect multiple computing elements is to use a common bus or Ethernet. This scheme is used in most of the current commercially available multiprocessors [Jone83] and multicomputers [Katz82]. Unfortunately, the use of a common bus with a finite bandwidth limits the maximum number of processing elements that can be effectively utilized in the system. While a bus may be the best choice for a system with, say, ten processing elements, it is clearly not appropriate for a system with hundreds of elements.

The key to implementing a high-bandwidth interconnection network that can support a large number of processing elements is to ensure that different "messages" can be transmitted on different "wires" so that the transmission of multiple messages can

occur in parallel. Interconnection networks that allow several units to communicate simultaneously are called *alignment networks* (*switching networks*)[Kuck78]. Many multiprocessor systems use alignment networks that allow N processors to communicate with N memory modules[Gott83]. Each processor can communicate with each one of the memory modules so that all the memory can be shared by all the processors. An example of such an alignment network is shown in Fig. 1.5.

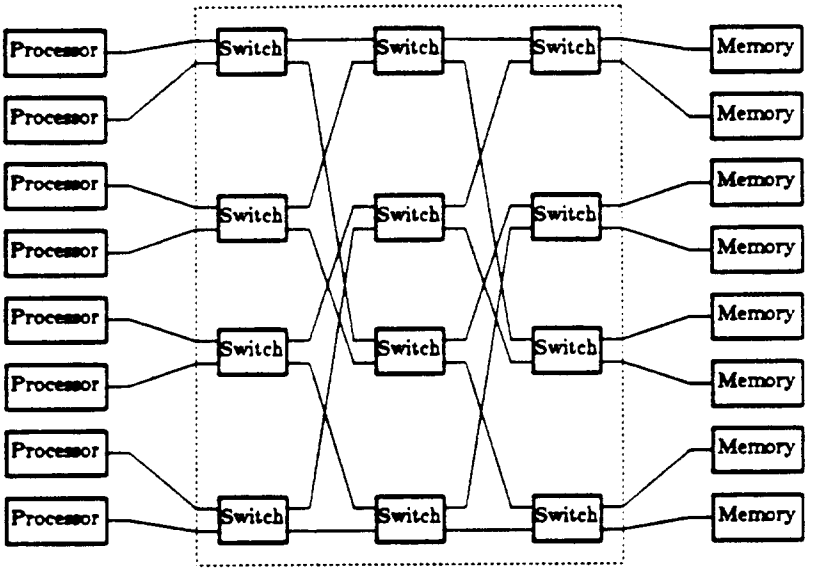


Fig. 1.5: An Alignment Network

In multiprocessors that use an alignment network of the form shown in Fig. 1.5 to interconnect the processors to memory modules, all memory accesses are performed through the interconnection network and the access time from each processor to each memory location is uniform. Since accessing memory through the network is expensive, a cache is added to each processor module so that a large percentage of the accesses are actually performed locally without traversing the network[Gott83]. The use of caches introduces the problem of ensuring that all the processors access a consistent version of the data. The problem occurs when one of the processors modifies data held in the cache of a second processor. Before the second processor attempts to read the data, the modification must be propagated to the shared memory and the data held in the second processor's cache must be invalidated to ensure that the data will be read from the shared memory. The protocols required to ensure *cache coherency*[DuBo82] increase the communication overhead and slow down the system[Kell84].

In multicomputer systems each processor has exclusive control over the memory in

its own computing element. Sharing of data is accomplished by exchanging messages and is under the control of the software. The local memory in a computing element is simpler and cheaper to implement than a cache so that it is practical to implement a larger and faster memory tightly coupled with each processor. As a result, if the application exhibits high locality so that there is relatively little sharing between processors, higher performance may be achievable in the multicomputer than in a multiprocessor.

It is possible to implement a multicomputer which provides uniform communication between every pair of computing elements. For example, all the computing elements can be on the same bus, or an alignment network of the form shown in Fig. 1.5 may be used. If an alignment network is used, instead of connecting N processors to N memory modules, as in a multiprocessor, N computing elements are interconnected by attaching the output port of each element to one end of the network and the input port to the other end.

An alternative approach to implementing a multicomputer is to use high-speed point-to-point dedicated links to connect each computing element to a small subset of all the computing elements in the system [Desp78, Séqu83, Kell84, Barr83]. Two computing elements connected by a dedicated link are called *neighbors*. Communication with a computing element that is not a neighbor is accomplished by transmitting a message to a neighbor that is closer to the final destination with instructions to forward the message to the final destination. This process is repeated until the final destination is reached. We call a multicomputer in which the computing elements are interconnected by point-to-point dedicated links a *PTPI* (point-to-point interconnection) multicomputer. In the rest of this thesis, unless otherwise specified, the term *multicomputer* will be used to denote a PTPI multicomputer (Fig. 1.2).

In a PTPI multicomputer communication with a neighbor is significantly faster than with other nodes. Communication between neighbors can be faster than communication between computing elements in systems that use a uniform interconnection network. Furthermore, communication between any two neighbors cannot affect (slow down) communication between other neighbors in the system. Based on the above characteristics, the PTPI multicomputer is particularly well-suited for applications that can be partitioned so that each computing element communicates mostly with a small number of other elements and has relatively little direct interaction with the majority of

computing elements in the system. For such applications, the PTPI multicomputer has the potential of higher performance than other types of multicomputer or multiprocessor systems constructed using similar technology.

1.2. Achieving High Reliability Using Redundancy

Over the past thirty-five years, the reliability of the basic components used to implement electronic computers has increased by several orders of magnitude. In the days of relays, vacuum tubes, and delay-line storage, it was considered a difficult task to simply keep the system operating for more than a few minutes [Aviz78]. With current VLSI components, whose failure rates are only a few hundred per billion *part* hours [Peat81, Budz82], systems are built whose expected down time is only a few minutes per year [Toy78] or with a failure rate of less than one failures per billion *system* hours [Hopk78].

Despite the low failure rates of the available components, the level of reliability desired for many of the current applications of computers cannot be achieved by simply relying on the high reliability of the components and allowing the system to fail whenever one of the components fails. For example, in a system that consists of 10,000 components, each with a failure rate of 500 per billion part hours, the probability that none of the components will fail during 100 hours of continuous operation is only 0.6.

As technology progresses, fewer chips are needed to implement a system with a given functionality and performance. However, the demand for ever more powerful computers for each application keeps up with technological developments. Thus, the additional functionality and performance per chip are often used to increase system performance rather than decrease the number of chips in the system. Hence, the reliability of systems normally used for a particular application area is likely to increase only to the extent that the reliability of the new, more powerful chips is higher than that of the previous generation of chips.

Over the past two decades, the reliability of chips has increased simultaneously with increases in their functionality and performance. For example, the reliability of current microprocessor chips is higher than the reliability of the first NAND gate chips. However, there are three major interrelated factors that limit reliability improvements achievable by technological developments alone: unexpected failure modes, incomplete testing, and

the use of state-of-the-art technology.

With any new technology there may be unexpected failure modes that defy detection using prevalent testing procedures or increase the chip's sensitivity to environmental factors. For example, smaller feature sizes in memory chips increased the sensitivity to alpha particles and cosmic rays [Brod80]. Testing procedures had to be modified for CMOS circuits since one of their major failure modes (stuck-open faults) was not usually considered with other technologies [Wads78]. Subtle problems that may make the testing of certain CMOS circuits unreliable have only recently been recognized [Redd83]. Thus, products that use state-of-the-art technology may have unexpectedly low reliability. This problem has been recognized by NASA and as a result, in computers used in spacecrafts, only proven (5-10 years old) technology is used [Renn78].

Due to their complexity, exhaustive functional testing of VLSI chips is impossible [Rasm82]. Manufacturers of VLSI chips perform partial testing of their chips based on the known likely failure modes. The tests used are often functional tests that are derived in an ad hoc way. In other cases the tests are based on restricted fault models, such as single stuck faults [Frie71], that do not cover all possible physical defects that can occur in VLSI circuits [Gali80, Koda80]. The result of the incomplete testing is that some of the chips that are delivered to the customers are faulty. The percentage of such faulty VLSI chips ranges from 0.1 percent for relatively simple chips that have been in production for a while to several percent for new complex chips [Peat81].

In implementing any system there is an important tradeoff between either using the most advanced chips for high performance and taking the risk that the resulting system will be unreliable, or using proven technology for high reliability and failing to achieve the highest possible performance. As mentioned above, for specialized critical applications only "proven" technology is used. Even in less critical systems, which do not include any special provisions for fault-tolerance, it is always necessary to be somewhat conservative in making this tradeoff in order to ensure that the resulting system will be usable. However, the use of the most advanced chips available can result in systems with lower price/performance ratios than their predecessors or in systems that achieve a level of performance that has not been achievable in the past. Thus, there are strong pressures to use state-of-the-art technologies for most applications. This is particularly true for supercomputers where, as discussed in the previous section, the demand for more powerful

computers always far exceeds the capabilities of the available systems.

The previous subsection discussed the use of the hardware resources of the system to process different parts of the information simultaneously, thereby increasing the overall throughput of the system. Since the required reliability from systems cannot be achieved by relying on the "raw" reliability of the hardware, another use of system resources is to perform *redundant* operations that increase the reliability of the system rather than its performance. The redundant operations may be performed by dedicated (redundant) hardware whose sole purpose is to check and/or correct the results produced by other, possibly faulty, hardware. It is also possible to use *redundancy in time* where the same hardware reexecutes the original operations, verifies the validity of the results, and attempts to correct invalid results.

Redundancy is currently used in most computer systems to increase their reliability. The simplest example is the use of a parity bit to detect erroneous information as it is retrieved from main memory. The memory dedicated to the storage of the parity bit with each byte or word is redundant hardware. Redundancy in time is used when, for example, the processor spends time calculating the CRC code for a block of data before storing it on disk. When the block is read, the CRC check bits are used to determine whether the data has been corrupted.

There is a wide range of choices as to where and how redundancy is used to increase the reliability of a system. The choices that have to be made include: hardware versus software, the extent of hardware redundancy, the granularity of hardware redundancy, the extent of time redundancy, and granularity of time redundancy.

The fault tolerance features of a computer system can be made entirely transparent to the software. For example, the output from the system is a majority vote on the outputs of three identical processors that operate in lock-step executing identical software [Plat80]. In this case the system can tolerate the failure of any one of the three processors. On the other hand, it is also possible to construct a system in which all the fault tolerant features are implemented in software. For example, the application program may periodically perform "acceptance tests" on intermediate results. If the test indicates an error, the last subtask may be reexecuted using a different procedure [Rand78]. The alternate procedure can attempt to use the hardware in a different way so that the error will not be duplicated.

If redundant hardware is used in the system in order to increase its reliability, an important design decision is what percentage of the system hardware is dedicated to increasing system reliability. A simple example is a memory system in which, using appropriate coding techniques, one redundant bit per word can provide error detection while several redundant bits per word can provide error correction.

The granularity of hardware redundancy can be at the level of individual gates or complete processors. An arbitrary logic circuit that can tolerate any single line stuck-at-zero or stuck-at-one can be implemented by a technique, called *quadded logic*, that requires quadrupling the number of gates and interconnections [Tryo62, Koha78]. Hardware redundancy at the level of a complete processor is used in some multiprocessor systems where each task is simultaneously executed on several processors and the results are compared to determine their correctness [Wens78].

The percentage of system processing time devoted to increasing the reliability of the system is another important parameter of any system. For example, an application may periodically perform low-cost "reasonableness" tests on intermediate results or, alternatively, the entire computation may be repeated, using a different algorithm, in order to provide a more accurate test of whether the results are correct.

When time redundancy is used, a fixed percentage of processing time for redundant operations may be used for a large number of short operations or a small number of large complex operations. For example, if the system uses periodic acceptance tests to detect errors, those tests could be performed after every few instructions, only when a procedure is about to return control to its caller, or only at the end of the entire program. The choice of the granularity of time redundancy cannot be based only on the probability of detecting errors. Even if one complex test at the end of the program has a higher probability of detecting errors, simpler intermediate tests might be chosen in order to fulfill requirements of low latency between error occurrence and detection or faster recovery when an error is detected.

The issues discussed above do not include all possible options that must be considered when choosing a scheme for enhancing system reliability. Rather, these are meant to be example of the *type* of issues that come up. In general, the choice of a particular scheme for using redundancy is a result of complex tradeoffs involving performance requirements, reliability requirements, available technology, cost, market

pressures, etc. This thesis discusses the options and tradeoffs for one particular type of system — a PTPI multicomputer implemented using VLSI technology.

1.3. Architectures of Future Supercomputers

The goal of high-end computer systems is to execute compute-intensive tasks quickly. At the present time one of the critical research issues in computer science is what type of architecture is most appropriate for computer systems whose goal is to achieve orders of magnitude greater performance than today's high-end systems. Proposed architectures include: bigger and faster vector uniprocessors, such as the Cray-1 [Russ78, Miur84], a small number of interconnected high-end vector uniprocessors [Widd80, Lars84], several hundred or a few thousand of the most powerful microprocessors available interconnected with each other [Desp78, Gott83], or tens of thousands of very small processors each performing simple (possibly bit-serial) operations [Hill81, Shaw84].

There are clear advantages to computers whose performance does not rely on a high degree of parallelism. With such systems there is no need to develop new parallel algorithms or sophisticated compilers that can extract the parallelism from programs that were written for sequential execution. However, as discussed earlier, there are fundamental limitations on the speed of logic circuits so that significant performance improvements in the future will require exploiting parallelism. Hence, future supercomputers will be multiprocessors or multicomputers. The only question that remains is whether it will be possible to develop algorithms and software to effectively utilize these systems.

While there is general agreement that it is possible to utilize a small number of processors effectively [Widd80, Lars84], it is unlikely that thousands of processors can be effectively used for all general purpose computation [Nico84]. On the other hand, for some important applications in scientific computations, simulation studies have shown that it is possible to utilize several hundred [Nico84] or a few thousand [Gott83] processors. For some applications in artificial intelligence it is claimed that hundreds of thousands of processors could be effectively utilized [Hill81, Shaw84].

The need to deal with the unreliability of the hardware is not usually considered when comparing different architectures for supercomputers. As discussed earlier, large systems consisting of tens of thousands of chips have a significant hardware failure rate.

Fault tolerance techniques must therefore be used to allow the system to continue correct operation despite hardware faults.

If the system is one large "monolith," it is difficult to design it in such a way that it can isolate any faulty component and reconfigure itself to continue normal operation without that component. It is here that a multiprocessor or multicomputer with a large number of processors has a distinct advantage. The "components" of such systems are processors which are capable of independent "intelligent" actions. One processor can detect that another is faulty and modify its behavior to allow the system to continue operating correctly. It is much more difficult for part of a large complex ALU to detect the failure of another part of the ALU and change its operation to compensate for that failure.

The fact that a system consists of multiple processors does not guarantee that it is easy to isolate a faulty component and reconfigure the system to continue operating without that component. For example, in a multiprocessor system where a multistage alignment network is used to interconnect N processors with N memories [Gott83] the switches used in the alignment network are not "intelligent." Therefore it may be difficult for the system to identify a faulty switch and for the other switches to accommodate that failure. Furthermore, in order to be able to tolerate a switch failure, the alignment network must have more than one path between each processor and each memory. This requirement leads to a more complex network with higher latency than in a network where there is only one path between each processor and each memory [Adam82].

It may also be difficult to implement fault tolerance in any multiprocessor or multicomputer system with a very large number of very small processors [Hill81, Shaw84]. If a node in the system contains a 32-bit microprocessor and thousands of bytes of memory, it is feasible to add to the node extra hardware and software that allows it to handle exceptional situations, such as the failure of another node. On the other hand, if the node is a small bit-serial processor with a few dozen bytes of memory, it necessarily has only a small repertoire of actions that it can take during *normal operation*. Such a node will have to be made many times more complex to provide it with the capabilities to diagnose other parts of the system and to modify its behavior accordingly. This may make the entire system infeasible.

Based on the previous paragraphs, we can conclude that no one type of computer

architecture will emerge as *the* clear choice for all applications. Instead, different architectures, utilizing different degrees of parallelism, will be used for different applications. Reliability considerations are likely to be just as important as performance requirements in determining the type of system to be used. In this context, a multicomputer system with several hundred or a few thousand nodes, each containing a relatively powerful microprocessor, has several distinct advantages: (1) Nodes are "disposable" since the loss of a few nodes does not significantly reduce the hardware resources of the system, as they would in a system with a small number of processors. (2) There is no central resource, such as the alignment network in a multiprocessor, whose performance is critical to the performance of the entire system. (3) The nodes are sufficiently powerful to handle exceptional conditions.

Recent experiments have demonstrated that, in terms of performance, a multicomputer system can be utilized effectively for important applications [Seit85]. Given the advantages of a multicomputer architecture for implementing fault tolerance, there is no doubt that this type of architecture will be used in a significant number of future supercomputers.

1.4. Thesis Organization

This thesis focuses on the use of fault tolerance techniques to increase the reliability of multicomputers implemented with VLSI and using point-to-point dedicated links for interprocessor communication. Effective implementation of highly reliable systems requires the use of a combination of hardware and software techniques carefully tailored to the technology as well as to the intended applications. Accordingly, the different fault tolerance techniques for the multicomputer are considered in the context of the entire system rather than in isolation. In each chapter relevant previously published work is reviewed and the choice of the approach most appropriate for a VLSI multicomputer is described.

Technical discussions require the use of precise terminology whose meaning is agreed upon by all. Unfortunately, there is no "agreed upon" terminology for discussing "fault-tolerant" computer systems. In Chapter 2 the terminology that is used throughout this thesis is introduced. In addition, Chapter 2 includes a discussion of the fundamental issues of the causes of faults in digital computers and fault modeling.

In order to be able to continue correct operation despite errors produced by faulty hardware, the system must identify the erroneous information. Various *error-detection* techniques for use in a multicomputer are discussed in Chapter 3. It is shown that, given the need for effective error-detection, computing elements that report their failure to the rest of the system at the same time they produce erroneous information, are the most desirable building blocks for multicomputers. With VLSI technology, such *self-checking* computing elements are best implemented using duplicate functional modules that operate in lock-step, performing identical operations on the same inputs. The outputs of the two modules are continuously compared and any mismatch signals an error.

No error detection scheme can guarantee that all hardware errors will be detected. One of the potential weaknesses of duplication and comparison for implementing the self-checking computing elements is that if the comparator fails, a subsequent mismatch between the outputs of the two functional modules is not detected and erroneous information is accepted as correct by the rest of the system. It is therefore imperative that faults in the comparator be detected soon after they occur so that the rest of the system can be informed that the supposedly self-checking computing element has lost its self-checking capabilities. This requirement can be fulfilled by using a *self-testing* comparator that signals its own faults during normal operation. The implementation of such a comparator is discussed in Chapter 4. The proposed implementation uses MOS PLAs and is shown to be self-testing with respect to a new fault model that represents many of the possible physical defects that were not considered in previously published models.

When an error occurs, the state of the system is corrupted and correct operation cannot be resumed unless a valid system state is restored. Furthermore, if the faulty component that caused the error remains in the system after a valid system state is restored, it is likely to cause further errors and eventual system failure. Hence, after an error is detected, as part of the process of "recovering" from the error, the faulty component must be located and isolated from the rest of the system. Chapter 5 includes a review of several techniques for locating a faulty component that has caused an error, restoring a valid system state, and reconfiguring the system so that it can operate without the faulty component. A new error recovery scheme for multicomputers is proposed. This new scheme is particularly well suited for a multicomputer executing non-interactive

applications. It involves periodic checkpointing of the entire system state and rolling back to the last checkpoint when an error is detected. No restrictions are placed on the actions of the application tasks, and, during normal computation, there are no complex communication protocols of the type required by most other schemes.

This thesis does not provide a complete detailed design of a high-performance fault-tolerant multicomputer. There are many implementation details that have not been considered. For example: power distribution, clock signal distribution, synchronization between computing modules, and packaging. In addition, the topology of the interconnections between computing modules may be a major factor in determining system performance and reliability. A brief overview of these issues is presented in Chapter 6. One of the potential problems in using duplication and comparison for error detection is that if the two functional modules fail simultaneously in exactly the same way, the erroneous output is not detected. Chapter 6 also includes a discussion of implementation techniques that can help reduce the probability of such undetected errors.

The major results of this thesis are summarized in Chapter 7. A few conclusions, that may serve as guidelines for future research on the implementation of fault tolerance in multicomputers, are presented.

References

Adam82. G. B. Adams and H. J. Siegel, "The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems," *IEEE Transactions on Computers* **C-31**(5) pp. 443-454 (May 1982).

Aviz78. A. Avizienis, "Fault-Tolerance: The Survival Áttribute of Digital Systems," *Proceedings IEEE* **66**(10) pp. 1109-1125 (October 1978).

Barr83. I. Barron, P. Cavill, D. May, and P. Wilson, "Transputer Does 5 or More MIPS Even When Not Used in Parallel," *Electronics*, pp. 109-115 (November 1983).

Brod80. M. Brodsky, "Hardening RAMs Against Soft Errors," *Electronics* **53**(10) pp. 117-122 (April 24, 1980).

Budz82. R. L. Budziniski, J. Linn, and S. M. Thatte, "A Restructurable Integrated Circuit for Implementing Programmable Digital Systems," *Computer* **15**(3) pp. 43-54 (March 1982).

Cray74. S. Cray, "Design of Large Computers," Oral Presentation, Stanford University, Stanford, CA (December 5, 1974).

Desp78. A. M. Despain and D. A. Patterson, "X-TREE: A Tree Structured Multi-Processor Computer Architecture," *Proceedings of the Fifth Annual Symposium on Computer Architecture*, pp. 144-150 (April 1978).

DuBo82. M. DuBois and F. A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers* **C-31**(11) pp. 1083-1099 (November 1982).

Fern84. S. Fernbach, "Applications of Supercomputers in the U.S. - Today and Tomorrow," pp. 421-428 in *Supercomputers: Design and Applications*, ed. K. Hwang, IEEE Computer Society (1984).

Frie71. A. D. Friedman and P. R. Memon, *Fault Detection in Digital Circuits*, Prentice Hall (1971).

Fuji82. R. M. Fujimoto and C. H. Séquin, "The Impact of VLSI on Communications in Closely Coupled Multiprocessor Networks," *Proceedings of COMPSAC 82*, Chicago, IL, pp. 231-238 (November 1982).

Fuss84. D. Fuss and C. G. Tull, "Centralized Supercomputer Support for Magnetic Fusion Energy Research," *Proceedings of the IEEE* **72**(1) pp. 32-41 (January 1984).

Gali80. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers* **C-29**(6) pp. 527-531 (June 1980).

Gott83. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers* **C-32**(2) pp. 175-189 (February 1983).

Hill81. W. D. Hillis, "The Connection Machine (Computer Architecture For a New Wave)," AI Memo 646, Massachusetts Institute of Technology, Cambridge, MA (September 1981).

Hopk78. A. L. Hopkins, T. B. Smith, and J. H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings IEEE* **66**(10) pp. 1221-1239 (October 1978).

Hwan84. K. Hwang, *Supercomputers: Design and Applications*, IEEE Computer Society (1984).

Jone83. S. E. Jones, "The Synapse Approach to High System and Database Availability," *Database Engineering* **6**(2) pp. 29-34 (June 1983).

Katz82. J. A. Katzman, "The Tandem 16: A Fault-Tolerant Computing System," pp. 470-480 in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill (1982).

Kell84. R. M. Keller, F. C. H. Lin, and J. Tanaka, "Rediflow Multiprocessing," *COMPCON* , San Francisco, CA, pp. 410-417 (February 1984).

Koda80. K. L. Kodandapani and D. K. Pradhan, "Undetectability of Bridging Faults and Validity of Stuck-At Fault Test Sets," *IEEE Transactions on Computers* **C-29**(1) pp. 55-59 (January 1980).

Koha78. Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill (1978).

Kozd80. E. W. Kozdrowicki and D. J. Theis, "Second Generation of Vector Supercomputers," *Computer* **13**(11) pp. 71-83 (November 1980).

Kuck78. D. J. Kuck, *The Structure of Computers and Computations*, Wiley (1978).

Kuck84. D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Retargetable Vectorizer," pp. 163-178 in *Supercomputers: Design and Applications*, ed. K. Hwang, IEEE Computer Society (1984).

Lars84. J. L. Larson, "Multitasking on the CRAY X-MP-2 Multiprocessor," *Computer* **17**(7) pp. 62-69 (July 1984).

Linc82. N. R. Lincoln, "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer," *IEEE Transactions on Computers* **C-31**(5) pp. 349-362 (May 1982).

Miur84. K. Miura and K. Uchida, "FACOM Vector Processor System: VP-100/VP-200," pp. 59-73 in *Supercomputers: Design and Applications*, ed. K. Hwang, IEEE Computer Society (1984).

Nico84. A. Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers* **C-33**(11) pp. 968-976 (November 1984).

Peat81. G. Peattie, "Quality Control for ICs," *IEEE Spectrum* **18**(10) pp. 93-97 (October 1981).

Plat80. D. G. Platteter, "Transparent Protection of Untestable LSI Microprocessors," *10th Fault-Tolerant Computing Symposium*, Kyoto, Japan, pp. 345-347 (October 1980).

Rand78. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

Rasm82. R. A. Rasmussen, "Automated Testing of LSI," *Computer* **15**(3) pp. 69-78 (March 1982).

Redd83. S. M. Reddy, M. K. Reddy, and J. G. Kuhl, "On Testable Design for CMOS Circuits," *International Test Conference*, Philadelphia, PA, pp. 435-445 (October 1983).

Renn78. D. A. Rennels, "Architectures for Fault-Tolerant Spacecraft Computers," *Proceedings IEEE* **66**(10) pp. 1255-1268 (October 1978).

Russ78. R. M. Russell, "The CRAY-1 Computer System," *Communications of the ACM* **21**(1) pp. 63-72 (January 1978).

Seit85. C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1) pp. 22-33 (January 1985).

Séqu83. C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," pp. 299-326 in *VLSI Architecture*, ed. B. Randell and P.C. Treleaven, Prentice Hall, Englewood Cliffs, NJ (1983).

Shaw84. D. E. Shaw, "SIMD and MIMD Variants of the NON-VON Supercomputer," *COMPCON* , San Francisco, CA, pp. 360-363 (February 1984).

Tami83. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).

Toy78. W. N. Toy, "Fault-Tolerant Design of Local ESS Processors," *Proceedings IEEE* **66**(10) pp. 1126-1145 (October 1978).

Tryo62. J. G. Tryon, "Quadded Logic," pp. 205-228 in *Redundancy Techniques for Computing Systems*, ed. R. H. Wilcox and W. C. Mann, Spartan Books (1962).

Wads78. R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *The Bell System Technical Journal* **57**(5) pp. 1449-1474 (May-June 1978).

Wens78. J. H. Wensley, L. Lamport, J. Golberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings IEEE* **66**(10) pp. 1240-1255 (October 1978).

Widd80. L. C. Widdoes and S. Correl, "The S-1 Project: Developing High-Performance Digital Computers," *COMPCON* , San Francisco, CA, pp. 282-291 (February 1980).

Chapter Two
Basic Concepts and Terminology

This chapter discusses the concept and the need for fault tolerance in VLSI multicomputers—the system of interest in this thesis. The terminology to be used throughout this thesis is introduced in Section 2.1. The nature of hardware faults in systems implemented using MOS VLSI chips is discussed in Section 2.2. It is shown that acceptable system reliability cannot be achieved unless the multicomputer system can "tolerate" hardware faults, i.e., continue correct operation despite the failure of one of its components. A detailed discussion of the behavior required from a fault-tolerant multicomputer after one of its components has failed is presented in Section 2.3.

2.1. Terminology

Technical discussions require the use of terminology whose meaning is agreed upon by all. Unfortunately, there is no "agreed upon" terminology for computer systems in general and "fault-tolerant" computer systems in particular. The terminology used in this thesis is derived mainly from the proposals of Anderson and Lee [Ande81, Ande82] (also in Randell *et al* [Rand78]). A few of the terms specific to the study of fault-tolerant systems are from proposals by Avizienis [Aviz78, Aviz82]. Most of the terms related to the interconnection topology of the multicomputer are discussed by Tanenbaum [Tane81].

The main characteristic of the proposed terminology is that it is fundamentally hierarchical and thus corresponds to the hierarchical structure of computer systems in general and the multi-level hierarchical implementation of fault-tolerance schemes in particular. Many alternative system models and terminology schemes have been proposed in the literature. One of the most widely used proposals is the "four-universe information system model" which is based on four fixed views of the system: physical, logical, informational, and external [Aviz82]. These views correspond to the "universe of discourse" of the engineer, the logic designer, the programmer, and the user, respectively. In this model there is no representation of the hierarchical nature of the *structure* of systems. One of the difficulties in using this alternative terminology is evident when considering systems composed of fault-tolerant subsystems.

In the following three subsections the terminology used throughout the thesis is

summarized. The emphasis is on a clear semantic understanding of the terms rather than on abstract mathematical definitions.

2.1.1. Systems and their Components

A *system* is any identifiable mechanism that maintains a pattern of behavior at an interface between the mechanism and its environment.

An *interface* is the place of interaction between two systems.

The *environment* of a system is another system that provides inputs to and receives outputs from the first system.

The external behavior of the system can be described by a set of *external states* (output values), and a function defining the transitions between these states.

A system consists of a set of *components* which interact under the control of a *design*. Each component is itself a system. The design of the system is the way in which the components are interconnected. The *internal state* of a system is an ordered set of the external states of its components.

If the internal structure of a relatively simple system is of no interest and is to be ignored, the system is said to be *atomic*. For example, if the system under consideration is a circuit board populated by resistors, capacitors, and discrete transistors, each one of these components would typically be considered atomic.

2.1.2. Multicomputer Terminology

The term *multicomputer*, as used in this thesis, was defined in Chapter 1 to mean a collection of *computing elements* interconnected by high-speed point-to-point dedicated links. A computing element is a complete Von Neumann computer containing local memory as well as a processor. Others have called this type of system a *network computer*. The computing elements are also called *nodes*. A *link* is a bidirectional connection between two nodes called *neighbors*.

A multicomputer system must be connected to various peripheral devices, such as disk drives, tape drives, printers, and terminals. Furthermore, there may also be connections to other systems via a local-area-network. Each of these devices is connected to one or more of the nodes of the multicomputer. When discussing a multicomputer system, the nodes and links are considered the *components* as discussed in Section 2.1.1.

Devices that are usually considered input/output devices, such as disk drives, but which exchange information only with nodes of the multicomputer, must also be viewed as components of the system. On the other hand, a tape drive that is used to read a tape created on some other system must be considered part of the *environment*.

Nodes in a multicomputer exchange information via *messages*. A message is simply a sequence of bits containing all the information transferred as well as any overhead necessary to ensure that the information reaches the desired destination. Messages may be very long (millions of bytes) and may have to pass through several intermediate nodes if the source and destination nodes of the message are not neighbors. A *packet* is the smallest unit of information transferred between nodes. In order to transmit a large message, the sender breaks it up into packets and it is the responsibility of the receiver to assemble the original message from the sequence of packets it receives.

The *interconnection topology* of the multicomputer is the specification of which nodes are neighbors. A *path* between two nodes, i and j, is a sequence of adjacent (neighbor) nodes starting with i and ending with j. The length of the path is the number of links in the path (*i.e.* one less than the number of nodes). A *geodesic* is a path of minimum length between a given pair of nodes. Between a given pair of nodes there may be several different paths. These paths are said to be *disjoint* if they have only nodes i and j in common. The system is said to be *connected* if there is at least one path between every pair of nodes.

Two important parameters characterize the interconnection topology: *diameter* and *connectivity*. The diameter of the system is the length of the longest geodesic. The node (link) connectivity is the minimum number of nodes (links) that must be removed in order to partition the system so that it is no longer connected.

In this thesis, the interconnection topology, the protocols used for inter-node communication, the operating system, and the application software, are all considered part of the *design* of the system.

2.1.3. Terminology for Fault Tolerance

As discussed in Section 2.1.1, a system is defined to the "outside world" by its specified behavior at its interface with the environment. System *failure* occurs when its behavior deviates from the specifications.

An *erroneous* internal state is a state that could lead to a failure by a sequence of valid transitions. An *error* is a part of an erroneous state that constitutes a difference from a valid state.

An erroneous internal state is a result of either design failure or component failure. A component failure is a result of an *error* in the internal state of the component. A design failure means that the choice of components or the way in which the components are interconnected is incorrect. As a result, although each one of the components is operating according to its specifications, the set of external states of all of the system's components (at a particular point in time) is erroneous and may lead to system failure.

An error in a component is a (component) *fault* in the system. Such a fault *may* lead to a component failure which is a system error. The system error *may* lead to system failure.

Permanent faults are faults that are present for a long period of time (longer than some threshold). *Transient faults* are present for a limited period of time (less than the threshold) and then disappear from the system. An *intermittent fault* is a recurring transient fault.

As discussed above, system failure may occur, as a result of design faults, even if all the components are operating according to their specifications. Component failures may, in turn, be a result of faulty design of the components. On the other hand, it is often the case that the design of the entire hierarchy of components above the atomic components is correct, and the failure of the system is the result of the failure of atomic components. Under these circumstances it is said that the system failure is the result of a *hardware fault* or a *physical defect*. Since the design of the system and all of its components is fixed when the system is constructed, any system failure that is a consequence of physical changes in the system or its environment is classified as the result of a hardware fault.

There are two basic approaches to the construction of highly reliable systems: *fault prevention* and *fault tolerance*. With the first approach an attempt is made to ensure that the design of the system is correct, and all of its components are functional when installed in the system and are highly reliable so that they will not fail in the future. Fault prevention is accomplished using a combination of *fault avoidance* and *fault removal* techniques. *Fault avoidance* techniques, such as specialized design methodologies (such as the use of extra large tolerances in critical components) and strict quality control,

help avoid introducing faults into the system. *Fault removal* techniques, such as testing and validation, are used to find and remove faults that were introduced into the system during its construction.

Fault tolerance techniques attempt to prevent component failures (which are caused by faults) from causing system failure. The process of "tolerating" a fault involves four phases:

(1) *Error detection*: The existence of a fault can be detected by the system only after the fault generates an error somewhere in the system. Detection of an erroneous state is the starting point for any fault tolerance technique.

(2) *Damage assessment*: Between the time a fault occurs and the resulting error is detected, invalid information may spread throughout the system and lead to additional errors. Before an attempt to "recover" from the error is made, the extent to which the system state has been damaged must be determined.

(3) *Error recovery*: The erroneous system state detected and assessed in phase (1) and (2) is transformed into an error-free state, from which normal system operation can continue.

(4) *Fault treatment and continued service*: If the fault that had caused the error is permanent, steps must be taken to repair the fault or "reconfigure" the system to avoid the fault. This is necessary to prevent the same fault from generating a new error in the system.

The above four phases are not always distinct and their identification in a particular system may not be clear. Nevertheless, these phases are, at least conceptually, part of all fault tolerance techniques. The effectiveness of any fault tolerance technique depends on how well these phases are implemented explicitly or how valid are the implicit assumptions made regarding these phases.

In a system where fault tolerance technique are employed, some faults are "tolerated" (*i.e.* they cannot lead to system failure) while other faults are still potentially fatal. Hence there is an ambiguity in the use of the terms *erroneous state*, *error*, and *fault*. In order to resolve this ambiguity, these terms are defined as *recoverable* or *fatal*: A *recoverable erroneous internal state* is a state that could lead to system failure by a sequence of valid transitions in the absence of actions for fault tolerance. This term is only meaningful in a system where fault tolerance techniques are employed. In such a system, there is no sequence of valid transitions that begins with a *recoverable erroneous*

internal state and leads to system failure. A *fatal erroneous internal state* is a state that could lead to system failure by a sequence of valid transitions despite any actions for fault tolerance in the system. A *recoverable error* is a part of a recoverable erroneous state that constitutes a difference from a valid state. A *fatal error* is a part of a fatal erroneous state that constitutes a difference from a valid state. A *recoverable fault* is an error in a component that **may** lead to a component failure resulting in a recoverable erroneous state. A *recoverable fault* **cannot** lead to a component failure that will result in a *fatal erroneous state*. A *fatal fault* is an error in a component that **may** lead to a component failure resulting in a fatal erroneous state. Note that both recoverable and fatal faults are fatal with respect to the component in which they occur since they may lead to component failure.

A "fault-tolerant" system only "tolerates" recoverable faults. Whether or not a fault is recoverable depends on the system. It is possible to have two systems with the same specifications and the same components, yet a specific fault in one of the components may be recoverable in one system and fatal in the other. Furthermore, a given fault in a particular component of a system may be recoverable at some point in time but fatal at some later time, if, in the interim, some characteristic of the system has changed. This change may be due to reconfiguration done in order to "tolerate" some previous (recoverable) fault.

In order to be able to use the terms "fault-tolerant system" or "fault tolerance," we must establish the following convention: When discussing a system that has special provisions for "fault tolerance," the terms "erroneous state," "error," and "fault," when they are not further qualified, mean "recoverable erroneous state," "recoverable error," and "recoverable fault," respectively.

2.2. The Nature and Consequences of Hardware Faults

As discussed above, a multicomputer system may fail due to faulty design or faulty components. For the rest of this thesis it is assumed that the design of the system and all of its components is correct. Consequently, any system failure must be a result of component failures caused by hardware faults, *i.e.*, caused by physical changes in the system or its environment, that prevent the components from operating according to their specifications.

The question arises whether or not the frequency of component failures due to hardware faults is sufficiently high to warrant special provisions for fault tolerance. In order to answer this question and choose appropriate fault tolerance techniques, it is necessary to understand the nature of physical defects in the hardware components and the effects of these defects on the behavior of the components at their interfaces with their environments. This section discusses the causes and characteristics of physical defects in hardware components implemented with the technology assumed in this thesis—MOS VLSI. A simplified model of the behavior of MOS digital circuits under faults is presented. Since faults that are not detected during fabrication can cause component failure when the component is part of a working system, the problem of testing VLSI components during fabrication is discussed. Finally, it is shown that in a large VLSI multicomputer hardware faults cannot be ignored and there must be special provision in the system that allow it to continue correct operation despite such faults.

2.2.1. Hardware Faults in MOS Digital Circuits

VLSI chip failure may be due entirely to design or fabrication flaws, due entirely to environmental factors, or is the end result of a degenerative process due to operational and environmental stresses but partially attributable to design or manufacturing defects [Doyl81, Howa82]. Fabrication defects in MOS chips consist mainly of shorts and opens in each interconnection level (metallization, diffusion, and poly-silicon), shorts through the insulator separating different levels, and large imperfections such as scratches across the chip [Gali80]. Other fabrication defects include incorrect dosage of ion implants, contact windows that fail to open, misplaced or defective bonds, and penetration of the package by humidity and other contaminants [Doyl81]. During the operation of the chip, faults may be caused by electromigration, corrosion, electrical breakdown of oxide, cracks due to different thermal expansion coefficients, power supply fluctuation, and ionizing or electromagnetic radiation [Doyl81, Cast82].

The probabilities (rates) of the different types of chip failures are difficult to determine experimentally and the manufacturers are reluctant to release any available information. In 1981 it was reported that, due to manufacturing defects and incomplete testing, the number of defective VLSI chips reaching customers is between 100 to 1000 parts per million [Peat81]. The rate of permanent hardware faults, which are a result of aging and deterioration during the operation of the chip (*e.g.* corrosion or oxidation,

insulation breakdown or leakage, ionic migration of metals, shrinking or cracking of plastic material), has been reported to be on the order of 300 per billion part hours [Peat81, Budz82]. Measurements of complete systems indicate that the the rate of transient faults, which are related to environmental factors (*e.g.* electromagnetic radiation received by interconnections, power supply fluctuations, ionizing radiation), is at least an order of magnitude greater than the rate of permanent faults [Cast82].

The failure rates above reflect the reliability of relatively mature chips that have been in production for several years. New chips exhibit much higher failure rates. For example, the Texas Instruments TMS-1000 microcomputer, which had a failure rate of 300 per billion part hours in 1979, had a failure rate of 5000 per billion part hours when it was introduced in 1974-1975 [Budz82]. Similarly, when a new complex VLSI chip is first introduced, it is common for more than one percent of the chips sent to customers to be defective.†

It should be noted that the above failure rates are for a single VLSI chip, such as a microprocessor, and do not take into account the other sources of failure in a complete system, which include printed circuit boards, cables, interconnections between the chips and the boards, etc. For example, measurements on a "real" system, the Cm*, have shown that the rate of permanent failures of a "Computer Module," consisting of an LSI-11 microprocessor, memory, and a "switch" (approximately 400, mostly SSI/MSI, chips), is on the order of 200 per million module hours [Siew78]. Since the failure rate of the standard SSI and MSI chips is at least an order of magnitude smaller than the failure rates mentioned above for VLSI chips, the rate of module failures cannot be explained by chip failures alone.

Although design faults are not considered in this thesis, faults caused by *marginal* design are often indistinguishable from faults caused by environmental factors or marginal fabrication and must therefore be taken into account even if the "basic" design is assume to be correct. For example, if the specified width of a metal line is too small, the result of an open line due to electromigration is identical to the result of a fabrication defect that causes a properly specified metal line to be too narrow. Due to the steadily increasing complexity of chips and the fact that they must deal with interacting asynchronous events, it is becoming more difficult to ensure that the design is correct so that the chip

† No formal reference for this information is known to the author.

can properly deal with all possible combinations of events. Incorrect behavior due to a rare, unexpected combination of events may be indistinguishable from incorrect behavior due to a random burst of cosmic rays modifying a value in memory.

Marginal design problems are often detected only after the chip has been in use. The detection and correction of marginal design faults are one important explanation of the previously discussed increased reliability of later releases of chips compared to their reliability when they are first introduced.

2.2.2. A Fault Model for MOS Digital Circuits

At the lowest level, any digital circuit is a non-linear electrical circuit with analog values of voltages and currents. Determining the precise effects of the physical defects discussed in the previous subsection on the operation of the circuit requires complex analysis which involves solutions of non-linear differential equations. Given the size of VLSI chips and the variety and complexity of the physical defects that can occur, it is very difficult (practically impossible) to perform such analysis for each possible defect. A higher level simplified model (*fault model*) of how the circuit is affected by physical defects must therefore be used.

Even low-level design of digital circuits is often done without direct consideration of the detailed electrical characteristics of the components. Instead, the designer works at the level of Boolean logic ("ones and zeros"). The electrical characteristics are taken into account by following simple "design rules" that are expressed at the level of ones and zeros, logic gates, etc., rather than at the level of voltages and currents. While these design rules prevent the use of some circuits that could provide an effective implementation of desired functionality, the simplification of the design process and the analysis of that design is, in most cases, worth the potential limitations. Just as design rules make it possible to design very complex systems, high-level fault models make it possible to predict the behavior of complex faulty circuits.

The highest level fault model is the *general functional* model [Haye85]. It allows arbitrary changes in the functionality of the circuit due to physical defects. In its "purest" form this model is useless since it simply indicates that the circuit can exhibit arbitrary behavior due to defects. This model is used by applying it to simple modules that make up a more complex system. It is then assumed that only a small number of

modules (typically one) will fail simultaneously and the behavior of the system due to the arbitrary failure of one or several of these modules is determined. The result is a restricted functional fault model for the entire system [That80].

High level functional fault models are specific to a particular system and are of no help in understanding the effects of physical defects on the operation of arbitrary circuits. To gain such an understanding, it is necessary to consider the characteristics that are common to all circuits of a particular technology. A logic level fault model for MOS circuits is therefore introduced in the remainder of this subsection.

Prior to the advent of VLSI, most digital circuit design was done at the level of basic combinational logic gates (AND, OR, NOT) and simple flip-flops. A relatively low level fault model was developed to correspond to these modules. This so called stuck-at model was based on the assumption that most physical defects have the same effect on the operation of the circuit as a set of gate inputs and outputs that are stuck at logic 0 or logic 1 [Frie71]. While the stuck-at fault model can represent the effects of a significant percentage of the physical defects that occur in modern NMOS and CMOS VLSI circuits, it cannot represent the effects of several other possible defects and is therefore insufficient [Cour81, Gali80, Wads78].

The effects of most defects can be represented, at the logical level, by a circuit model that consists of a network of switches, loads (for NMOS), and interconnection lines which directly correspond to the transistors and interconnections in the actual circuit [Gali80]. Shorts and breaks in lines can be represented with this circuit model in an obvious way [Cour81]. Shorts to "ground" and "power" are the traditional stuck-at faults. A "switch" may be permanently on or permanently off, corresponding to a gate input stuck-at-1 or 0, respectively. Shorted NMOS loads (pull-ups) are equivalent to an output line s-a-1. Disconnected gate inputs are usually equivalent to s-a-0 or s-a-1 faults. A single break in a line that fans out to many inputs is equivalent to multiple stuck-at faults (all of the same type).

Some physical defects have a more complex effect on the circuit. In NMOS, incorrect dosage of ion implants may cause a threshold shift in a load transistor. This can result in an output voltage that lies between the voltages assigned to logic 0 and logic 1. If the fanout from the gate is greater than one, some of the attached gates may "interpret" its output as logic 1 while others will interpret it as logic 0. If, at some point

in time (clock cycle), the line is supposed to be a logic 1 but is interpreted by at least one of the gates as logic 0, it is called a *weak 1* fault. Conversely, if the line is supposed to be a logic 0 but is interpreted by at least one of the gates as logic 1, it is called a *weak 0* fault [Tami83]. A line may exhibit both a weak 0 fault and a weak 1 fault, as a result of a single physical defect.

A stuck-at-1 fault is a degenerate case of a weak 0 fault while a stuck-at-0 fault is a degenerate case of a weak 1 fault. If a line is stuck-at-1, *all* the devices connected to it *always* interpret its value as logic 1. If a line has a weak 0 fault, *at least one* of the devices connected to it *always* interprets it as a logic 1.

Breaks in lines are another possible source of weak 0 and weak 1 faults. A break may result in a segment of the line that is only connected to gates of MOS transistors and is therefore essentially "floating." The gates connected to the floating segment of the line receive an incorrect value for the line in one of its states (0 or 1).

A single break in the line can result in the line being stuck-at-1 if all the pull-down devices are disconnected from the rest of the line, and in the line s-a-0 if all the pull-up (or load) devices are disconnected from the rest of the line. Furthermore, if only some of the pull-up or pull-down devices are disconnected from the line, the line may not be s-a-0 or s-a-1 but assume the wrong value for some inputs that only turn on the disconnected pull-ups or pull-downs. A particularly troublesome case may arise in CMOS or dynamic logic circuits: a break in a line or a transistor that is permanently off can make the output of a supposedly combinational logic circuit dependent on the previous output rather than the current input alone. Such a fault is called a *stuck-open* fault [Wads78]. A testing procedure that is designed to detect any single fault but assumes that the circuit is strictly combinational, may fail to detect a stuck-open fault.

A short between adjacent or crossing lines that are supposed to have complementary values may affect the value of one or both of the lines, depending on the conductivity of the short and the strength of the drivers attached to the two lines. A line whose value is affected is either forced to the value of the other line or to an intermediate value between logic 0 and logic 1. In the worst case, the result of a short is that the line that is supposed to be at logic 1 has a weak 1 fault, and the line that is supposed to be at logic 0 has a weak 0 fault. The circuit may be designed so that most shorts force both lines to a well defined logic 1 or logic 0. This value may be always the value of one of the two lines that

"dominates" the other because it is driven with larger devices. Alternatively, the value may always be logic 0 (AND operation) or always logic 1 (OR operation).

Traditionally, the term *single fault* has been used to denote an erroneous logic value on a single line in the circuit. From the above it is clear that a single physical defect may result in erroneous logic values on several lines in the circuit. Hence, the term *single fault* will be used in this thesis to denote the effect, at the logical level, of a single physical defect.

If faults randomly appear and disappear in the circuit having a different effect on its operation every time, the fault model is of no use for either determining how to test the circuit or predicting its behavior when a fault occurs. Similarly, if it is assumed that an arbitrary number of faults may occur simultaneously in a complex VLSI circuit, the result is nearly identical to, and just as useless as, a general functional fault model. Several restrictive assumptions must therefore be made. These assumptions are based on the likely consequences of the physical defects under consideration.

It is assumed that, for the duration of the fault (defect), the effects of the defect are *deterministic* so that under identical conditions the effects of a particular defect are always the same. Thus, if a line has a weak 1 fault due to its driver, those devices connected to it that misinterpret the logic 1 as a logic 0, *always* misinterpret the logic 1 as a logic 0. Although a transient fault may cause a permanent change in the *state* of a circuit with memory elements, it is assumed that the circuit returns to its original physical structure after the fault has disappeared.

2.2.3. Testing and Its Limitations

The final step in the fabrication process of VLSI chips is extensive testing which attempts to ensure that no faulty chips reach the customers. Conceptually, the simplest way to test a circuit is based on the general functional fault model. The circuit can be placed in a system where it performs all its specified functions and the results are compared with the correct results. Unfortunately, such testing is not practical. For example, in order to test a microprocessor every possible instruction must be executed, with every possible addressing mode, with every possible data combination, starting with all possible internal states, and "modulated" by all possible external events (interrupts). Daniels and Bruce [Dani85] estimate that such testing of a simple 8-bit microprocessor

would take two million years!

Typically, integrated circuits are either "very good" or "very bad."[Dani85] It has been shown that less than a hundred test patterns, even if randomly generated, are sufficient to detect most faulty chips[Dani85, Will85]. The problem is how to identify the very small percentage of faulty chips that pass the initial test.

Given the impracticality of complete functional testing, the test procedure must be based on a more restrictive fault model. The single-stuck-at fault model has traditionally been used to evaluate the effectiveness of various testing procedures. It is assumed that chips that pass the initial test may have only one stuck-at fault. After a testing sequence is developed, simulation is used to determine the percentage of single stuck faults detected by that sequence. It has been shown that a few hundred to a few thousand test patterns are sufficient to achieve nearly a hundred percent coverage of single stuck faults in a microprocessor chip[Dani85, That80].

Unfortunately, *test grading* based on single stuck faults is too optimistic. It ignores the fact that many of the possible physical defects cannot be modeled by single stuck faults (see the previous subsection).

Another problem is caused by the widespread use of a facility intended to simplify testing: modern VLSI chips often allow direct control over all the latches on chips by chaining them together into a large shift-register[Eich78]. With this scan-in/scan-out facility, the problem of testing a large sequential circuit (the chip) is reduced to testing many combinational circuit blocks. A test pattern is shifted into the latches and after one clock cycle, results are shifted out. With this testing procedure the chip is not tested under normal operating conditions. Since many circuits used in VLSI chips are dynamic, it is either impossible to test them at all with this scheme or, even if testing is possible, the test does not reflect their normal operation.

Even the testing of small combinational circuits is not as simple as it might first appear. In particular, as discussed in the previous subsection, a break in a line of a CMOS logic gate can cause its output to be dependent on the previous output rather than the current input alone. This *stuck-open* fault may escape detection even if all possible input vectors are used to test the circuit[Wads78].

Using *sequences* of test patterns it is usually possible to detect the stuck-open faults discussed above[Wads78]. Unfortunately, even this is not guaranteed. In particular,

Reddy et al[Redd83] have shown that for a multi-input combinational CMOS gate it is possible for small time skews in changes of the inputs to invalidate all possible test sequences for detecting particular stuck-open faults. Thus, a chip may pass an extensive testing procedure only to fail later due to slight changes in the environment (temperature, time skews in inputs to the chip, etc.)[Redd83].

2.2.4. The Need to "Tolerate" Hardware Faults

The previous subsections discuss a wide variety of physical defects and their effects on the operation of VLSI integrated circuits. From this discussion, it is clear that chips cannot be guaranteed to always operate according to their specifications. They will always be susceptible to internal physical changes that permanently modify their functionality, as well as to environmental factors that can modify their internal state and cause them to fail without any physical changes in their structure. Furthermore, since it is impossible to completely test chips during production, it can be expected that a small percentage of the chips installed in any system are faulty.

As discussed in Chapter 1, as long as the most up-to-date technology is always used, unreliability of chips will continue to be a problem. With advances in technology, chips become more complex and thus more difficult to test. Furthermore, with each change in technology, new unexpected failure modes may become important, and the identification of these failure modes may only occur after years of experience.

Given the fact that chips do, and will, fail, the question arises whether they fail often enough to significantly affect the operation of a large multicomputer system. To answer this question, consider a system with ten-thousand VLSI chips. From the discussion in the previous subsections, the rate of permanent hardware faults for each chip can be expected to be between one hundred and five hundred per billion part hours. The rate of transient fault can be expected to be at least an order of magnitude greater. Assuming that chips fail independently and that the failure of a single chip will cause the entire system to fail, the mean time between failure (MTBF) of the system due to permanent faults can be expected to be between two hundred and one thousand hours. The MTBF due to transient fault can be expected to be between twenty and one hundred hours.

It should be noted that the above calculation of the system MTBF takes into account only chips failures. The failure of the interconnections between chips is ignored. Thus, in

reality, the MTBF of the system can be expected to be significantly lower. However, even if an optimistic MTBF of, say, fifty hours is assumed, there are severe consequences as to the ability of the system to perform its intended function. For example, in such a system, assuming that component failures are exponentially distributed, the probability that a task that executes for one hundred hours will produce the correct results is only fourteen percent!

2.3. The Requirements of a "Fault Tolerant" Multicomputer

Since a system *fails* when its behavior deviates from its specifications, whether a system is or is not fault-tolerant depends on the point of view. The "point of view" is the system's specifications. A system "tolerates" a specific fault if it continues to behave according to its specifications despite the existence of that fault. Thus a "fault-tolerant" system does not necessarily exhibit identical behavior before and after the occurrence of a fault. All that is required is that it continues to comply with its specifications. Hence, if the specifications are sufficiently lenient, almost every system can be described as "fault-tolerant." For example, if the specifications of a computing system indicate that "the system will never blow up, even in the presence of all possible faults," then all computing systems may be considered "fault-tolerant." If, on the other hand, the specifications require that "the system will continue normal operation in the presence of all possible faults," then no computing system is "fault-tolerant."

The minimal meaningful requirement is that for a large majority of faults, the system will either continue to function correctly (perhaps at lower performance) or indicate that an error has occurred. Hence the system must be *self-checking*. If this requirement is satisfied, it is highly unlikely that incorrect outputs will be considered correct. The maximal practical specifications require that the system continue normal operation (perhaps at lower performance) despite the occurrence of most faults. The system must continue to be *self checking* at all times, including during and after "recovery" from a previous fault.

Based on the previous section, the minimal requirement, that the system be self-checking, is insufficient for a large multicomputer. If the system is self-checking but is unable to continue any tasks that it was executing when a fault occurs, such tasks must be restarted from scratch after the system is repaired. Considering, once again, a system with an MTBF of fifty hours and a task that requires one hundred hours to execute, the

expected execution time for this task, due to the need to restart it after each fault, is more than seven hundred hours. Thus, the multicomputer must be able to recover from most faults and continue the correct execution of tasks in the system.

References

Ande81. T. Anderson and P. A. Lee, *Fault Tolerance Principles and Practice*, Prentice-Hall (1981).

Ande82. T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 29-33 (June 1982).

Aviz78. A. Avizienis, "Fault-Tolerance: The Survival Attribute of Digital Systems," *Proceedings IEEE* **66**(10) pp. 1109-1125 (October 1978).

Aviz82. A. Avizienis, "The Four-Universe Information System Model for the Study of Fault-Tolerance," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 6-13 (June 1982).

Budz82. R. L. Budziniski, J. Linn, and S. M. Thatte, "A Restructurable Integrated Circuit for Implementing Programmable Digital Systems," *Computer* **15**(3) pp. 43-54 (March 1982).

Cast82. X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* **C-31**(7) pp. 658-671 (July 1982).

Cour81. B. Courtois, "Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits," pp. 341-350 in *VLSI 81*, ed. J. P. Gray, Academic Press (1981).

Dani85. R. G. Daniels and W. C. Bruce, "Built-In Self-Test Trends in Motorola Microprocessors," *IEEE Design and Test* **2**(2) pp. 64-71 (April 1985).

Doyl81. E. A. Doyle, "How Parts Fail," *IEEE Spectrum* **18**(10) pp. 36-43 (October 1981).

Eich78. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Journal of Design Automation and Fault-Tolerant Computing*

2(2) pp. 165-178 (May 1978).

Frie71. A. D. Friedman and P. R. Memon, *Fault Detection in Digital Circuits*, Prentice Hall (1971).

Gali80. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers* **C-29**(6) pp. 527-531 (June 1980).

Haye85. J. P. Hayes, "Fault Modeling," *IEEE Design and Test* **2**(2) pp. 88-95 (April 1985).

Howa82. R. T. Howard, "Packaging Reliability: How to Define and Measure It," *32nd Electronic Components Conference*, San Diego, CA, pp. 376-384 (May 1982).

Peat81. G. Peattie, "Quality Control for ICs," *IEEE Spectrum* **18**(10) pp. 93-97 (October 1981).

Rand78. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

Redd83. S. M. Reddy, M. K. Reddy, and J. G. Kuhl, "On Testable Design for CMOS Circuits," *International Test Conference*, Philadelphia, PA, pp. 435-445 (October 1983).

Siew78. D. P. Siewiorek, V. Kini, R. Joobbani, and H. Bellis, "A Case Study of C.mmp, Cm*, and C.vmp: Part II - Predicting and Calibrating Reliability of Multiprocessor Systems," *Proceedings IEEE* **66**(10) pp. 1200-1220 (October 1978).

Tami83. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).

Tane81. A. S. Tanenbaum, *Computer Networks*, Prentice Hall (1981).

That80. S. M. Thatte and J. A. Abraham, "Test Generation for Microprocessors," *IEEE Transactions on Computers* **C-29**(6) pp. 429-441 (June 1980).

Wads78. R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *The Bell System Technical Journal* **57**(5) pp. 1449-1474 (May-June 1978).

Will85. T. W. Williams, "Test Length in a Self-Testing Environment," *IEEE Design and Test* **2**(2) pp. 59-63 (April 1985).

Chapter Three
Error Detection in Multicomputers

When one of the components of a system fails, it causes an *error* in the system, i.e., it results in an erroneous internal system state that can lead to system failure unless some special action is taken by the system to recover or to reconstruct a valid internal state. In order to prevent system failure, such errors must be detected soon after they occur. Thus, the system must include a mechanism for detecting the failure of its components.

The only way to guarantee error detection is to compare the outputs from all components with apriori known correct results using a comparison mechanism the can never fail. Since the correct outputs are not known ahead of time and there is no failure-proof comparison mechanism, this "scheme" is not useful for implementing error detection in a multicomputer. It is clear that no error detection is perfect; for every scheme there is a subset of the possible errors that cannot be detected. Different error detection schemes must therefore be evaluated with respect to the percentage of errors that can be detected as well as the required system overhead in hardware, design complexity, and extra operations that must be performed during normal operation. In addition, an important consideration is how difficult it is to *locate* the faulty component once an error is detected.

This chapter discusses various techniques for error detection and location in multicomputers. An overview of system level techniques is presented in Section 3.1. With these techniques the key to the detection of errors and the identification of faulty nodes is the information exchanged between the nodes. An alternative approach to error detection and location relies on the node to test itself and to notify the rest of the system when an error occurs. This approach is discussed in Section 3.2.

3.1. System-Level Error Detection Techniques

As mentioned above, the best way to detect erroneous outputs from a component is to compare those outputs with known correct outputs. Since the correct outputs are not known ahead of time, this technique can only be approximated. One way to accomplish this approximation is for each component in the system to subject all the information it receives from other components to *acceptance tests* [Rand78]. It is assumed that the

information is either correct, or it will fail to satisfy some simple criterion. For example, if it is known that the correct results must be positive, negative results indicate an error.

Error detection based on acceptance tests is well-suited for a multicomputer where the components (nodes) are "intelligent" enough so that they can be programmed to perform the necessary acceptance tests. The problem is that the acceptance tests are dependent on the application. For some applications it may be easy to come up with simple low-cost acceptance tests that will detect most errors. For other applications, the only acceptance tests that yield a sufficiently high probability of error detection require as much computation as the original task that produced the results. In either case, the application programmer is burdened with the task of developing and evaluating the acceptance tests. While this may be reasonable for some special purpose systems, it is not acceptable for a multicomputer intended for a variety of general purpose applications.

Instead of evaluating the results of each task using acceptance tests, error detection can be achieved by performing the task simultaneously on two components (subsystems) and comparing the results. As long as both components do not fail in exactly the same way, errors are identified by results that are not identical. This scheme can be implemented in a multicomputer so that when a process is initiated a duplicate process is initiated on a different node. Messages intended for the process are sent to both nodes and both copies of the process send messages to other nodes. A node receiving "a message" actually receives two copies of the message that are supposed to be identical. Error detection is accomplished by the comparison of the two copies.

The above system-controlled node-level duplication and comparison scheme has the advantage that error detection is implemented entirely in software and there is no need for any special hardware. Furthermore, the scheme is completely independent of the application. On the other hand, task assignment and message routing are more complicated. Furthermore, as discussed below, the restrictions on the operation of the system posed by this scheme result in lower performance.

The need to assign each process to two nodes increases the overhead of initiating new processes. Since a node can fail while simply forwarding a message, the system must ensure that the results from the two copies of each process are sent to their destinations via different paths. Thus, message routing can no longer always be done in a distributed dynamic manner. Specifically, the source node of a message has to compute the entire

path for that message rather than simply forward it to a neighbor and let that neighbor decide on the next step in the path towards the final destination. Thus, dynamic local load balancing of the link traffic and of the forwarding overhead of nodes is precluded. The two copies of an application process must coordinate their choices of message paths to ensure that they are disjoint. Since all the destination nodes for messages from a process may not be known when the process is initiated, this coordination must be done during normal operation whenever communication with a new node is initiated. The pre-computed communication paths also make error recovery more complicated: messages "in transit" that have pre-computed paths through a failed node cannot simply be re-routed locally; they must be removed and the source node has to send a new copy of such messages with pre-computed paths that are disjoint from the paths used by the duplicate application process on some other node.

The two copies of a process cannot be allowed to both execute independently without any coordination. For example, without coordination, if the process is accessing a disk file, duplicate access requests will be received by the disk controller and result in duplicate "writes" and in sequential "reads" that provide different data to the two copies of the process. Furthermore, accesses to any centralized resource must be checked to make sure that they were not initiated by a process on a faulty node. Thus, all centralized resources must be controlled by "reliable" nodes that wait for two copies of each request and compare them in order to detect errors. The implementation of these "reliable" nodes requires the use of other fault tolerance techniques and the coordination of each access is bound to result in lower performance.

Since each process can spawn new processes, and two copies of each new process must be initiated, without coordination there will be an exponential growth in the number of processes in the system. Furthermore, if one node is allowed to spawn new processes, both copies of the new process may begin execution in an erroneous internal state that will not be detected and may cause system failure. Coordination of process initiation requires overhead in time and in the storage for the extra "bookkeeping" information.

To minimize communication delays, the system must attempt to assign processes that communicate often to nodes that are close to each other. With the above scheme, the number of processes that interact "often" in a particular application is doubled. Since each node has only a small number of neighbors (equal to the number of its

communication ports), the average distance between nodes executing these processes is increased, resulting in increased communication delays in the system.

Even if the above problems with system-controlled duplication and comparison are overcome, the scheme does not provide adequate support for locating faulty components so that after an error is detected they can be logically removed from the system before normal operation is resumed. When a node receives two unequal copies of a message it cannot determine which one of the originating nodes failed or whether the error was caused by a faulty communication link or by one of the nodes that forwarded the message.

The need to locate faulty nodes and links, rather than simply determine that they exist, favors more localized schemes for error detection. If the failure of a node can be detected by an immediate neighbor that is fault-free, the entire system can determine which node has failed if the diagnostic information can be reliably distributed from the neighbor.

One way for nodes to determine if their neighbors (or adjacent links) are faulty is to periodically execute "diagnostics" on these neighbors. Preparata et al [Prep67] performed pioneering work on the correct interpretation of such diagnostic information under the assumption that a test performed by a faulty node is invalid. Using this work, Hakimi and Amin [Haki74] have shown that if all the nodes test their neighbors simultaneously, all faulty nodes can be identified based on the results of these tests if the following conditions hold: (1) the number of faulty nodes is less than half the total number of nodes, (2) the node connectivity of the system is greater than the number of faulty nodes.†

The results of the work mentioned above are not directly applicable to the diagnosis of a multicomputer. In a multicomputer the nodes operate asynchronously and cannot perform the tests "simultaneously." Furthermore, there is no "central observer" that can reliably obtain and interpret the results of all the local tests. The need for a central observer is eliminated in a *distributed* diagnosis algorithm developed by Kuhl and Reddy [Kuhl80]. In a system using this algorithm each node tests its neighbors directly and then forwards to all of them the results of these tests. A node accepts and propagates diagnostic information for a neighbor only after testing that neighbor to ensure that it is not faulty. Based on the tests it performs and the diagnostic information it receives, each node can *independently* diagnose the system as long as the node connectivity of the

† This is a simplification of the actual results.

system is greater than the number of faulty nodes.†

There are several difficulties with the use of distributed diagnosis for error detection as part of a fault tolerance scheme:

(1) The diagnosis relies on the ability of nodes that are interconnected via asynchronous communication links to test each other. Furthermore, since these tests are performed during normal operation, they must be relatively short. Given the difficulties of testing VLSI chips (see Chapter 2), the quality (coverage) of such tests is doubtful.

(2) Most faults are transient rather than permanent [Cast82]. A node may produce incorrect outputs and later pass an exhaustive test.

(3) A node can only "trust" a message it receives after every node on the path of the message has tested the link through which the message arrived as well as the node that forwarded it. This requirement leads to more complicated communication protocols and restrictions on the allowable behavior of applications [Hoss83].

3.2. Error Detection Using Self-Checking Nodes

The problems with error detection schemes based on periodic testing stem from the distance between error occurrence and detection in both space and time. As a result of the distance in space, once an error is detected, it is difficult to determine which component was originally responsible for the error. As a result of the distance in time, erroneous information is able to spread throughout the system before an error is detected.

The key to developing an effective error detection scheme is thus to minimize the distance between error occurrence and detection in both space and time. Ideally, these distances can be reduced to zero so that as soon as an error occurs, i.e., a component produces incorrect results, the error is detected by all the other system components that are receiving this erroneous information. This "ideal" can be achieved if all the components in the system are *self-checking* so that in addition to their normal outputs they also indicate to the rest of the system whether these outputs are correct.

All possible outputs from a self-checking component are divided into two disjoint sets. Outputs that contain an error indication are called *noncode* outputs, and outputs that do not contain an error indication are called *code* outputs. The self-checking

† This is a simplification of the actual results.

mechanism of the component is said to have failed if the component produces *code* output that is incorrect. Another component tests the validity of the outputs of a self-checking component by determining whether these outputs are code or noncode.

Just as complete testing of a VLSI chips is impossible (see Chapter 2), no component can be self-checking with respect to all possible combinations of hardware faults. Instead, for all *likely* faults, the component must either produce the correct outputs or produce noncode outputs. A component that satisfies this requirement is said to be *fault secure* [Ande73].

When a fault occurs, it does not necessarily cause immediate component failure; the outputs from the component may continue to be indistinguishable from those of a fault-free component for a long time. The fact that a fault occurs only means that there is a possibility that, for a particular input or state, the outputs will deviate from those of a fault-free component some time in the future, and that the deviation will be directly attributable to the occurrence of the fault.

No component is fault-secure with respect to all possible combinations of all possible faults. Since the component is not guaranteed to produce a noncode output immediately following the occurrence of the first fault, several different faults may exist in a component without any indication to the rest of the system. If the component is not fault secure with respect to the particular combination of faults, future incorrect outputs from the component may be accepted as correct by the rest of the system. In order to prevent this situation, there should be a high probability that after a small number of faults occur, a noncode output is produced by the component before the fault-secure property of the component is destroyed by additional faults.

A component is *self-testing* [Ande73] if it is guaranteed to produce a noncode output, due to the occurrence of one or more faults, before additional faults can occur and lead to the failure of the self-checking mechanism (in which case the component may no longer be fault-secure). Thus, if one or more faults occur during normal operation of the component, a noncode output will be produced within some bounded period of time. In order to achieve the goal of providing reliable error detection, self-checking components must be both fault-secure and self-testing. Such components are said to be *totally self-checking* [Ande73] (*TSC*).

One of the difficulties in implementing self-checking nodes in a multicomputer is that

such nodes must be capable of sending an error indication to the rest of the system while failing to correctly perform their normal function. This implies that the transmission of the error indication must not rely on the correct functionality of most of the node. It is therefore unacceptable for the error indication related to a particular packet to be sent separately from that packet. Not only does this introduce a distance in time between error occurrence and detection, but also, the process of recording the error indication and later transmitting it is relatively complex and unreliable in a node that is already failing. The indication of whether a particular output is correct must therefore be transmitted together with that output.

In one approach to implementing TSC nodes, a variety of techniques are used for different parts of the node. For example, busses, memories and registers may include one or more "parity bits" carrying redundant information that can be used to detect errors [Tsao82]. Complex residue codes and parity prediction schemes can be used for checking ALUs [Aviz71, Kraf81]. Error detecting codes are not useful for self-checking shifters, modules that perform logic functions, and control logic; self-checking modules of this type can only be implemented by duplication of the functional modules and comparison of the results [Tsao82].

If the TSC node is implemented using a variety of self-checking techniques for different parts of the node, error signals from all the self-checking subcomponents must somehow be combined to generate the error signal to the rest of the system. Unfortunately it is not clear how such signals can be combined in a "reliable" manner. The need to consider error detection in every part of the module increases the complexity of the design and its verification, thereby decreasing the confidence that the design is correct and reducing the overall reliability of the system. Furthermore, the effectiveness of a combination of different localized self-checking techniques inside a chip is very difficult to evaluate.

An alternative approach to implementing a TSC node uses two identical, *independent* modules, each performing the function of the node. If the two modules operate synchronously, their outputs should always be identical. As long as the two modules do not fail simultaneously in exactly the same way, producing identical *incorrect* outputs, an error can be detected by a simple comparison.

If the outputs from the two modules are transmitted through independent links to

the neighbors, then the comparison can be done by the neighbors. As long as the two parts of the duplex link never fail in exactly the same way at the same time, the comparison done by the neighbor checks the link as well as the source node. There are two disadvantages to this approach: (1) it doubles the required communication bandwidth, and (2) the error signal is not available within the faulty node for possible local action in response to a mismatch (see below).

Instead of transmitting the outputs from both modules, the comparison can be done inside the node (Fig. 3.1). The outputs from the comparator can be used as an error indication to the rest of the system as well as for local action. The output from one of the modules is the "functional" output from the node.

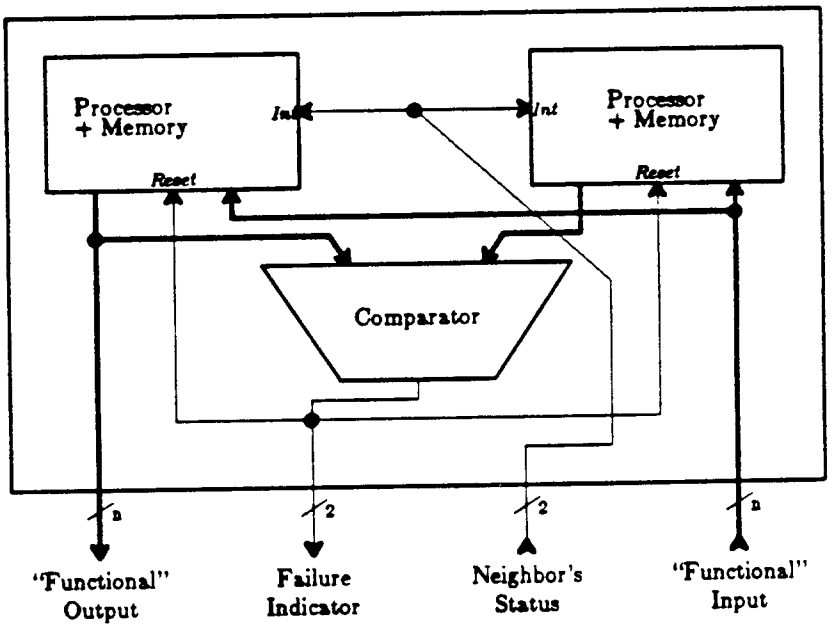


Fig. 3.1: A Self-Checking Self-Resetting Node

An important property of the self-checking node in Fig. 3.1 is that the output from the comparator is used to reset the node when an error occurs. This allows the node to attempt to reestablish a "sane state" so that the system can continue to use it. At the same time, the error signal is also received by neighbor nodes which can make an independent decision whether they are willing to continue to interact with this node.

Since the comparison of the outputs of the two modules is done locally within each node, some other technique is used to detect errors caused by faulty communication links. This can be done using error-detecting codes. For example, using cyclic codes, any desired probability of detecting errors can be achieved by adjusting the number of check bits sent

with each packet [Elki82]. Errors in the encoding and decoding of packets are also detected since packets are encoded *before* they are compared at the output of a node, and packets are decoded separately by the duplicate modules in each node.

A component in which the *TSC* property is achieved by duplication and comparison requires more than twice the hardware of a component that is not *TSC*. While at first glance this approach may seem wasteful, it is well worth the cost when the resulting low design complexity and high fault coverage are considered.

There still are situations in which duplication and comparison fails to detect errors: (1) the comparator may fail and mask a mismatch between the outputs of the two modules, and (2) the two modules may fail simultaneously in exactly the same way. Due to the first problem, faults in the comparator must not remain undetected, *i.e.*, the comparator must be self-testing. The probability that two complex VLSI modules will simultaneously fail in exactly the same way is very low. In Chapter 6 it is shown how this probability can be reduced further, but for now it will be assumed that this probability is zero.

Even without the self-testing requirement, all possible outputs from the comparator can be divided into two disjoint domains: code outputs that indicate that the two modules are producing identical results and noncode outputs that indicate an error. For the rest of the system it is important to determine whether the outputs of a particular node can be "trusted." It does not matter whether the node cannot be trusted due to the failure of the functional modules or due to the failure of the comparator. Thus, the self-testing comparator should be implemented in such a way that a fault in the comparator will result in an output that is in the same domain as comparator outputs that indicate a mismatch between the functional modules.

Unfortunately, it is not possible to implement a comparator that will produce a noncode output immediately when a fault occurs. Different parts of any self-testing circuits are tested by different inputs. Thus, the comparator must be driven by some subset of the possible inputs in order to perform a complete self-test. Since during normal operation the outputs from the two functional modules are always identical, only inputs to the comparator generated by identical outputs from the functional modules are considered (these inputs are *code inputs* to the comparator). In the worst case, *all* possible code inputs may be required for a complete self-test.

The realistic requirements from the self-testing comparator are therefore that it will produce noncode output a "short time" after a fault occurs. In this case "short" is relative to the failure rate of the hardware. The noncode output must be produced soon enough after the fault occurs so that there is a very low probability that one of the functional modules will fail in the interim. Thus, the minimal requirement of the comparator used in a TSC node is that for any likely single fault there is some code input that results in a noncode output. The implementation of a comparator that satisfies this requirement is discussed in Chapter 4.

References

Ande73. D. A. Anderson and G. Metze, "Design of Totally Self-Checking Check Circuits for m-Out-of-n Codes," *IEEE Transactions on Computers* **C-22**(3) pp. 263-269 (March 1973).

Aviz71. A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers* **C-20**(11) pp. 1322-1330 (November 1971).

Cast82. X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* **C-31**(7) pp. 658-671 (July 1982).

Elki82. S. A. Elkind, "Reliability and Availability Techniques," pp. 63-181 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).

Haki74. S. L. Hakimi and A. T. Amin, "Characterization of Connection Assignment of Diagnosable Systems," *IEEE Trans. Computers* **C-23**(1) pp. 86-88 (January 1974).

Hoss83. S. H. Hosseini, J. G. Kuhl, and S. M. Reddy, "An Integrated Approach to Error Recovery in Distributed Computing Systems," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 56-63 (June 1983).

Kraf81. G. D. Kraft and W. N. Toy, *Microprogrammed Control and Reliable Design of Small Computers*, Prentice-Hall (1981).

Kuhl80. J. G. Kuhl and S. M. Reddy, "Distributed Fault-Tolerance for Large

Multiprocessor Systems," *Proc. 7th Annual Symposium on Computer Architecture*, pp. 23-30 (May 1980).

Prep67. F. P. Preparata, G. Metze, and R. T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. Electronic Computers* EC-**16**(6) pp. 848-854 (December 1967).

Rand78. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

Tsao82. M. M. Tsao, A. W. Wilson, R. C. McGarity, C. Tseng, and D. P. Siewiorek, "The Design of C.fast: A Single Chip Fault Tolerant Microprocessor," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 63-69 (June 1982).

Chapter Four
Self-Testing Comparators

In Chapter 3 it was shown that duplication and comparison is an effective technique for implementing self-checking computing elements. One of the potential weaknesses of this technique is that if the comparator fails, a subsequent mismatch between the outputs of the two functional modules may not be detected and erroneous information will be accepted as correct by the rest of the system. Hence, it is imperative that faults in the comparator be detected soon after they occur so that the rest of the system can be informed that the supposedly self-checking computing element has lost its self-checking capabilities. As discussed in Chapter 3, this requirement can be fulfilled by using a *self-testing* comparator that signals its own faults during normal operation. The design, implementation, and application of such a comparator are discussed in this chapter.

As discussed in Chapter 2, large VLSI chips are far too complex to allow detailed analysis of all the possible physical defects that can occur and of the effects of these defects on the operation of the circuit. On the other hand, PLAs are characterized by a simple regular structure and are therefore more amenable to thorough analysis. PLAs are therefore a preferred implementation technique for combinational circuits whose behavior under faults is of critical importance. Since the correct operation of the self-testing comparator is critical to the error-detection technique proposed in this thesis, this chapter focuses on the use of PLAs for implementing the comparator.

Section 4.1 presents a new fault model for MOS PLAs that is based on the fault model for general MOS VLSI circuits that was discussed in Chapter 2. The model reflects some physical defects that are likely to occur in integrated circuits but are not taken into account in previously published models.

Comparators implemented with two-level AND-OR or NOR-NOR circuits, which are claimed to be self-testing, have been presented in the literature [Cart68, Wang79]. Some of this work and related terminology are reviewed in Section 4.2. The comparator implementation discussed in this chapter is based on the designs proposed by Carter and Schneider [Cart68] and Wang and Avizienis [Wang79]. In Section 4.3 it is proven that these previous designs, which require that the number of product terms grow

exponentially with the number of input bits, are *optimal* in terms of size. The correct operation of the proposed circuit under fault-free conditions is verified in Section 4.4. In Section 4.5 it is shown that the circuit cannot be self-testing with respect to several types of faults unless a few simple layout guidelines are observed in its implementation. Section 4.6 presents a formal proof that a comparator implemented as a NOR-NOR PLA, based on the design of Wang and Avizienis [Wang79] and following the layout guidelines of Section 4.5, is self-testing with respect to all single faults in the fault model introduced in Section 4.1. Section 4.7. discusses the application of the self-testing comparator as a basic building block for implementing fault-tolerant systems.

4.1. A Fault Model for MOS PLAs

The effect of a single physical defect on the output of an integrated circuit is dependent on layout details such as which lines are adjacent, which lines cross each other, *etc.* One of the advantages of using PLAs is that their regular structure simplifies analysis of the effects of faults on its outputs and therefore facilitates test vector generation and determination of fault coverage. This section describes how the faults discussed above affect the operation of a two-level NOR-NOR MOS PLA. To facilitate this discussion, a "typical" NMOS PLA is shown in Fig. 4.1.

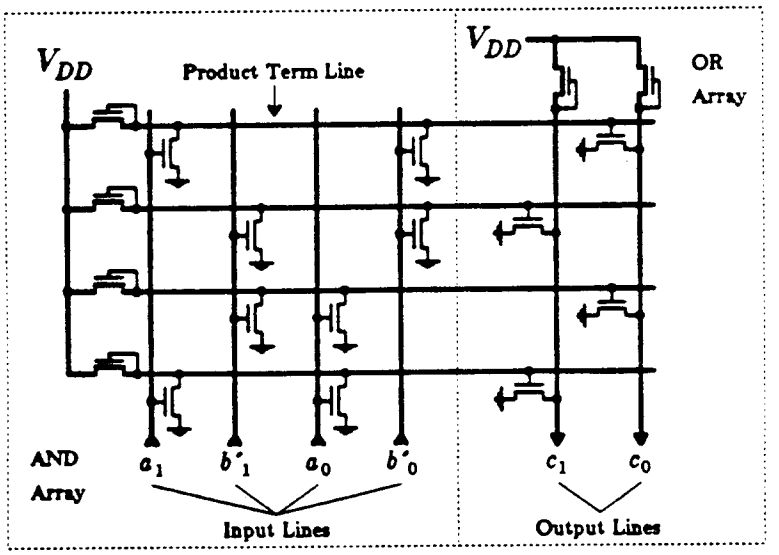


Fig. 4.1: A Self-Testing NMOS Two-Rail Code Checker

A commonly used fault model for MOS PLAs includes three types of faults [Mak82, Osta79, Wang79]:

(I) A stuck-at fault on an input line, product term line, or output line.

(II) A short between two adjacent or crossing lines that forces both of them to the same logic value.

(III) A missing or extra crosspoint device in the AND array or in the OR array.

The first two types of faults were explained above and correspond directly to physical defects in the circuit. The third type of faults refers to faults whose effect on the operation of the circuit is equivalent to the effect of a missing or extra crosspoint device. This may be the result of the gate of the crosspoint device stuck-at its "off" value (0 for NMOS, 1 for PMOS) when it should be connected to an input or product term line, or connected to an input or product term line when, by design, it should be permanently held at its "off" value.

A missing crosspoint device has the same effect as a device that always misinterprets the line that drives it as a logic 0 even when it is a logic 1. Thus, a missing crosspoint device fault in the AND array is equivalent to a weak 1 fault on the corresponding input line while a missing crosspoint device fault in the OR array is equivalent to a weak 1 fault on the corresponding product term line. Hence, if weak 1 faults on input lines and product term lines are considered, there is no need to consider missing crosspoint device faults separately.

The above three fault types do not include weak 0 and weak 1 faults or breaks in lines that are not equivalent to stuck faults. Since breaks in lines are one of the main causes of failures in VLSI circuits [Cour81, Gali80], it is clear that the above simple fault model does not accurately reflect likely physical defects in a MOS PLA.

Some of the effects of breaks on general MOS circuits cannot occur in PLAs due to their structure. This fact can be used to reduce the complexity of the fault model that must be considered in analyzing the operation of PLAs under faults. One such simplification relies on the fact that input lines are only connected to gates of devices in the PLA. A break in an input line causes a segment of that line to "float" and is therefore equivalent to a weak 0 and/or weak 1 fault. Hence, if weak 0/1 faults on inputs lines are taken into account, there is no need to consider breaks in input lines separately.

Another important simplification of the fault model is based on the fact that product term lines and output lines only have one pull-up (load) device and that this device is independent of the inputs to the circuit. Every point in a product term or output line is

either connected to the single pull-up (load) or permanently disconnected from it (due to a break). For any input, segments of the line that are connected to the pull-up are either set to logic 1 or set to logic 0 by some pull-down device that is turned on by that particular input. A segment of the line that is disconnected from the pull-up is set to logic 0 by the first input that is supposed to set it to 0 and stays stuck-at-0 for a long time thereafter. Hence no state is preserved on lines between inputs (clock phases). The troublesome faults that can convert a general combinational circuits into a sequential circuit cannot occur.

Based on the above discussion, a realistic fault model for PLAs must include weak 0/1 faults as well as the possible effects of breaks in product term lines and output lines. Specifically, the following faults must be considered:

(A) Weak 0 or weak 1 or both on one input line.

(B) A short between two adjacent input lines.

(C) Weak 0 or weak 1 or both on one product term line.

(D) A short between two adjacent product term lines.

(E) Weak 0 or weak 1 or both on one output line.

(F) A short between two adjacent output lines.

(G) A short between an input line and a crossing product term line.

(H) A short between a product term line and a crossing output line.

(I) An extra crosspoint device in the AND array.

(J) An extra crosspoint device in the OR array.

(K) A break in a product term line.

(L) A break in an output line.

4.2. Background and Terminology

Since self-testing comparators are key elements in many computer systems with on-line error detection, the design and implementation of self-testing comparators has been an active research area for many years. This subsection discusses some of that work and introduces the terminology and notation that will be used in the rest of this chapter.

It is assumed that two n-bit vectors, $A = (a_{n-1}, a_{n-2} \ldots, a_0)$ and $B = (b_{n-1}, b_{n-2} \ldots, b_0)$, are to be compared. In much of the literature *two-rail code checkers* rather than *comparators* are discussed. Given two n-bit vectors $X = (x_{n-1}, x_{n-2} \ldots, x_0)$ and $Y = (y_{n-1}, y_{n-2} \ldots, y_0)$, the combined $2n$ bit vector

$XY = (x_{n-1}, \ldots, x_0, y_{n-1}, \ldots, y_0)$ is a two-rail code word if $x_i = y'_i$ for all i such that $0 \leq i \leq n-1$ (where y'_i means the complement of y_i). An n-bit vector whose elements are the complements of the elements of B will be denoted B'. Thus, $B' = (b'_{n-1}, b'_{n-2}, \ldots, b'_0)$. A two-rail code checker whose input is the bit vector AB' is, effectively, a comparator of vectors A and B. Assuming that all the input bits are available in both complemented and uncomplemented form, there is no difference between the design of comparators and two-rail code checkers. Hence, the terms "comparator" and "two-rail code checker" will be used interchangeably.

Pioneering work in the field of self-testing checkers was reported by Carter and Schneider [Cart68] whose design of a self-testing two-rail code checker serves as a basis for the comparator discussed in this paper. For the case $n = 2$, Carter and Schneider presented a design of a circuit that checks whether the input is a two-rail code word and that is also self-testing with respect to any single stuck-at fault [Cart68]. The circuit, shown in Fig. 4.2 [Cart68], has two output lines c_1 and c_0 where $(c_1,c_0) = (0,1)$ or $(c_1,c_0) = (1,0)$ for code input, and $(c_1,c_0) = (0,0)$ or $(c_1,c_0) = (1,1)$ for noncode input.

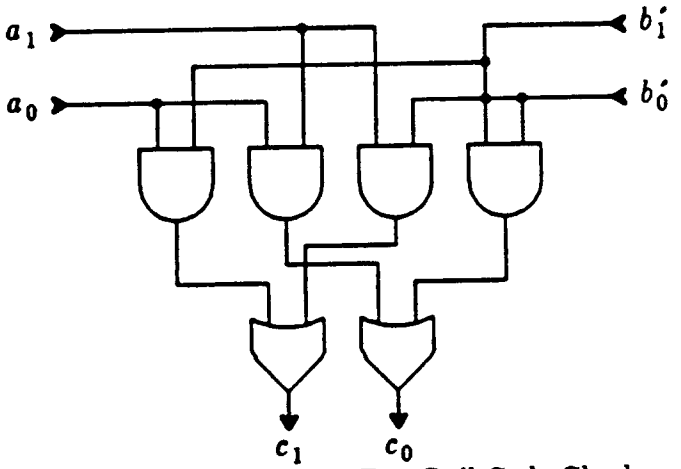


Fig. 4.2: A Self-Testing Two-Rail Code Checker

Carter and Schneider's checker has the property that, with no faults, every line in the circuit is 0 for at least one code input and 1 for at least one code input. If any line is stuck-at-0 (s-a-0) or s-a-1, the code input for which the line is supposed to be at 1 or 0, respectively, results in the output (0,0) or (1,1).

Wang and Avizienis [Wang79] extended Carter and Schneider's design to arbitrary size input vectors. For each one of the 2^n input code words there is a single unique product term that is selected (set to 1) only by that code word. Each product term line

selects exactly one of the two output lines. Depending on the parity of the vector $A = (a_{n-1}, a_{n-2}, \ldots, a_0)$, half the code inputs select output c_0 and the other half select output c_1.

The checker proposed by Wang and Avizienis can be described by sum-of-products equations as follows: For any integer k, let I_k denote the set of the k integers between 0 and $k-1$, *i.e.* $I_k = \{0,1,\ldots,k-2,k-1\}$. If Q is a set, let $|Q|$ denote the number of elements in Q.

$$c_0 = \sum_{\{Q \mid Q \subset I_n \text{ and } |Q| \text{ even}\}} \left\{ \left[\prod_{\{i \mid i \in Q\}} a_i \right] \left[\prod_{\{j \mid j \in (I_n - Q)\}} b'_j \right] \right\} \tag{1}$$

$$c_1 = \sum_{\{Q \mid Q \subset I_n \text{ and } |Q| \text{ odd}\}} \left\{ \left[\prod_{\{i \mid i \in Q\}} a_i \right] \left[\prod_{\{j \mid j \in (I_n - Q)\}} b'_j \right] \right\}$$

As will be shown in Section 4.4, similar functionality can be achieved in NOR-NOR form based on the Equation (2). An NMOS PLA which implements these equations for the case $n = 2$ is shown in Fig. 4.1. It should be noted that there are a total of $2n$ input bits to this circuit: all the "a" bits uncomplemented and all the "b" bits complemented. Each "product term" contains exactly n literals.

$$c_0 = \underset{\{Q \mid Q \subset I_n \text{ and } |Q| \text{ odd}\}}{NOR} \left\{ NOR\left[\{a_i \mid i \in Q\} \bigcup \{b'_j \mid j \in (I_n - Q)\} \right] \right\} \tag{2}$$

$$c_1 = \underset{\{Q \mid Q \subset I_n \text{ and } |Q| \text{ even}\}}{NOR} \left\{ NOR\left[\{a_i \mid i \in Q\} \bigcup \{b'_j \mid j \in (I_n - Q)\} \right] \right\}$$

4.3. Optimal Design of Self-Testing Comparators Using Two-Level Logic

Published work on self-testing checkers usually consists of a circuit design and a proof that the circuit is self-testing. There has been no attempt to show that the proposed designs are optimal in any respect. This section discusses the design of self-testing comparators which are optimal with respect to the number of output lines, the number of input lines, and the number of product term lines.

Since the comparator must be self-testing with respect to stuck-at faults on the output lines, it must have at least two output lines [Cart68]. The use of more than two

lines has been proposed [Son81]; however, since limited communication bandwidth is a severe problem in VLSI systems, it is preferable to minimize the bandwidth dedicated to transmitting self-testing information. Hence only comparators with two output lines will be considered.

There are two possible ways to "code" the output from the comparator and still allow self-testing of the output lines: (A) The code (correct) output is (0,1) or (1,0) and the noncode (error indication) output is (0,0) or (1,1). (B) The code output is (0,0) or (1,1) and the noncode output is (0,1) or (1,0). Option (A) is preferable since it allows the comparator to be self-testing with respect to shorts between the output lines as well as any other fault that causes a *unidirectional error*. A unidirectional error means that, due to a fault, some lines that are supposed to be at logic 0 are at logic 1 *or* some of the lines that are supposed to be at logic 1 are at logic 0, *but not both*. It has been shown that the faults that are most likely to occur in PLAs (fault types (I), (II), and (III) in Section 4.1), can result only in a unidirectional error [Mak82]. Therefore only comparators with the option (A) encoding of the outputs will be considered.

The self-testing comparator design proposed by Wang and Avizienis requires 2^n product terms for comparing n-bit vectors. However, it is possible to implement a comparator that has two outputs that are (0,1) or (1,0) for code inputs and (1,1) for noncode inputs based on the equations:

$$c_0 = a'_0 + b'_0 + \sum_{i=1}^{i=n-1} (a_i b'_i + a'_i b_i) \qquad c_1 = a_0 + b_0 + \sum_{i=1}^{i=n-1} (a_i b'_i + a'_i b_i)$$

This comparator is self-testing with respect to faults in the input lines and output lines but requires only $4n$ product terms. Unfortunately, this comparator is *not* self-testing with respect to stuck faults on the product term lines. The question thus arises what is the minimum number of product terms necessary for a comparator that is self-testing with respect to a realistic fault model that also takes into account faults affecting the product term lines.

Although the design of a self-testing comparator presented by Wang and Avizienis [Wang79] uses 2^n product terms, one for each code input, this is never shown to be a necessary property of self-testing comparators implemented with PLAs. In several papers [Khak82, Wang79] it is claimed that it is "desirable" to use PLAs that are *nonconcurrent*, *i.e.*, where each code input selects only one product term. Wang and

Avizienis propose a general approach to the design of self-testing PLAs that always results in a nonconcurrent circuit. They also give an example of a PLA where concurrency leads to a circuit which is not self-testing [Wang79]. However, nonconcurrency is not presented as a *necessary* property nor is there any mention of a problem with product terms that are selected by more than one code input.

In the following we will show that the exponential growth in the number of product term lines is indeed necessary for self-testing. For any two-level NOR-NOR implementation, we also show that every code input must select exactly one product term line and that no two different code inputs can select the same product term line. This is necessary even if the only faults considered are single stuck-at faults on the input, output and product lines. The proof that the same requirement also applies to two-level AND-OR implementations is almost identical and will not be presented explicitly.

Lemma 1: Every product term must be selected (set to 1) by least one code word.

Proof: Assume that there is a product term that is not selected by any code word. A stuck-at-0 fault on this product term line will not be detected during normal operation, thus violating the self-testing requirement.

Lemma 2: Every code word must select at least one product term.

Proof: If there is some code word that does not select any product term, the comparator output for that code word will be the noncode output (1,1), which incorrectly signals an error.

Lemma 3: All the product terms selected by a single code word must be connected to the same single output in the OR array.

Proof: If any of the product terms selected by a code word is connected to both outputs in the OR array, then, for that code word, the output will be the noncode output (0,0), which incorrectly signals an error. Similarly, the output will be (0,0) if the product terms are not all connected to the same output line.

Lemma 4: No product term can be selected by more than one code word.

Proof: ***By contradiction.*** Assume that P_i is a product term which is selected by the two code inputs

$AA = (a_{n-1}, \ldots, a_0, a_{n-1}, \ldots, a_0)$ and $BB = (b_{n-1}, \ldots, b_0, b_{n-1}, \ldots, b_0)$.

Since the two code words are different, there exists an integer k $(0 \leq k \leq n-1)$ such

that $a_k \neq b_k$. P_i is selected only if all the literals in the expression corresponding to P_i are 0. Since P_i is selected by both code words, it must be independent of bit k from the two functional modules. Hence P_i is also selected by the code input $WW = (a_{n-1}, \ldots, a'_k, \ldots, a_0, a_{n-1}, \ldots, a'_k, \ldots, a_0)$ and by the *noncode* input $Q = (a_{n-1}, \ldots, a'_k, \ldots, a_0, a_{n-1}, \ldots, a_k, \ldots, a_0)$.

Since Q is a noncode input, the corresponding output produced by the comparator must be noncode. When P_i is selected, it sets to 0 the one output line it is connected to. Hence, Q must select another product term, P_j, connected to the other output line, so that the noncode output (0,0) will be produced. By Lemma 1, P_j must also be selected by at least one code word $CC = (c_{n-1}, \ldots, c_k, \ldots, c_0, c_{n-1}, \ldots, c_k, \ldots, c_0)$.

Since in CC, bit k from both functional modules is the same, and in Q bit k from one unit is the complement of bit k from the other unit, P_j must not include the literal corresponding to bit k from at least one of the two functional modules. Hence, since Q selects P_j, either AA or WW must also select P_j. Without loss of generality, assume WW selects P_j. From the above, WW also selects P_i. But in the OR array P_j is connected to a different output line from P_i. Hence, Lemma 3 above is "violated" and the code word WW results in the *noncode* output (0,0). Thus the assumption that there exists a product term that is selected by more than one code input must be incorrect.

Lemma 5: Every code word must select one, and only one, product term.

Proof: By Lemma 2, every code word must select *at least* one product term. Assume that the code word $AA = (a_{n-1}, \ldots, a_0, a_{n-1}, \ldots, a_0)$ selects the two product terms P_i and P_j. By Lemma 4, no other code word except AA can select P_i or P_j. Hence, a stuck-at-0 fault on the P_i or P_j lines can only be detected by the input AA. By Lemma 3, both P_i and P_j must be connected to the same output line in the OR array. Hence, when the code word AA is applied, the output from the PLA will be the same whether or not one of the product term lines P_i or P_j is stuck-at-0. Thus a stuck-at-0 fault on one of the product term lines P_i or P_j will not be detectable by any code word, thus violating the self-testing requirement.

Theorem 1: A self-testing comparator of two n-bit vectors that has two output lines and is implemented as a two level NOR-NOR PLA, must have *exactly* 2^n product terms.

Proof: By Lemma 1, every product term line is selected by at least one code word. By Lemma 5, every code word selects one, and only one, product term line. Hence, the

number of product term lines is equal to the number of code words. Since there are n bits of output from each one of the two functional modules, there are 2^n code words. Therefore the number of product term lines is *exactly* 2^n .

Q.E.D.

In summary: any comparator of two n-bit vectors must have at least $2n$ input lines (two lines for every pair of bits being compared), at least two output lines are necessary, and, based on the proof presented in this section, exactly 2^n product term lines are necessary for any two-level NOR-NOR implementation. Hence, the design based on Equation (2), which was discussed in the previous section, is optimal. In the next three sections a PLA implementation of a self-testing comparator based on this design is analyzed in detail.

4.4. Fault-Free Operation of the Comparator

In the previous section it was shown that any self-testing comparator implemented as a single two-level NOR-NOR PLA must have 2^n product terms. This section and the two subsequent sections discuss a specific self-testing comparator, based on Equation (2) in Section 4.2, which satisfies this necessary property.

Although a comparator based on Equation (2) has been discussed in the literature [Wang79], there is no rigorous proof that it indeed functions as a comparator. Such a proof is presented in this section. To prove that, with no faults, the circuit described by Equation (2) is a comparator, it is shown that if $A = B$, the output is (0,1) or (1,0) . It is then shown that if $A \neq B$, the output is (0,0) or (1,1) .

If $A = B$, there are exactly n ones and n zeros at the inputs. If U is a set of integers $U = \{i \mid a_i = 0\}$, then for every integer j such that $j \in (I_n - U)$, $a_j = b_j = 1$. Thus, $b'_j = 0$, and the one product term that corresponds to $Q = U$ in Equation (2) is selected. Every other product term includes the literal a_j for some $j \in (I_n - U)$ or b'_i for some $i \in U$. Hence, all of the other product terms are set to 0. Thus, only the one output line connected to the single selected product term is set to 0, and the output is (0,1) or (1,0) .

If $A \neq B$, the two bit-vectors differ by at least one bit. Consider the product term

$$NOR\left[\{a_i \mid i \in Q\} \bigcup \{b'_j \mid j \in (I_n - Q)\}\right] \tag{3}$$

for some $Q \subset I_n$. Assume that A and B differ in bit r, $r \in I_n$, so $a_r = 1$ and

$b'_r = 1$ in the input AB. If $r \in Q$, the product term is set to 0 since it contains the literal a_r. If $r \notin Q$, the product term is set to 0 since it contains the literal b'_r. Hence, all of the product terms are set to 0 and the output is (1,1).

Since the two bit-vectors differ by at least one bit, if there does *not* exist any integer $r \in I_n$ such that $a_r = 1$ and $b'_r = 1$, there must exist an integer $s \in I_n$ such that $a_s = 0$ and $b'_s = 0$ in the input AB. If AB doesn't select any product term, the output is (1,1). Assume that the product term that corresponds to $Q = Q_1$ (Equation (3)) is selected. If $s \in Q_1$, consider the set $Q_2 = Q_1 - \{s\}$. Since $Q_2 \subset Q_1$, $I_n - Q_2 = I_n - Q_1 + \{s\}$, and $b'_s = 0$, the product term that corresponds to $Q = Q_2$ will also be selected. If $s \notin Q_1$, consider the set $Q_3 = Q_1 + \{s\}$. Since $a_s = 0$ and $I_n - Q_3 \subset I_n - Q_1$, the product term that corresponds to $Q = Q_3$ will also be selected. Thus, either the product terms corresponding to Q_1 and Q_2 will be selected, or the product terms corresponding to Q_1 and Q_3 will be selected. The number of elements in Q_1 is one greater than the number of elements in Q_2 and is one less than the number of elements in Q_3. Hence, either $|Q_2|$ and $|Q_3|$ are even while $|Q_1|$ is odd, or $|Q_2|$ and $|Q_3|$ are odd while $|Q_1|$ is even. Thus, the product terms corresponding to Q_2 and Q_3 are connected to the same output line, which is different from the output line to which the product term corresponding to Q_1 is connected. Therefore, product terms connected to both output lines are always selected and the output is (0,0).

4.5. Identification and Elimination of Undetectable Faults

Given that the circuit described by Equation (2) functions as claimed when it is *fault-free*, it remains to be shown that the circuit is self-testing with respect to any single fault in the fault model described in Section 4.1. Specifically, it must be shown that for any such fault there exists a code input that results in a noncode output (0,0) or (1,1) from the comparator. In this section it is shown that there are a few faults in the fault model with respect to which the circuit is *not* self-testing. These problematic faults are referred to as *undetectable* faults. Layout guidelines that prevent these faults from occurring in the actual circuit are discussed.

4.5.1. A Short Between Adjacent Product Term Lines

One of the possible faults is a short between two adjacent product term lines that forces both of the lines to logic 1 when they are supposed to be carrying different values (fault type (D)). If the two product term lines are connected to the same output line, there is no code input that results in a noncode output. In fact, the circuit continues to function correctly despite this fault. The reason for this is that if one of the product term lines connected to an output line is selected, that output line is set to logic 0 regardless of the value of any other product term connected to it. It is undesirable to allow this fault to remain undetected since the situation may deteriorate in time and intermittently cause weak 0 or weak 1 faults that will not be detected and will later combine with an additional fault to cause more serious undetectable faults.

As indicated by Wang and Avizienis [Wang79], the possibility that this undetectable fault will occur can be eliminated by ensuring that product term lines connected to the same output line are not adjacent. Since the same number of product term lines are connected to each output line, this guideline is easy to obey and incurs no penalty in terms of the size or performance of the circuit. The guideline is satisfied by simply alternating between product term lines connected to one output line and those connected to the other line.

4.5.2. A Short Between a Product Term Line and an Output Line

Another potentially undetectable fault is a short between a product term line, P_i, and an output line, c_m, where there is no device at the crosspoint of the two lines. This fault is undetectable if whenever the two lines are supposed to carry a different logic value, they are both forced to logic 1.

The short between P_i and c_m is not detectable since the faulty circuit will behave as follows: For the code input XX that is supposed to select P_i, P_i is supposed to be at logic 1 and c_m at logic 1 (since the other output line, $c_{m'}$, is supposed to be at logic 0). Hence there is no change in the output from the circuit. On the other hand, P_i is supposed to be at logic 0 and c_m is supposed to be at logic 1 for every code input, YY, such that $YY \neq XX$ and the number of a_i ($i \in I_n$) inputs that are at logic 0 in YY has the *same* parity as the number of a_i inputs that are at logic 0 in XX. For these code inputs, P_i is forced to logic 1 but this has no effect on $c_{m'}$ which is supposed to be at

logic 0. For the remaining 2^{n-1} code inputs, P_i is supposed to be at logic 0 and c_m is supposed to be at logic 0. Hence there is no change in the output from the circuit.

The short between P_i and c_m can be made detectable if it is possible to ensure that when P_i is at logic 0, it forces c_m to logic 0 as well. In NMOS, this can be done by using large crosspoint devices in the AND array so that a single device can pull down two load devices — the output line pull-up as well as the product term line pull-up. In CMOS, this can be done by using large crosspoint devices in the AND array so that a single device can discharge the precharged output line and product term line together within the circuit's clock period. Unfortunately, larger AND array crosspoint devices lead to a larger PLA that is also slower due to larger capacitances.

It is possible that, due to a short between a product term line, P_i, and an output line, c_m, both lines always assume the value at which c_m is supposed to be (c_m dominates). In this case, the short is undetectable regardless of whether or not there is a device at the crosspoint of the two lines. This short is not detectable since the faulty circuit behaves as follows: The output line c_m always forces P_i to the value that c_m is supposed to be at. If there is a device at the crosspoint of P_i and c_m, there is no device at the crosspoint of P_i and the other output line, $c_{m'}$. Hence $c_{m'}$ cannot be affected by P_i, so the output of the circuit cannot be affected and is a code output despite the fault. If there is no device at the crosspoint of P_i and the output line, c_m, then when c_m forces P_i to logic 0, $c_{m'}$ is supposed to be at logic 1 and a possible change in P_i to logic 0 cannot change the value of $c_{m'}$ from logic 1. When c_m forces P_i to logic 1, $c_{m'}$ is supposed to be at logic 0 so the change in P_i from logic 0 to logic 1 cannot possibly change the value of $c_{m'}$ from logic 0 to logic 1. Hence the output from the circuit remains the code output despite the fault.

The short between P_i and c_m can be made detectable if it is possible to ensure that when P_i is at logic 0, it forces c_m to logic 0 as well. As previously discussed, this can be done by using large crosspoint devices in the AND array.

4.5.3. Shorts Resulting in Simultaneous Weak 0 and Weak 1 Faults

In this subsection we consider the possibility that, due to a short, two lines that are supposed to carry complementary values are both forced to a value between logic 0 and logic 1. The result is a weak 0 fault on one of the lines and a weak 1 fault on the other

line. Such shorts may be undetectable by any code input. To show that the circuit is not self-testing with respect to such a short, it is sufficient to show that the fault is undetectable under the worst possible combination of devices that misinterpret the values on the lines.

1) *A Short Between Adjacent Product Term Lines:* As discussed in Subsection 4.5.1, adjacent product term lines should be connected to different output lines. If a short between two product term lines, P_i and P_j, forces both to a value between logic 0 and logic 1 when they are supposed to be carrying different values, this fault may be undetectable. For a code input XX, the short can affect the output only if XX is supposed to select either P_i or P_j. Without loss of generality, assume that XX is supposed to select P_i. All other product term lines (including P_j) are not supposed to be selected by XX. However, a short between P_i and P_j can cause the OR array device connected to P_i to misinterpret it as logic 0 and the device connected to P_j to misinterpret it as logic 1. Hence, despite the fault, only one product term line (P_j) is interpreted as being selected and the output from the circuit is a code output. Thus, this short is not detected by any code input.

It can be shown that there exists a *noncode* input that, due to the short between product term lines, results in code output. Hence this short, that is not detectable by code inputs, can mask noncode inputs. Thus, the PLA should be laid out in such a way that either this short cannot occur, or if it does occur, both lines are guaranteed to be forced to the same logic value instead of some value between logic 0 and logic 1.

We have already shown that the crosspoint devices in the AND array should be made large enough so that they can pull down both the product term line and an output line that it may be shorted to. If pull-ups of the same size are used for the product term lines and the output lines, each crosspoint device in the AND array is also able to pull down *two* product term lines. Hence, a short between two product term lines is guaranteed to force them both to logic 0 when they are supposed to be carrying complementary values. It will be shown in Section 4.6 that this ensures that the short can be detected by some code input.

2) *A Short Between Adjacent Input Lines:* A short between adjacent input lines may also be undetectable by any code input if, whenever the lines are supposed to be carrying complementary values, both lines are forced to a value between logic 0 and

logic 1. Consider a short between two adjacent input lines a_h and a_j ($h \neq j$). There exists a code input $XX = (x_{n-1}, \ldots, x_0, x'_{n-1}, \ldots, x'_0)$ for which a_h is supposed to be at logic 0 and a_j is supposed to be at logic 1. Clearly $x_h = 0$ and $x_j = 1$ so

$$XX = (x_{n-1}, \ldots, x_{h+1}, 0, x_{h-1}, \ldots, x_{j+1}, 1, x_{j-1}, \ldots, x_0,$$
$$x'_{n-1}, \ldots, x'_{h+1}, 1, x'_{h-1}, \ldots, x'_{j+1}, 0, x'_{j-1}, \ldots, x'_0)$$

Assume that the single product term line that is supposed to be selected by XX is P_i. Since $x_h = 0$, there is a device CA_{hi} at the crosspoint of the input line a_h and the product term line P_i. Assume that, due to the short, the common value of both a_h and a_j is forced to some value between logic 0 and logic 1 and that CA_{hi} misinterprets line a_h to be at logic 1. Hence product term line P_i is *not* selected by code input XX. In the fault-free circuit, the code input

$$YY = (x_{n-1}, \ldots, x_{h+1}, 0, x_{h-1}, \ldots, x_{j+1}, 0, x_{j-1}, \ldots, x_0,$$
$$x'_{n-1}, \ldots, x'_{h+1}, 1, x'_{h-1}, \ldots, x'_{j+1}, 1, x'_{j-1}, \ldots, x'_0)$$

is supposed to select product term line P_k. Hence there is a device CA_{jk} at the crosspoint of input line a_j and product term line P_k. Assume that, due to the short, when the input is XX, CA_{jk} misinterprets a_j to be a logic 0 although it is supposed to be at logic 1. In addition, we assume that CA_{hi} and CA_{jk} are the only AND array crosspoint devices that misinterpret the values of a_h and a_j (in particular CA_{hk} interprets a_h *correctly*). Under these assumptions, all the input lines that are supposed to be at logic 0 when the input is YY are interpreted as being at logic 0 by all the AND array crosspoint devices connected to P_k when the input is XX. Hence P_k is selected by input XX while P_i is not selected by XX. Since no other crosspoint devices are effected by the short, no other product term line except P_k is selected by XX, and the output is a code output. This short does not affect the output from the circuit for any other code input since such input selects a product term other then P_k or P_i. Hence, the short is not detectable by any code input.

In the fault-free circuit, the *noncode* input

$$W = (x_{n-1}, \ldots, x_{h+1}, 0, x_{h-1}, \ldots, x_{j+1}, 1, x_{j-1}, \ldots, x_0,$$
$$x'_{n-1}, \ldots, x'_{h+1}, 1, x'_{h-1}, \ldots, x'_{j+1}, 1, x'_{j-1}, \ldots, x'_0)$$

does not select *any* product term and the output is noncode. However, due to the short described above between a_h and a_j, W selects P_k and the result is a *code* output from the circuit. Hence this short, that is not detectable by code inputs, masks a noncode

input.

It can be shown that if the adjacent input lines are a_h and b'_j, a short between these lines may also be undetectable by code inputs and can mask noncode inputs. Thus, in order to ensure that the comparator is self-testing, it is necessary to prevent shorts between input lines that can force both lines to a value between logic 0 and logic 1 from occurring. This can be done by laying out the PLA so that the separation between input lines is large enough that the probability of a short between them is negligible. Alternatively, the circuits that drive the inputs of the PLA can be designed so that a single pull-down device can overcome *two* pull-up devices so that a short between input lines when they are supposed to be carrying different values will always results in both of them being forced to logic 0. Unfortunately, these solutions lead to a larger PLA that is also slower due to larger capacitances.

3) *A Short Between an Input Line and a Product Term Line:* Using arguments similar to the above, it can be shown that if a short between an input line and a product term line is allowed to force both of them to a value between logic 0 and logic 1, an undetectable fault, that can mask noncode inputs, may result. Here again, one way to prevent this situation is to guarantee that when the lines are supposed to be at complementary values they are both always forced to logic 0. This can be done using large pull-down devices in the circuits that drive the inputs of the PLA and using large AND array crosspoint devices. A single AND array crosspoint device or a single pull-down in an input driver must be able to overcome both the pull-up device of the input driver and the pull-up device of the product term line.

4.5.4. Layout Guidelines for Eliminating Undetectable Faults

In the previous three subsections we identified several possible faults that are not detectable by any code inputs. All of these faults are shorts between adjacent or crossing lines. In particular, any short that results in both lines being forced to a value between logic 0 and logic 1 when they are supposed to be carrying complementary values may lead to an undetectable fault. The layout guidelines for preventing these faults from occurring in the actual circuit are summarized below.

(1) Adjacent product term lines must be connected to OR array crosspoint devices that control different output lines.

(2) The AND array crosspoint devices must be large enough so that a single device can pull down two pull-ups — a product term line pull-up and an output line pull-up or two product term line pull-ups.

(3) The circuits that drive the inputs of the PLA must be designed so that a single pull-down device can overcome *two* pull-up devices.

(4) The separation between adjacent input lines and between adjacent product term lines should be larger than the minimum separation required by the design rules. This can help reduce the probability of a short between adjacent lines.

4.6. The Self-Testing Property of the Comparator

In the previous section it was shown that the proposed comparator is *not* self-testing with respect to some of the possible faults, unless certain guidelines about the layout of the circuit and the size of some of its devices are followed. In this section we will show that the circuit is self-testing with respect to all the other faults in the fault model. It is assumed that some measures, such as those discussed in the previous section, are taken so that the undetectable faults cannot occur. In particular, it is assumed that if there is a short between two lines and the lines are supposed to be carrying complementary values, the value of one of the lines is modified so that they both carry the logic value of the other line.

4.6.1. A Weak 0 and/or Weak 1 Fault on a Single Input Line

1) *A Weak 0 Fault:* Assume that the input line with a weak 0 fault is a_k for some $k \in I_n$. By definition, there is at least one AND array crosspoint device, CA_{ki}, connected to a_k that always misinterprets a logic 0 on a_k as a logic 1. Hence, the device CA_{ki} is always turned on. Thus, the product term line P_i that is connected to CA_{ki} can never be selected. Therefore the code input that is supposed to select P_i results in no product term line being selected and the output is noncode (1,1). An identical argument can be made regarding a weak 0 fault on a b^*_j $(j \in I_n)$ input line.

In the presence of a weak 0 fault on one of the input lines, for every crosspoint device that misinterprets the input line to be a logic 1 when it is supposed to be a logic 0, the code input that selects the corresponding product term line in the fault-free circuit results in a (1,1) output. Thus the number of code inputs that detect this fault varies between 1 and 2^{n-1}, depending on the number of affected crosspoint devices.

2) *A Weak 1 Fault:* Assume that the line with a weak 1 fault is a_k for some $k \in I_n$. By definition, there is at least one AND array crosspoint device, CA_{kl}, connected to a_k that always misinterprets a logic 1 on a_k as a logic 0. We denote the product term line connected to that crosspoint device by P_i. In the fault-free circuit, P_i is selected by some code input $XX = (x_{n-1}, \ldots, x_0, x'_{n-1}, \ldots, x'_0)$. Since there is a device at the crosspoint of a_k and P_i, the literal a_k is in the product term that corresponds to P_i. Hence $x_k = 0$. Thus,

$$XX = (x_{n-1}, \ldots, x_{k+1}, \mathbf{0}, x_{k-1}, \ldots, x_0, x'_{n-1}, \ldots, x'_{k+1}, \mathbf{1}, x'_{k-1}, \ldots, x'_0).$$

In the fault-free circuit, the code input

$$YY = (x_{n-1}, \ldots, x_{k+1}, \mathbf{1}, x_{k-1}, \ldots, x_0, x'_{n-1}, \ldots, x'_{k+1}, \mathbf{0}, x'_{k-1}, \ldots, x'_0)$$

selects some other product term line P_j. Since CA_{kl} misinterprets a logic 1 on a_k to be a logic 0, code input YY selects P_i. Since there is no device at the crosspoint of a_k and P_j, P_j is independent of a_k. Thus YY also selects P_j despite the fault.

Since the number of a_i ($i \in I_n$) inputs that are at logic 0 in XX has a different parity from the number of a_i inputs that are at logic 0 in YY, P_i and P_j are connected to different output lines (see Equation (2) Section 4.2). Since in the faulty circuit the code word YY selects both P_i and P_j, the output is (0,0). An identical argument can be made regarding a weak 1 fault on a b'_j ($j \in I_n$) input line.

4.6.2. A Short Between Two Adjacent Input Lines

As previously mentioned, we assume that appropriate layout guidelines are followed so that a short between lines always forces both of the lines to the same logic value rather then to a value between logic 0 and logic 1. Since the inputs to the comparator are the outputs from one functional module and their complements from the duplicate module, no two input lines are supposed to have the same value for *all* code inputs. If the two adjacent shorted lines are a_k and b'_k ($0 \leq k \leq n-1$), *every* code input is transformed to noncode input which, as previously shown, results in (0,0) or (1,1) output. Any other two input lines are supposed to transfer different values for half of the code inputs. For these code inputs, the short forces a change in value on one of the lines. Since we assume that there are no other faults, this is equivalent to noncode input which, as previously shown, results in (0,0) or (1,1) output.

4.6.3. A Weak 0 or Weak 1 Fault on a Single Product Term Line

Each product term line is connected to only one output line. Hence, a weak 0 fault on a product term line is simply a stuck-at-1 fault and a weak 1 fault is a stuck-at-0 fault.

1) *A Weak 1 (s-a-0) Fault:* If one of the product terms is s-a-0, for the code input that is supposed to select that product term, all product terms are set to 0 and the output is (1,1).

2) *A Weak 0 (s-a-1) Fault:* Assume that the product term line P_i that corresponds to set $Q = Q_1$ in Equation (3), is s-a-1. For any code input that selects a product term corresponding to some $Q = Q_2 \in I_n$, where the parity of $|Q_1|$ and $|Q_2|$ are different, product terms connected to both output lines are selected, and the output is (0,0). Thus half the code inputs will result in a (0,0) output due to the s-a-1 fault on P_i.

4.6.4. A Short Between Two Adjacent Product Term Lines

Since only one product term line is supposed to be selected by every code input, for any pair of adjacent product term lines, P_i and P_j there is one code input that is supposed to select P_i but not P_j and there is another code input that is supposed to select P_j but not P_i. We consider the three possible effects of the short when the lines are supposed to carry complementary values:

(1) Both lines are always forced to logic 0: In this case, for the two code inputs that correspond to the two product terms (*i.e.* that are supposed to select them), no product term line will be set to 1 and the output will be (1,1).

(2) Both lines are always forced to logic 1: Since the two product term lines are connected to different output lines, for the two code inputs that correspond to these product term lines, the output will be (0,0).

(3) Both product term lines always assume the value of one of the lines: Assume that the two lines are P_i and P_j, and that P_i always dominates. The code input YY, that is supposed to select P_j, does not select it, since P_j is pulled to logic 0 by P_i, which is not selected by YY. Hence YY does not select any product term and the output is (1,1). The code input XX, that selects P_i also selects P_j which is pulled to logic 1 by P_i. Since adjacent product term lines are connected to different output lines, XX results in (0,0) output.

4.6.5. A Weak 0 or Weak 1 Fault on a Single Output Line

The output lines do not fan out within the comparator circuit. Thus, we need only consider the value on the output line at the point of interface with the "outside world." Hence, a weak 0 fault on a product term line is equivalent to a stuck-at-1 fault and a weak 1 fault is equivalent to a stuck-at-0 fault.

Based on Equation (2), any code input where the number of a_i $(i \in I_n)$ bits that are at logic 0 is odd, is supposed to result in the output $(c_1,c_0) = (1,0)$. Thus, in the faulty circuit, if c_0 is s-a-1, the output is (1,1), and if c_1 is s-a-0, the output is (0,0). A similar argument can be made for any code input where the number of a_i bits that are at logic 0 is even and the output is supposed to be $(c_1,c_0) = (0,1)$. Hence 2^{n-1} code inputs will detect a s-a-1 on c_0 and a s-a-0 on c_1 while the other 2^{n-1} code inputs will detect a s-a-0 on c_0 and a s-a-1 on c_1.

4.6.6. A Short Between Two Adjacent Output Lines

There are only two output lines that are supposed to carry different values for every code input. Hence, a short will result in (0,0) or (1,1) output for every code input.

4.6.7. A Short Between an Input Line and a Crossing Product Term Line

Assume that the short is between input line a_k and product term line P_i. Let XX denote the code input that selects P_i in the fault-free circuit. We must consider the case where a_k is connected to a crosspoint device that is connected to P_i (CA_{ki} exists) as well as the case where CA_{ki} does not exist.

If CA_{ki} exists, every one of the 2^{n-1} code inputs for which a_k is supposed to be at logic 1, is supposed to result in P_i at logic 0. The code input XX is the only code input for which a_k is supposed to be at logic 0 while P_i is supposed to be at logic 1. For the rest of the $2^{n-1}-1$ code inputs, both a_k and P_i are supposed to be at logic 0.

If CA_{ki} does not exist, the product term corresponding to P_i includes the literal b'_k. For the 2^{n-1} code inputs with b'_k at logic 1, both a_k and P_i are supposed to be at logic 0. For the code input XX, b'_k is supposed to be at logic 0, and both a_k and P_i are supposed to be at logic 1. For the rest of the $2^{n-1}-1$ code inputs, b'_k is supposed to be at logic 0, a_k is supposed to be at logic 1, and P_i is supposed to be at logic 0. Thus, if CA_{ki} does not exist, there is *no* code input for which a_k is supposed to

be at logic 0 and P_i is supposed to be at logic 1.

As in the proof for a short between product term lines, we consider the three possible effects of the short when a_k and P_i are supposed to carry complementary values:

(1) Both lines are forced to logic 0: If CA_{ki} exists, for the code input XX, P_i is supposed to be the only selected product term line, while a_k is supposed to be at logic 0. We assume that, due to the short, P_i is forced to logic 0 by a_k. Hence, no product term line is selected and the output is (1,1).

On the other hand, if CA_{ki} does not exist, the literal a_k is not included in the product term that corresponds to P_i. Hence, the code input that selects P_i in the fault-free circuit is of the form:

$$XX = (x_{n-1}, \ldots, x_{k+1}, 1, x_{k-1}, \ldots, x_0, x'_{n-1}, \ldots, x'_{k+1}, 0, x'_{k-1}, \ldots, x'_0) .$$

Let YY be one of the $2^{n-1}-1$ code inputs such that $YY \neq XX$ and YY is also of the form

$$YY = (y_{n-1}, \ldots, y_{k+1}, 1, y_{k-1}, \ldots, y_0, y'_{n-1}, \ldots, y'_{k+1}, 0, y'_{k-1}, \ldots, y'_0) .$$

In the fault-free circuit YY selects some product term line P_j. Since there is no device at the crosspoint of a_k and P_j, P_j is independent of a_k so the short between a_k and P_i cannot affect P_j. Thus P_j is selected by YY despite the fault. In the fault-free circuit, the code input

$$ZZ = (y_{n-1}, \ldots, y_{k+1}, 0, y_{k-1}, \ldots, y_0, y'_{n-1}, \ldots, y'_{k+1}, 1, y'_{k-1}, \ldots, y'_0)$$

selects some product term P_s. Since there is no device at the crosspoint of b'_k and P_s, P_s is independent of b'_k. For the code input YY, a_k is supposed to be at logic 1 and P_i at logic 0. Due to the fault, P_i forces a_k to logic 0. Therefore, YY selects P_s as well as P_j. Since the number of a_i ($i \in I_n$) inputs that are at logic 0 in YY has a different parity from the number of a_i inputs that are at logic 0 in ZZ, P_j and P_s are connected to different output lines (see Equation (2) Section 4.2). Hence, for the code input YY the output is (0,0).

(2) Both lines are forced to logic 1: Let YY be one of the 2^{n-2} code inputs for which a_k is supposed to be at logic 1 *and* the number of a_i inputs that are at logic 0 in YY has a different parity from the number of a_i inputs that are at logic 0 in XX. Due to the short, when the input is YY, a_k forces P_i (that is supposed to be at logic 0) to logic 1. In addition, as in the fault-free circuit, YY selects another product term that controls a different output line from P_i. Hence the output from the circuit is (0,0).

(3) Both lines are always forced to the value of a_k *or* they are always forced to value of P_i :

(a) Line a_k always dominates: The proof is identical to case (2) above.

(b) Line P_i always dominates: There are at least $2^{n-1}-1$ code inputs of the form

$$YY = (y_{n-1}, \ldots, y_{k+1}, 1, y_{k-1}, \ldots, y_0, y'_{n-1}, \ldots, y'_{k+1}, 0, y'_{k-1}, \ldots, y'_0)$$

that do not select P_i in the fault-free circuit. In the faulty circuit, if P_i always "dominates," YY selects two product term lines that are connected to different output lines. One is the product term line selected by YY in the fault-free circuit and the other is the product term line selected by

$$ZZ = (y_{n-1}, \ldots, y_{k+1}, 0, y_{k-1}, \ldots, y_0, y'_{n-1}, \ldots, y'_{k+1}, 1, y'_{k-1}, \ldots, y'_0)$$

in the fault-free circuit. Hence, the output is (0,0).

4.6.8. A Short Between a Product Term Line and a Crossing Output Line

Assume that the short is between product term line P_i and output line c_m , where $m \in \{0,1\}$. Let m' denote 0 when m is 1 and denote 1 when m is 0. Let XX denote the code input that selects P_i in the fault-free circuit.

As in the proof for a short between product term lines, we consider the three possible effects of the short when P_i and c_m are supposed to carry complementary values:

(1) Both lines are forced to logic 0: In the fault-free circuit there are at least $2^{n-1}-1$ code inputs that do not select P_i and for which the output is $(c_m,c_{m'}) = (1,0)$. For any one of these inputs, due to the short, P_i forces c_m to logic 0 and the output is (0,0).

(2) Both lines are forced to logic 1: If there is a device at the crosspoint of P_i and c_m (CO_{im} exists), in the fault-free circuit, for the code input XX that selects P_i , the output is $(c_m,c_{m'}) = (0,1)$. In the faulty circuit, due to the short, c_m is forced to logic 1. Since none of the product term lines are affected, the output is (1,1). If CO_{im} does not exist, then, as discussed in Subsection 4.5.2, the fault cannot be detected by any code input.

(3) Both lines are always forced to the value of P_i *or* they are always forced to value of c_m :

(a) If the value of P_i always "dominates," the proof is identical to case (1) above.

(b) If the value of c_m always "dominates," then, as discussed in Subsection 4.5.3, the fault

cannot be detected by any code input.

4.6.9. An Extra Crosspoint Device in the AND Array

In the fault-free circuit, every product term line, P_i, is connected to n crosspoint devices in the AND array. For every code input, n of the input lines are at logic 0 and n are at logic 1. If, due to a fault, there are $n+1$ crosspoint devices connected to P_i, every code input turns on at least one of these devices and sets P_i to logic 0. Thus, the single code input that selects P_i in the fault-free circuit does not select P_i in the faulty circuit. Hence, for that input, no product term line is selected, and the output is (1,1).

4.6.10. An Extra Crosspoint Device in the OR Array

An extra crosspoint device in the OR array means that there is one product term line, P_i that is connected to both output lines. Hence, for the single code input that selects P_i, the output is (0,0).

4.6.11. A Break in a Product Term Line

Each product term line controls one OR array crosspoint device and is controlled by n AND array pull-down devices and one pull-up (or precharge) device. All the pull-down devices are connected to the "middle" of the line. The pull-up device and the OR array crosspoint device are either connected on opposite ends of the product term line (as shown in Fig. 4.1) or on the same end of the line.

If the product term line pull-up device and the OR array crosspoint device are on opposite ends of the line, any break in the product term line prevents the segment of the line connected to the OR array device from being pulled up. As a result, the product term line is either floating or stuck-at-0. If the line is floating, its value is constant and independent of the input. Hence, in any case, the product term line segment that controls the output line is either stuck-at-0 or stuck-at-1. Earlier in this section it is shown that a stuck-at fault on a product term line is detectable by some code input.

If the product term line pull-up device and the OR array crosspoint device are on the same end of the line, a break in the product term line disconnects some of the AND array crosspoint devices from the segment of the line connected to the OR array device. As a result, the product term line is selected when it is not supposed to be selected. Let P_i denote the product term line that is selected by the code input

$XX = (x_{n-1}, \ldots, x_h, \ldots, x_0, x'_{n-1}, \ldots, x'_h, \ldots, x'_0)$ in the fault-free circuit. A break in P_i disconnects some AND array crosspoint device, CA_{hi}, from the segment of P_i that controls the OR array device. Since CA_{hi} is controlled by input line a_h, in the fault-free circuit, P_i can only be selected if $a_h = 0$. Hence, $x_h = 0$. In the fault-free circuit, the code input $YY = (x_{n-1}, \ldots, x'_h, \ldots, x_0, x'_{n-1}, \ldots, x_h, \ldots, x'_0)$ selects the product term P_j where P_i and P_j are connected to different output lines. Since the crosspoint device CA_{hi} is disconnected from P_i in the faulty circuit, P_i is not affected by a_h and the code input YY selects *both* P_i and P_j. Hence, the output is a (0,0).

4.6.12. A Break in an Output Line

Each output line is controlled by 2^{n-1} OR array pull-down devices and one pull-up (or precharge) device. All the pull-down devices are connected to the "middle" of the line. The pull-up device and the output from the circuit are either on opposite ends of the output line (as shown in Fig. 4.1) or on the same end of the line.

If the output line pull-up device and the circuit output are on opposite ends of the line, any break in the output line prevents the segment of the line that serves as the output from the circuit from being pulled up. As a result the output line is either floating or stuck-at-0. If the line is floating, its value is constant and independent of the input. Hence, in any case, the segment of the line that serves as the circuit output is either stuck-at-0 or stuck-at-1. Earlier in this section it is shown that a stuck-at fault on an output line is detectable by some code input.

If the output line pull-up device and the circuit output are on the same end of the line, a break in the output line disconnects some of the OR array crosspoint devices from the segment of the line that is the output from the circuit. As a result, the output line is selected when it is not supposed to be selected. Let c_m denote the output line with a break. Let CO_{im} denote an OR array crosspoint device that is disconnected from the segment of c_m that serves as the circuit output. In the fault-free circuit, the product term line P_i, that controls CO_{im}, is selected by the code input XX. In the faulty circuit, due to the break, the crosspoint device CO_{im} cannot affect the output line c_m. For the code input XX, P_i is the only selected product term. Hence CO_{im} is the only OR array crosspoint device that is turned on. Therefore neither output line is pulled down and the circuit produces the noncode output (1,1).

4.7. Implementation and Application Considerations

In the previous three sections it was shown that, using a single two-level NOR-NOR PLA, it is possible to implement a comparator that is self-testing with respect to any single fault that is likely to occur in MOS VLSI circuits. This result is a necessary prerequisite for the use of duplication and comparison as the basic scheme for implementing error detection. However, two main problems remain to be discussed: First, the size of the comparator, implemented as a single PLA, grows exponentially with the number of bits in the two vectors to be compared. Second, it is necessary to ensure that all the code inputs will appear as inputs to the comparator often enough so that a complete self-test of the comparator will be performed before there is a chance for multiple faults to occur in the system.

In Section 4.3 it was shown that a self-testing comparator implemented as a single two-level NOR-NOR PLA, must have 2^n product term lines. If the output from each one of the duplicated functional modules is, say, 16 bits, this implementation is impractical since it requires $2^{16} = 65536$ product terms. Fortunately, efficient implementations of a self-testing two-rail code checker (comparator) for large input vectors can be achieved by using checkers for smaller input vectors as *cells* that are connected together in a tree structure (Fig. 4.3)[Khak82, Wake78].

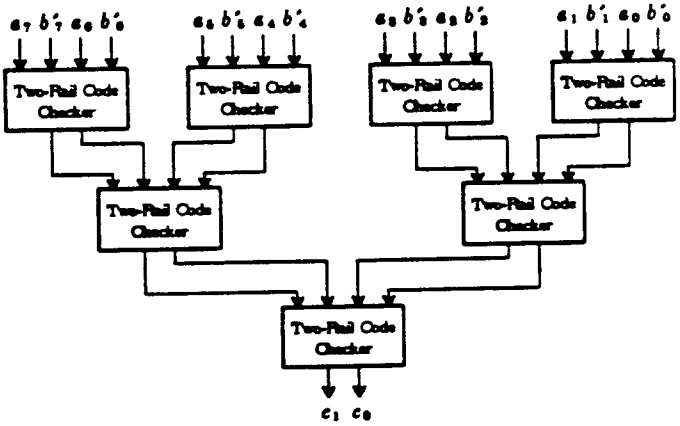


Fig. 4.3: A Self-Testing Two-Rail Code Checker Tree

Each cell is a self-testing comparator for relatively small bit vectors (two to six bits wide) which is implemented with a single two-level NOR-NOR PLA as outlined in Section 4.2. A complete tree with h levels of cells, where each cell is an m-bit comparator, can be used to compare m^h bits and contains $(m^h-1)/(m-1)$ cells. Hence, if the vectors to be compared are n bits wide, the number of levels in the tree is $\lceil \log_m n \rceil$

while the total number of cells in the tree is at most $(n-1)/(m-1)$. Thus the number of cells is (approximately) linearly related to n. Hence, tree-structured cellular implementations of self-testing comparators are practical for large input bit vectors.

In the cellular tree-structured implementation of the comparator, a noncode output from any one of the cells presents a noncode input to the cells at the next level. This forces the output from the entire tree to be noncode. Hence, the tree-structured implementation preserves the self-testing property of the cells.

If duplication and comparison is used for error detection, the first fault that occurs in the comparator must be detected before additional faults can occur in the comparator or in the functional modules. Thus, a set of code words that achieves a complete self-test of the comparator must appear as inputs to the comparator within a time interval that is significantly smaller than the mean time between failures for the two functional modules and the comparator together. Based on the results of sections 4.3 and 4.6, a complete self-test of a comparator implemented as a single NOR-NOR PLA requires all 2^n code words to appear at the inputs. If n is large, this requirement may imply that the complete self-test takes so much time that there is an unacceptably high probability that additional faults may occur in the comparator or functional modules before the self-test is completed. Fortunately, for the tree-structured cellular implementation, the number of code inputs required for a complete self-test is only 2^m, where m is the size of the bit vectors compared by each cell [Boss70, Khak82]. This efficient self-test is possible since, assuming that only one of the cells may be faulty, 2^m properly selected code inputs test all the cells in parallel. Thus, if the cells are 2-bit comparators, four code inputs are sufficient for a complete self-test of the entire tree.

Even with the relatively small number of code inputs needed for a complete self-test, it may still be difficult to satisfy the requirement that all code words appear as inputs to the comparator with some specified frequency. This would be particularly problematic if the duplicate functional modules were low-level passive circuits such as an ALU or an instruction decoder within a processor. Hence, for such low-level modules duplication and comparison is *inappropriate*. On the other hand, the technique is highly effective if the modules are high-level, "intelligent" subsystems, such as the computation nodes in a multicomputer system, which are interacting with similar high-level subsystems. In this case, a subsystem may periodically initiate action that causes it to generate all the

necessary patterns at its interface with other subsystems. The subsystem initiating the self-test of its comparator can inform the other subsystems that the next "message" is simply a test and should not be interpreted as "real work."

References

Boss70. D. C. Bossen, D. L. Ostapko, and A. M. Patel, "Optimum Test Patterns for Parity Networks," *AFIPS Proceedings* **37** pp. 63-68 (1970).

Cart68. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *IFIPS Proceedings*, Edinburgh, Scotland, pp. 878-883 (August 1968).

Cour81. B. Courtois, "Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits," pp. 341-350 in *VLSI 81*, ed. J. P. Gray, Academic Press (1981).

Gali80. J. Galiay, Y. Crouzet, and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers* **C-29**(6) pp. 527-531 (June 1980).

Khak82. J. Khakbaz and E. J. McCluskey, "Concurrent Error Detection and Testing for Large PLA's," *IEEE Journal of Solid-State Circuits* **SC-17**(2) pp. 386-394 (April 1982).

Mak82. G. P. Mak, J. A. Abraham, and E. S. Davidson, "The Design of PLAs with Concurrent Error Detection," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 303-310 (June 1982).

Osta79. D. L. Ostapko and S. J. Hong, "Fault Analysis and Test Generation for Programmable Logic Arrays," *IEEE Transactions on Computers* **C-28**(9) pp. 617-627 (September 1979).

Son81. K. Son and D. K. Pradhan, "Completely Self-Checking Checkers in PLAs," *1981 International Test Conference*, Philadelphia, PA, pp. 231-237 (October 1981).

Tami84. Y. Tamir and C. H. Séquin, "Design and Application of Self-Testing Comparators Implemented with MOS PLAs," *IEEE Transactions on Computers* **C-33**(6) pp. 493-506 (June 1984).

Wads78. R. L. Wadsack, "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *The Bell System Technical Journal* **57**(5) pp. 1449-1474 (May-June 1978).

Wake78. J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland (1978).

Wang79. S. L. Wang and A. Avizienis, "The Design of Totally Self Checking Circuits Using Programmable Logic Arrays," *9th Fault-Tolerant Computing Symposium*, Madison, WI, pp. 173-180 (June 1979).

Chapter Five
Error Recovery in Multicomputers

In Chapter 3 and Chapter 4 it was shown that a VLSI multicomputer can be implemented using self-checking nodes that ensure that there is a very high probability of detecting any error caused by node failure. As discussed in Chapter 3, errors caused by faults in the communication links can be detected using error-detecting codes. However, detecting an error is only the first step towards fault tolerance, i.e., the first step of any technique that allows the system to continue correct operation despite a hardware fault. When a component fails, the part of the system state that is stored in that component may become inaccessible to the rest of the system. Thus, even if the fault-free components of the system never accept the erroneous output of failed components, it may be impossible to restore a valid system state from which normal operation can be resumed. The recovery of a valid system state following component failure is discussed in this chapter.

Section 5.1 presents some of the basic concepts and techniques for error recovery. Section 5.2 is a brief survey of the current state of the art in error recovery techniques for multiprocessors and multicomputers. Section 5.3 presents a new technique for error recovery in multicomputers. This technique involves periodically saving the entire system state and restoring a previously saved state when an error is detected. The section includes algorithms for checkpointing the entire system state, distributing diagnostic information, and using the checkpointed state for error recovery. An informal "proof" that the algorithms are correct is presented in Section 5.4. An estimate of the overhead required by this scheme is given in Section 5.5. Section 5.6 discusses how the scheme can be expanded to allow interactions with the "outside world," deal more effectively with transient faults, reduce the latency in detecting errors in communication links, and handle the failure of disks and the nodes that control them.

5.1. Basic Concepts and Techniques

Error recovery is the process of transforming an erroneous system state, which may lead to system failure, into a valid system state, that guarantees correct system operation as long as all the system components continue to operate correctly. Most schemes for

performing error recovery can be classified as *forward error recovery* schemes or *backward error recovery* schemes [Rand78]. Forward error recovery techniques attempt to modify an erroneous system state so that it becomes a valid state. Backward error recovery techniques involve resetting (backing up) the system to a previous valid state rather than trying to modify the current state.

Forward error recovery techniques are usually designed as an integral part of the system they serve. By their very nature, they are only useful for recovering from *anticipated* errors, *i.e.*, the designer anticipates that a particular erroneous state may occur and provides a specific technique for transforming that state into a valid state.

Forward error recovery is often used in systems where there are strict real-time constraints. For example, the main controller of an unmanned aircraft my operate by continuously reading several sensors and sending commands to various actuators. If one of the sensors fails and sends a reading that does not pass the *acceptance test* [Rand78], the controller does not use this erroneous reading for computing the next set of commands to the actuators. Instead, the controller recovers from the error by replacing the erroneous value with some "guess" of a reasonable value that is unlikely to have any disastrous consequences.

If the use of fault tolerance techniques in the system has to be taken into account by the application programmer, the programming task becomes more complicated, time-consuming, and error-prone. Hence, fault-tolerant systems in general and, particularly, those that are intended for more-or-less general-purpose use, attempt to "hide" their use of fault tolerance techniques from the application programmer. Since forward error recovery techniques are usually dependent on a particular application and can handle only anticipated errors, such techniques will not be discussed any further in this thesis.

Backward error recovery techniques can cope with *unanticipated* errors. The state of the system is periodically recorded. When an erroneous state is detected, it is abandoned and the system is reset to this previously recorded error-free state, called a *recovery point* or a *checkpoint*. The process of creating a recovery point is called *checkpointing*.

The main advantage of backward error recovery techniques is their ability to handle unanticipated errors. No matter what type of error occurs, as long as it can be detected, some valid system state can be reinstated. Hence, a backward error recovery scheme can

be totally independent of the application.

The main disadvantage of backward error recovery techniques is the overhead of establishing and maintaining the recovery points. In a uniprocessor the recovery point includes the contents of memory and other storage devices as well as the contents of all the processor registers.

There are two basic approaches to maintaining the information necessary for backward error recovery: (1) Maintaining, at all times, multiple up-to-date copies of the entire system state [Kast83]. (2) Maintaining information that allows the restoration of a valid system state by redoing some computations that were already performed by the system [Bari83, Borg83]. The first approach allows nearly instantaneous error recovery without any loss of work. The system is restored to its state that immediately preceded the occurrence of the error. This scheme requires the duplication of all system resources just for error recovery, in addition to any redundancy used for error detection or system reconfiguration.

The second approach above requires periodic saving of the entire system state. When an error is detected, the system is restored to a previous state and the computations that were performed since that state are redone. This scheme involves overhead in both time and storage. The time overhead results from the periodical creation of the recovery points as well as from the computation that has to be redone when an error occurs. The storage overhead is the extra storage required to save the recovery points. The frequency of generating recovery points is an important parameter for minimizing the time overhead. Too much time may be spent generating recovery points if the frequency is "too high." If the frequency is "too low," too much time will be lost (on the average) redoing computations following an error.

The simplest way to generate a recovery point is to save the entire state of the system, i.e., the contents of all registers, memory, and secondary storage. An alternative technique is to usually save only the *changes* in the system state since the last recovery point. Periodically, the entire state is saved since, at some point, the "history" of changes can take up more space than the entire state and/or generating the entire state from some original state and a sequence of changes would take too much time.

In a uniprocessor system the recovery technique discussed in the previous paragraph can be facilitated by a device called a *recovery cache* [Ande76, Lee78, Lee80, Rand75]. In

order to establish a recovery point, the contents of the CPU's registers are saved. The contents of memory are *not* saved. During normal operation, whenever the processor modifies the contents of a memory location, the old contents are saved in the recovery cache. The state at the recovery point can be reestablished at a later time by restoring the contents of memory from the recovery cache and the contents of the registers from where they were saved when the recovery point was established.

The recovery cache technique has been used for recovering from errors caused by transient hardware faults in a microprocessor chip. Kubiak et al designed a VLSI recovery cache chip, called Penelope, that is connected to the processor/memory bus of a microprocessor system [Kubi82]. Penelope maintains the previous content of modified memory locations. Whenever Penelope's "save stack" becomes full, a new recovery point is established by saving the contents of the processor's registers and reinitializing the "save stack" to an "empty" state. Initial measurements of Penelope's performance with a save stack of 256 bytes show that the performance penalty caused by Penelope, when compared with an equivalent system with no provisions for error recovery, is less than 10%.

A recovery point may be generated by periodically "freezing" the entire system and saving this frozen state (or information necessary to generate this state). When error recovery occurs, the restored system state is a state that actually existed in the past. An alternative is to store different parts of the system state independently. For example, in a uniprocessor system that is executing several processes, the states of the different processes may be checkpointed at different times. If there is any interaction between the processes, when recovery occurs, care must be taken to ensure that the recovery points to which the different processes are restored are *consistent* with each other (i.e., that the ordered set of the recovered external states of all the processes constitute a valid system state). However, since the processes do not run in lock-step, communicating at each step, the state of one process, after executing t_i time units, may be consistent with the state of a second process that has executed anywhere between t_j and t_k time units. Thus the restored system state is not necessarily a state that actually existed in the past. Rather, it is a state that *could have* existed in the past and will result in correct system output.

The error recovery techniques discussed so far all require some special action following error detection in order to recover a valid system state. *Masking*

redundancy [Rand78] or *error masking* techniques involve always performing some action that hides the effects of a certain class of errors. These actions are performed regardless of whether or not an error has actually occurred. A canonical example of error masking is Triple Modular Redundancy (TMR) [Plat80, Siew78, Wake76]. In this approach a system consists of three identical subsystems and a majority voting circuit. The output of the system is the majority vote of the outputs of the three subsystems which are executing identical tasks. An error caused by the failure of one of the subsystems is masked since the majority vote is the correct output of the two fault-free subsystems. A relatively high overhead during normal operation is always associated with error-masking techniques. However, when an error occurs, the system continues to operate normally and error recovery occurs "automatically."

5.2. Error Recovery Techniques for Multicomputers

A multicomputer system consists of several more-or-less independent components (the nodes) that interact with each other asynchronously. This makes the coordination of any joint task, including error recovery, difficult and prone to subtle "bugs." Error masking and backward error recovery techniques for multicomputers are discussed in this section.

In the previous section, TMR with hardware voting circuits was mentioned as an example of error masking. The same idea can be used in a multicomputer in which each task is executed on three (or more) different nodes and the results are transmitted to other nodes via independent communication paths. A node receiving the results can take a bit-by-bit majority vote, thereby masking the erroneous output from one of the nodes or the corruption of one of the outputs during transmission. This technique has been used in the SIFT multiprocessor system that was designed to serve as the main controller of an aerodynamically unstable airplane [Wens78]. The performance requirements from SIFT are rather modest since it reads and controls mechanical devices that change relatively slowly. Due to the nature of the application of SIFT, there must not be any sudden break in its operation, i.e., error recovery cannot involve temporarily stopping normal processing. This type of TMR error masking is particularly well suited to the combination of modest performance requirements and strict constraints on the operation of the system following an error.

There are many similarities between the error recovery technique described in the

previous paragraph and the error detection scheme based on system-controlled node-level duplication and comparison discussed in Chapter 3. In Chapter 3 this error detection scheme was shown to be unsuitable for a multicomputer used for general purpose applications. For similar reasons, the error recovery scheme discussed in the previous paragraph is also not suitable for such a system. Specifically, the shortcomings of this technique are: (1) it dedicates two thirds of the hardware for error recovery, (2) it increases interprocessor communication delays, (3) it is not, by itself, sufficient for locating faulty components, (4) it does not adequately handle erroneous routing of packets by intermediate nodes on a communication path, and (5) it poses severe requirements on the routing and task assignment algorithms used in the system.

Over the last five years, a great deal of research has been devoted to the development of algorithms that enable all the working nodes in a multicomputer to reach a unanimous decision despite the failure of some nodes and links [Dole81, Peas80, Stro83]. Specifically, if one node broadcasts a packet, the problem is to ensure that all the other working nodes agree on the content of that packet or agree that the sender is faulty since the packets it sent to its immediate neighbors were not all equal. When these algorithms are used, the faulty (even malicious) behavior of links and nodes are masked. Hence, these are error-masking algorithms. In the literature they are called algorithms for reaching *Byzantine Agreement*.

Algorithms for reaching Byzantine Agreement are extremely useful for the very specialized task of ensuring that all the nodes in a system reach a consistent decision. However, they are not useful for masking errors in transmission between pairs of nodes unless every message in the system is broadcast. Hence, these algorithms are not directly applicable to general-purpose computations performed on a multicomputer and will not be discussed any further in this thesis.

As mentioned in the previous section, one technique for backward error recovery involves maintaining, at all times, multiple up-to-date copies of the entire system state [John84, Kast83]. In a multicomputer, one way to maintain a copy of the system state is for the state of each node (*primary node*) to be maintained on some other node (*backup node*) [Bari83]. The primary node and the backup node contain the same code and data, and execute this code at approximately the same time. If an error occurs as a result of a transient fault in a node, the node may be reset and its state restored to what

it was immediately prior to the fault using the information in the backup node. Since an error may be a result of a permanent fault, the backup node must be able to take over the task of a failed primary node.

In a multicomputer the system state is the ordered set of the external states of all the nodes. For the rest of the system, the external state of a node is defined by the set of packets that it has already received and the set of packets that it has already sent. Keeping the backup node "completely up-to-date" with the primary node requires that the external states of these nodes be kept identical. This can be accomplished by sending each packet to the destination node and the destination backup and notifying the sender's backup that the packet has been sent. Since this entire operation must be performed *atomically* (i.e. it must either be completed or aborted but never partially completed), a two-phase-commit [Gray78] must be performed for *each* packet (or message). The disadvantages of this technique in terms of overhead and restriction of system operation are obvious and similar to the problems with the the error detection scheme based on system-controlled node-level duplication and comparison that was discussed in Chapter 3. This technique is therefore not suitable for a multicomputer executing general-purpose applications.

It should be noted that the above backward error recovery technique is used in several multiprocessor and multicomputer systems in which all interprocessor communication is over a common bus or Ethernet [John84, Kast83]. Since all the "nodes" can easily monitor all interprocessor communication, it is possible to implement an efficient *atomic* (indivisible) operation that transmits a message to a primary node and its backup [Borg83].

Since neither error masking nor backward error recovery with up-to-date backups are well-suited for multicomputers used for general-purpose applications, backward error recovery in which some computations have to be redone as part of the recovery process appears to be the best technique for such systems. As mentioned in the previous section, a recovery point may be generated by periodically freezing and saving the entire system state. The main disadvantage of this technique is that it requires normal computation throughout the system to stop for the duration of the checkpointing process. An alternative technique that appears more attractive is for the nodes to establish recovery points independently and to attempt to restore a consistent system state from the

individual recovery points during error recovery.

Establishing independent recovery points for the different nodes poses the problem that restoring a previous state of one node may require that other nodes be backed up to previous states. In the following situation, for example, restoring a previous state of a node P_A requires an interacting node P_B to restore its state: At time t_1 node P_B establishes a recovery point. At some later time $t_2 > t_1$ node P_A establishes a recovery point. At time $t_3 > t_2$ P_A sends a message to P_B. The message causes P_B to change its state (e.g. modify a memory location) and send a message back to P_A at time $t_4 > t_3$. At time $t_5 > t_4$ an error in P_A is detected and requires the state of P_A to be restored to the state saved at time t_2. Since the state of P_A is restored, it will send to P_B a message identical to the one sent at time t_3. For the computation to proceed correctly, the message returned to P_A by P_B should be identical to the message returned at time t_4. If the state of P_B is not restored, the message it returns may be different leading to failure of the computation. Hence the state of P_B should be restore to its value before t_3. Since a recovery point for P_B was last established at time t_1, the state of P_B must be restored to its value then.

If, in the above example, P_A and P_B also interacted between time t_1 and t_2, then, restoring P_B to its state at time t_1 would, in turn, require that the state of P_A be restored to some recovery point established prior to t_1. In fact, it is possible that a single error in one node may result in an uncontrolled *domino effect* [Rand75], requiring that all the nodes in the system back up all the way to system initialization.

Wood [Wood81] has developed a scheme for keeping track of the recovery actions that must occur in all the nodes in the system if a particular node is rolled back. Each node is required to maintain several recovery points, starting from an initial state. Maintaining the multiple recovery points as well as all the information necessary to ensure consistent recovery poses significant overhead both in time and storage. Furthermore, there is always the possibility that the entire system will have to be rolled back to its initialization due to the domino effect. In order to increase the efficiency of the recovery scheme, either some restrictions have to be placed on the actions of the nodes or the nodes must no longer checkpoint their state completely independently but rather must somehow coordinate when and how to create checkpoints.

Barigazzi and Strigini [Bari83] propose an error recovery scheme for multicomputers

that does not require storing multiple recovery points. Like most other schemes, the states of individual processes are checkpointed and recovered rather than the complete state of nodes. The scheme involves periodic saving of the state of each process by storing it both on the node where it is executing and on another backup node. All interacting processes are checkpointed together, so that their checkpointed states are guaranteed to be consistent with each other. Thus, the domino effect is avoided.

The recovery scheme described in [Bari83] is well suited for applications that must satisfy strict real-time constraints. However, it results in significant performance degradation. For each process, a complete backup is maintained both on the node executing the process as well as on another node. Thus, a large percentage of the memory cannot be used by active processes. The resulting increase in paging activity also reduces performance by increasing the average memory access time and the load on the communication links.

Another difficulty with the recovery scheme described in [Bari83] is that it requires the "send" and "receive" operations to be atomic. In order to accomplish this, the use of a two-phase commit protocol [Gray78] is proposed. Such a protocol requires explicit acknowledgement for each message and implies that after a "send" operation, the sending process is not able to continue executing until an acknowledgement is received. This restriction on process "behavior" and the associated increase in message traffic leads to reduced performance relative to an identical system where no error recovery is implemented.

The idea of simultaneously checkpointing the state of all processes belonging to the same "task" [Jone79] can be taken a step further: simultaneous checkpointing of the complete state of all the user and system processes on the system, i.e., simultaneous checkpointing the complete state of all the nodes in the system. Creating and saving such a global checkpoint is expensive since it requires moving large blocks of data through the system and then storing them "reliably." However, if the time between checkpoints is sufficiently large compared with the time it takes to establish a new checkpoint, the net system overhead for error recovery is relatively small. In a large multicomputer the expected time to establish a new checkpoint is on the order of one minute (see Section 5.5). Thus, keeping the overhead low requires that a new checkpoint be established only once or twice an hour. It is clear that the loss of as much as an hour of

processing when an error is detected is tolerable only for non-interactive applications.

The rest of this chapter presents an error recovery scheme that is based on periodically checkpointing the entire system state. The global checkpoints are stored on disk so that all of local memory can be used for active processes. It is shown that if global checkpoints are used for error recovery, it is possible to avoid any restriction on the behavior of processes and to eliminate the need for message acknowledgement. Furthermore, there is no need to use up valuable communication bandwidth by encoding the messages in some error-detecting code (see Subsection 5.3.2).

5.3. Implementing Error Recovery Using Global Checkpoints

The basic idea is for some designated node to periodically initiate and coordinate the creation of a new global checkpoint. When any node detects an error, it initiates the distribution of diagnostic information throughout the system. All the nodes are then set to a consistent system state using the last global checkpoint. Finally, normal operation is resumed.

In the following six subsections we describe in detail the creation and storing of a global checkpoint and its use for recovery. In Subsection 5.3.1 we present some basic assumptions that are made about the system. In Subsection 5.3.2 we differentiate between *normal* packets that are used for the application tasks and *fail-safe* packets that are used to coordinate the creation of checkpoints and the recovery from errors. Subsection 5.3.3 contains a brief description of the eight types of fail-safe packets used by the system. At each point in time, a node may be engaged in normal computation, creation of a checkpoint, or recovery from an error. A description of the possible modes or *logical states* of a node is presented in Subsection 5.3.4. Subsection 5.3.5 describes how a consistent global checkpoint is established and stored on disk. Finally, in Subsection 5.3.6, we show how the global checkpoints can be used to recover from errors.

5.3.1. Assumptions

We will begin by introducing several simplifying assumptions that will be used in the algorithms for establishing global checkpoints and for using those checkpoints for error recovery. We will later discuss how some of these assumptions can be relaxed.

We assume a closed system that consists of nodes, links, and disks (or some other form of mass storage). All "input" is stored on disk before operation begins. All

"output" is stored on disk when the job ends (Fig. 5.1).

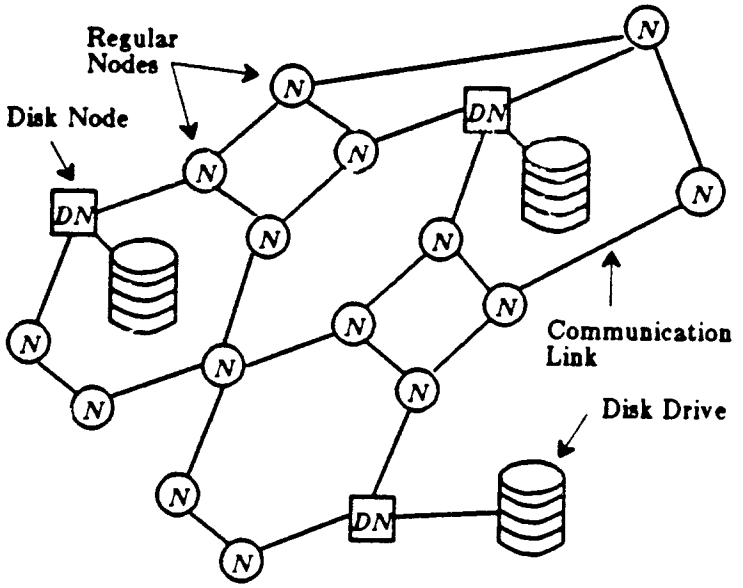


Fig. 5.1: A Multicomputer

As previously mentioned, the nodes are self-checking and are guaranteed to signal an error to their neighbors immediately when they send incorrect output [Tami83]. As a first step, any node that generates an error signal is assumed to be permanently faulty and no attempt is made to continue to use it.

Hardware faults either cause a node to generate an error signal or cause an error in transmission. It is assumed that a fault can occur at any time, including during the creation of a checkpoint. However, if a second fault occurs during recovery from a previous fault, the system stops execution and does not attempt recovery. This and other situations where the system must stop due to an unrecoverable state will henceforth be called a *crash*. It should be noted that, even if a crash occurs, the system still does not generate incorrect results. Furthermore, since recovery takes only a few minutes and the system has an MTBF of tens of hours, the probability of a fault occurring during recovery is very small.

Since the disks are extensively used for paging, checkpointing, and I/O, the average number of "hops" from each node to the nearest disk should be made small. Hence, disks are connected to several nodes throughout the system. As a first step, we allow an error in disk I/O or a fault in a node that controls a disk (henceforth called a *disk node*) to cause a crash. Each disk node uses an error-detecting code for all data written on the

disk. When the node reads from the disk, any error caused by a faulty disk or a fault on the path between the node and the disk is detected by using the code. If an error is detected, the disk node signals a crash.

The structure of the system is relatively stable — it changes only due to hardware faults. Since all the nodes in the system are informed of each fault, every node is able to maintain tables that reflect the structure of the operational part of the system. This includes information about which nodes and links are operational and which nodes are disk nodes.

Each node has a unique identifier and there is a total ordering of these identifiers (used to establish successorship for the designated node that initiates periodic checkpointing). All the nodes in the system know the identifiers of all the other nodes. For simplicity, we assume that the identifiers of an n node system are the integers 1 through n.

In order to send a message, a process assembles the message in memory and executes a system call. The kernel may divide the message into *packets* which are the unit of information actually transmitted. Packets may arrive at their destination out of order (if they arrive at all). We assume that it is the responsibility of the kernel of the receiving node to put the packets in order before they are made available to the receiving process.

The interconnection topology of the system is of crucial importance for achieving fault tolerance. A large body of research on the tradeoffs between various topologies is available [Witt81, Prad82] and will not be discussed in this paper. One parameter that is especially important for fault tolerance is the *connectivity* of the network. The node/edge connectivity is the minimum number of nodes/edges whose failure partitions the network so that there is at least one pair of working nodes that can no longer communicate. We assume that the connectivity of our system is sufficiently large that there is a very low probability of partitioning. Hence, it is acceptable if partitioning causes a crash.

If a node or a link fails, routing of packets through the network has to be modified to use alternate paths. This process of *reconfiguration* requires updating routing tables throughout the network. We assume that one of the well-known reconfiguration procedures [Taji77, Bozy82] is used in conjunction with our recovery scheme but do not discuss this problem any further in this paper.

5.3.2. Normal Packets and Fail-Safe Packets

As previously mentioned, one of the main advantages of our error detection and error recovery schemes is that it does not require the substantial delays in normal inter-processor communication that are a necessary part of most other such schemes. In particular, during normal operation (*i.e* not in the process of creating a checkpoint or recovering from an error), no redundant bits for error detection are transmitted with the messages or packets, no acknowledgement of messages or packets are transmitted by their recipients, and neither processes nor processors have to wait for acknowledgement of messages or packets.

Since errors can occur in transmission, there must be some provision for detecting errors in messages. However, since the probability of an error in transmission is low, it is wasteful to check the validity of each message or packet independently. Instead, in each node each port has two special purpose registers for error detection. One of these registers contains the CRC (Cyclic Redundancy Check) check bits for all the packets that have been sent from the port. The other register contains the CRC check bits of all the packets received. Initially these special purpose registers are initialized to some known value. By making these registers linear feedback shift registers (LFSRs) the contents of the register can be updated in parallel with the actual transmission of each packet [Elki82].

In order to check the validity of all the packets transmitted through a particular link, each node sends to its neighbor the contents of the LFSR used for outgoing packets. The neighbor can then compare the value it receives with the value in its LFSR for *incoming* packets and signal an error if it finds a mismatch. The normal procedure used to recover from a node failure is then initiated.

The validity of all packet transmissions must be checked immediately prior to the creation of a checkpoint. If this is not done, the state of a node corrupted by an erroneous message may be checkpointed and later used for recovery. The procedure for creating a checkpoint must therefore include checking all the links before committing to the new checkpoint.

The above procedure is not appropriate for the packets used to coordinate the creation of checkpoints and for error recovery. In this case the information in the packet must be verified before it is used. Hence, for these packets, an error detecting code such as CRC is used and redundant bits must be transmitted with the packet. Thus, there are

two types of packets in the system: the normal packets that do not include any information for error detection and special control packets that are used only for transmitting information between kernels and that include a sufficient number of redundant bits to recognize any likely error in transmission. These special packets are called *fail-safe* packets since they are either error-free or the error is easily detectable by the receiving node.

As discussed in Subsection 5.6.3, it is possible to speed up the detection of errors caused by faulty links if some redundant bits are transmitted with each normal packet. However, even if this is done, the normal packets can still be handled more efficiently than the fail-safe packets. In particular, the latency associated with forwarding a normal packet through a node can be significantly reduced if the node can begin forwarding the packet before all of it has arrived [Séqu83]. This is not possible for a fail-safe packet since a node receiving such a packet must verify that it is correct before forwarding it. A node receiving a normal packet may begin forwarding it immediately and initiate error recovery if, after the complete packet is received, it is found to be invalid.

The first bit of each packet is used to distinguish between a normal packet (0) and a fail-safe packet (1). If the bit is 0, the packet is usually accepted and processed or forwarded regardless of whether it is correct or not. The LFSR for incoming packets is updated as the packet is received. If the node is in the middle of making a checkpoint or recovering from an error, it may expect to receive only fail-safe packets. In this case, if the first bit of the packet is 0, the node signals an error. If the first bit of the packet is 1, the packet is not accepted until it is checked. If an error in the packet is found, the node signals an error. The two LFSRs in each port are not modified by incoming or outgoing fail-safe packets.

It should be noted that the above scheme works in the case where a fault on the link modifies the first bit of the packet. If the original packet is a normal packet, the fault causes it to become a fail-safe packet. The receiving node checks the packet, assuming it is coded using some error-detecting code, and finds an error. If the original packet is a fail-safe packet, the fault causes it to become a normal packet. The LFSR for incoming packets in the receiver node is modified. The error is detected when the two nodes compare the value of the sender's LFSR for outgoing packets with the value of the receiver's LFSR for incoming packets.

5.3.3. Types of Fail-Safe Packets

Any two nodes i and j are *neighbors* if, and only if, there is a link between them.

For every node j that is a neighbor of node i, $CKV(i,j)$ is the correct CRC check vector of all the normal packets sent by i to j since the last checkpoint was made. At any point in time $CKV_i(i,j)$ is the value of $CKV(i,j)$ generated and stored in the output LFSR in node i. $CKV_j(i,j)$ is the value of $CKV(i,j)$ generated and stored in the input LFSR in node j.

There are eight types of fail-safe packets:

checkp(CKV)

Used to initiate the creation of a new checkpoint. When sent by some node i to its neighbor node j it contains $CKV_i(i,j)$.

state(dest,node,seq,size)

Used to transmit the state of node *node* to node *dest*. The state is transmitted in fixed length packets. The *size* field contains the number of these packets required to transmit the entire state. Each packet includes a sequence number *seq*. These packets are used to transmit the state of a node to a disk node during checkpointing and to transmit the state of a node from a disk node during recovery.

saved(coord,node)

Used by a disk node to inform the checkpointing coordinator *coord* that the disk node is prepared to commit to a new state for node *node*.

resume

Used to signal the end of a checkpointing "session" or the end of a recovery session.

fault(type,location,source)

Used to broadcast the fact that a fault has occurred and to initiate recovery. The field *type* contains the type of fault detected: *node*, *link*, or *unknown*. The faulty node or link is indicated by *location*. The node that detected the error and initiates the distribution of diagnostic information is indicated by *source*.

recover(version)

Used to let the disk nodes know which version of the node states stored on their disks they should recover. *Version* may be 0 or 1.

restored(coord,node)

Used by the node *node* to inform the current checkpointing coordinator that *node* has received its complete state (as part of the recovery process) and is ready to resume normal operation.

crash(type,location,source)

Used to broadcast the fact that an unrecoverable situation has been encountered. The arguments are the same as those for the *fault* packet.

5.3.4. The Logical States of a Node

At any point in time, a node in the system may be engaged in normal operation, checkpointing, distribution of diagnostic information, or error recovery. The node's response to various packet types depends on its current activity. Hence, we can define several *logical-states* (henceforth *l-states*)† that are simply labels for the current activity of the node:

normal

This is the l-state of the node during normal operation. Normal packets are accepted and processed. The fail-safe packets *checkp* and *resume* may be received. A *checkp* packet causes an l-state transition to *checkp-begin*. A *resume* packet is ignored.

checkp-begin

This is the l-state of the node after it has received the first *checkp* packet from one of its neighbors but before it receives a *checkp* packet from all of its other neighbors. The checkpointing coordinator enters this l-state when it initiates checkpointing. Normal packets may be received only from neighbors that have not yet sent a *checkp* packet. Normal packets from other neighbors cause a transition to the *error* l-state. The arrival of valid *checkp* packets from all the neighbors causes an l-state transition to *checkpointing*.

checkpointing

This is the l-state of the node after it has received *checkp* packets from all its neighbors but before it completes sending its state to a disk node. No normal packet

† The l-state of a node is not to be confused with the node's "state" that is the contents of the node's memory that defines the state of all the processes and packets currently on the node.

should be received while in this l-state. If a normal packet is received, it causes a transition to the *error* l-state. The fail-safe packets *state* and *saved* may be received. When the node completes sending its entire state to a disk node, it changes its l-state to *checkpointed*.

checkpointed

This is the l-state of the node after it has completed sending its state to one of the disk nodes but before it receives the *resume* packet. If a normal packet is received by the node while in this l-state, it causes a transition to the *error* l-state. The fail-safe packets *resume*, *state*, and *saved* may be received. A *resume* packet causes an l-state transition to *normal*.

error

This is the l-state of the node after it has detected (or has been informed of) an error but before it is ready to accept its recovered state. The node enters this l-state when it receives a mismatch signal from a neighbor, receives an invalid fail-safe packet, receives a normal packet when only fail-safe packets are expected, or receives a *fault* packet. In addition, a transition to the *error* l-state may be caused by a valid fail-safe packet whose contents indicate some error condition (see next subsection). Normal packets are ignored if sent by neighbors that have not yet sent a *fault* packet. Other normal packets cause a transition to the *crashed* l-state. The fail-safe packets *checkp*, *saved*, *resume*, *state*, and *restored* are ignored if sent by neighbors that have not yet sent a *fault* packet. The *fault* packet is ignored if the *location* it refers to is the same as the location of the fault that caused the transition to the *error* l-state. Any other *fault* packet causes a transition to the *crashed* l-state. The fail-safe packet *recover* causes a transition the the *recovering* l-state.

recovering

This is the l-state of the node after it has received the *recover* packet but before it is ready to resume normal operation with its recovered state. If a normal packet is received, it causes a transition to the *crashed* l-state. The fail-safe packet *recover* is ignored. The fail-safe packets *state* and *restored* are processed (see Subsection 5.4.6). The arrival of the node's complete state via *state* packets causes a transition to the *recovered* l-state.

recovered

This is the l-state of the node after it has received its complete recovered state but before it actually resumes normal operation. If a normal packet is received, it causes a transition to the *crashed* l-state. The fail-safe packets *state* and *restored* are processed (see Subsection 5.3.6). All *recover* packets are ignored. The *resume* packet causes a transition to the *normal* l-state.

crashed .

This is the l-state of the node after an unrecoverable error has been detected.

Each node also includes the "state variable" *version* that determines what is the most recent *valid* version of the node's state that is stored on disk. This variable may have the values 0, 1, or *unknown*. When the system is initialized, the value of *version* in all the nodes is set to 0.

5.3.5. Creating a Global Checkpoint

Initially, a designated node, typically node 1, is assigned the task of serving as the coordinator for establishing global checkpoints. If the coordinator fails, all the other nodes in the system are notified and the next node, according to the total ordering between the nodes, takes over the task of being checkpointing coordinator.

Every node includes a "timer" that can interrupt the node periodically. Checkpointing is initiated by the checkpointing coordinator when it is interrupted by its timer [Bari83] and it is in the *normal* l-state. Checkpointing is also initiated when a task is complete so that the system can commit to the output stored on disk.

It should be noted that faulty operation of the timer is detected just like faulty operation of any other part of a node. As previously discussed, the self-checking node is implemented using duplication and comparison [Tami83]. Each duplicate module includes its own independent timer. Even if a fault disables one of the timers, the other timer still operates and causes the module it is part of to "behave" differently from the module with the faulty timer. As a result, the two modules produce different outputs, and an error signal is generated by the comparator that constantly monitors those outputs.

(I) The Actions of the Checkpointing Coordinator

When the checkpointing coordinator, say node i, initiates checkpointing, it does the following:

[1] Node i stops all work on application processes and stops transmitting normal packets. The node's l-state is changed to *checkp-begin*.

[2] Node i sends to every neighbor node j the *fail-safe* packet $checkp(CKV_i(i,j))$

[3] Node i waits for *checkp* packets from all its neighbors.

If a normal packet arrives, it is included with the rest of the node state that must be checkpointed.

If a $checkp(CKV_j(j,i)$ packet arrives from neighbor j then: If $CKV_j(j,i) \neq CKV_i(j,i)$, i changes its l-state to *error* and sends the packet $fault(link,(i,j),i)$ to all its neighbors.

If no *checkp* packet arrives from a neighbor j within some fixed time limit, i changes its l-state to *error* and sends the packet $fault(unknown,j,i)$ to all its neighbors.

[4] Node i changes l-state to *checkpointing* and sends its complete state to the disk node assigned to it.

[5] Node i changes l-state to *checkpointed* and waits for fail-safe packets of the type $saved(i,j)$ for all nodes j in the system that are known to be working.

If for one of the nodes, say node j, no such packet arrives within some fixed period of time, i changes its l-state to *error* and sends the packet $fault(unknown,0,i)$ to all its neighbors.

[6] After all the expected *saved* packets arrive, node i complements its *version* variable, changes l-state to *normal*, and sends the packet *resume* to all its neighbors.

[7] Node i resumes normal operation.

(II) The Actions of a Checkpointing Participant

Every node j, that receives the packet $checkp(CKV_i(i,j))$ from its neighbor i while in its *normal* l-state, does the following:

[1] Node j stops all work on application processes and stops transmitting normal packets. The node's l-state is changed to *checkp-begin*.

[2] If $CKV_i(i,j) \neq CKV_j(i,j)$, node j changes its l-state to *error* and sends the packet $fault(link,(i,j),j)$ to all its neighbors.

[3] For every neighbor node k (including $k = i$), node j sends to node k the *fail-safe* packet $checkp(CKV_j(j,k))$.

[4] Node j waits for *checkp* packets from all its neighbors except i.

If a normal packet arrives, it is included with the rest of the node state that must be checkpointed.

If a *checkp*$(CKV_k(k,j)$ packet arrives from neighbor k then: If $CKV_k(k,j) \neq CKV_j(k,j)$, j changes its l-state to *error* and sends the packet *fault*$(link,(k,j),j)$ to all its neighbors.

If no *checkp* packet arrives from some neighbor k $(k \neq i)$ within some fixed period of time, j changes its l-state to *error* and sends to all its neighbors the packet *fault*$(unknown,k,j)$.

[5] Node j changes its l-state to *checkpointing*, and begins to send its state to the disk node assigned to it using *state* packets. The complete node state except for one last *state* packet is sent.

[6] Node j changes its l-state to *checkpointed*, sets its *old-version* variable to *version*, sets its *version* variable to *unknown*, and sends the last packet containing its state to the disk node assigned to it.

[7] Node j waits for a *resume* packet from one of its neighbors.

[8] When node j receives a *resume* packet from its neighbor i, it sets its *version* variable to $1 -$ *old-version*, changes its l-state to *normal*, and sends a *resume* packet to all of its *other* neighbors.

[9] Node j resumes normal operation.

(III) The Actions of a Disk Node

A disk node may be a checkpointing coordinator or a checkpointing participant. In either case, it executes most of the protocols (I) or (II) as a regular node. However, a disk node also performs two additional tasks: (1) It stores node states so that they can be recovered in case of an error. (2) It handles input/output.

During a checkpointing session, a disk node accepts *state* packets and stores them on its disk. Once the complete state of some node j is received and stored on disk, the disk node sends a *saved*$(coord,j)$ packet to the checkpointing coordinator. Once a *resume* packet is received, the disk node commits to the most recently saved versions of the node states. The version of these node states is the current value of the *version* variable of the disk node.

In order to roll back the entire system to a previous state, it must be possible to roll

back the state of the disks as well as the state of the nodes. Hence, all files that are opened with write or read/write permission are duplicated by the disk node and I/O operations are performed on the duplicates until the disk node commits to the next checkpoint. Newly created files exist only as "duplicates" until the disk node commits to the next checkpoint. When the disk node commits to a new checkpoint, all the new and duplicate files are incorporated with the committed state. Files that remain open with write or read/write permission must, once again, be duplicated since all operations until the *next* checkpoint must not affect the files that are part of the current checkpoint.

5.3.6. Fault Handling

When a node detects an error, it informs its neighbors of the error. This diagnostic information is then distributed throughout the system. Since an error may be detected in the middle of creating a new checkpoint, some of the disk nodes may have access to subsets of two different system states: the state currently being checkpointed or the previous state. The first node that is informed of the error and which has the information needed to determine which version of the state should be used, distributes this information throughout the system. Once the disk nodes find out which version of the system state they should use, they begin sending the state to all the nodes in the system. When all the nodes receive their state, they inform the checkpointing coordinator, which subsequently initiates the resumption of normal operation. Since all working nodes in the system are informed of the cause of the error, these nodes are able to avoid using the failed node or link so that it cannot be the source of any additional errors.

The rest of this subsection is devoted to a detailed description of the actions of the nodes when an error is detected.

(I) The Actions of a Regular Node

When a node is in any l-state except *error* or *crashed*, it may change to the *error* l-state at any time, as described in Subsection 5.3.4. When a node j enters the *error* l-state, it does the following:

[1] Node j stops all work on application processes and deletes all normal *and* fail-safe packets that are waiting for transmission. If the fault is in one of the node's neighbors or in a link to a neighbor, all communication with that neighbor is terminated.

[2] Packets that arrive at the node are handled as follows: All normal packets and all fail-safe packets, except *fault* and *crash*, are ignored. *Fault* packets are ignored if their *location* field indicates that they were generated as a result of the same fault that caused node j to enter the *error* l-state.

[3] Node j sends *fault* packets to all its neighbors.

[4] If the *version* variable in node j is not set to *unknown*, node j sends a *recover*(*version*) packet to all its neighbors.

If *version* is set to *unknown*, node j waits for a *recover* packet from one of its neighbors. When the *recover* packet arrives, node j sets its *version* to the value in that packet and sends a *recover*(*version*) packet to all its neighbors.

If the error is a result of a fault in the checkpointing coordinator, all the working nodes in the system may have their *version* variable set to *unknown*. Hence, if node j is a neighbor of the checkpointing coordinator whose *version* is set to *unknown*, and if the fault packets indicate that the checkpointing coordinator has failed, node j waits for a *recover* packet only up to some preset time limit and then sets its *version* to *old-version* and sends a *recover*(*version*) packet to all its neighbors.

[5] Node j changes l-state to *recovering* and waits for its complete state to arrive from one of the disk nodes.

[6] Node j sends a *restored* packet to the (possibly new) checkpointing coordinator, changes l-state to *recovered*, and waits for a *resume* packet from one of its neighbors.

[7] If node j is the checkpointing coordinator, when it receives *restored* packets from all the nodes which are known to be working (including itself), it sends *resume* packets to all its neighbors and changes l-state to *normal*.

If node j is not the checkpointing coordinator, when a *resume* packet arrives from one of its neighbors, it sends a *resume* packet to all its neighbors and changes l-state to *normal*.

[8] Node j resumes normal operation.

(II) The Actions of a Disk Node

As previously mentioned, if an error is detected in a disk node, a *crash* is initiated (in Subsection 5.6.4 we discuss modifications to the system that enable it to recover from failed disk nodes). However, a disk node can participate, or even initiate, the recovery process when the source of the error is some other node.

During recovery, every disk node j sends the checkpointed state to those nodes whose state is stored on the disk controlled by j. The node states are sent using sequences of *state* packets. Sending the checkpointed state to the nodes can begin only when it is determined which version of the state should be used. Hence, the disk node begins sending the checkpointed states only after it has gone through step [4] of the recovery protocol described above. The value of *version* for the state that is sent by the disk node is the value of the *version* variable of the disk node following step [4] above.

The disk nodes must ensure that the "files" on the disks are restored to a state that is consistent with the state of the nodes. If the value of the *version* variable in some disk node is not *unknown* (*i.e.*, it is set to 0 or 1) when it first enters the *error* l-state, the system is rolled back to the checkpoint to which the disk node has already committed. All updates to the disk that were performed after the last checkpoint must be undone. Hence, the disk node removes all the duplicate files and creates new duplicates from the "master copy" for all the files that are marked in the master copy as open with write or read/write permission.

If the disk node is in the middle of a checkpointing session, its *version* variable may be set to *unknown*. If recovery requires rolling back to the previous checkpoint rather than to the one being established, the value of *version* changes to *old-version* when the l-state changes to *recovering*. In this case the disk node is required to perform the same actions as when the value of *version* is initially set to a value other than *unknown*.

If recovery requires "rolling back" to the checkpoint that is currently being established, the value of *version* is *unknown* when the disk node first enters the *error* l-state but changes to $1-$*old-version* when the l-state changes to *recovering*. In this case the disk node must commit to all updates that were done to the disk since the last checkpoint. Hence, the disk node commits to the node states that were just received and updates all files for which there are temporary duplicates with the contents of the duplicates. For files that have been closed since the last checkpoint, the duplicates are removed. For files that are still active, the duplicates remain and continue to be used by the disk node.

A disk node may fail or receive an *error* packet in the middle of committing to a new checkpoint. For the time being, we assume that any error in the operation of the disk node causes a crash. If the disk node receives an *error* packet in the middle of

committing to a new checkpoint, it forwards the packet to all its neighbors but does not proceed with any recovery actions until it completes the process of committing to the new checkpoint.

5.4. Correctness Arguments

Since a complete proof of the correctness of the protocols presented in this chapter requires a lengthy case analysis, we limit our discussion here to showing that, despite hardware faults, the system will never produce ("commit to") incorrect results. It is shown that errors that might lead to incorrect results are always detected. Furthermore, when recovery occurs, the state to which the system is rolled back is valid. This requires showing that disk nodes only commit to *valid* checkpoints and that during recovery all the working nodes are rolled back to the same *consistent* checkpoint.

As previously mentioned, the last action of the system, before committing to the output of a task, is to establish a new checkpoint. Thus, in order to show that the output is correct it is sufficient to show that a checkpointing session can terminate successfully only if the states of all the nodes are correct and there are no errors in transmitting output from the various nodes to the disks. In Subsection 5.4.1 it is shown that the states of the individual nodes are correct. In Subsection 5.4.2 it is shown that the states of all the nodes that are saved as part of a single system checkpoint are consistent with each other.

5.4.1. The Correctness of Individual Node States

The correct operation of a particular pair of neighboring nodes and the link connecting them is verified when both enter the *checkpointing* l-state. Errors that are a result of faults in either node are always detected immediately when they occur. The two neighbor nodes can both enter the *checkpointing* l-state only after they have exchanged *checkp* packets. These *checkp* packets follow any normal packets transmitted between the two nodes. If the *checkp* packets do not cause one or both of the nodes to enter the *error* l-state (*i.e.*, the CRC check bits match), then there have been no errors in the transmission of normal packets between the nodes since the last checkpoint. Thus, if both nodes enter the *checkpointing* l-state, the only way for one of the nodes to have received an incorrect normal packet from the neighbor is if the latter was correctly forwarding an incorrect packet from some other node. It is shown in the next paragraph that this

situation is also detected during the checkpointing process. Before checkpointing is complete, all the nodes in the system must go through the *checkpointing* l-state. Thus, all the normal packets received by any one of the nodes since the last checkpoint must have been correctly forwarded.

It is still possible for the internal state of a node to be erroneous. This erroneous internal state can only be a result of incorrect processing of *correct* packets. Since the nodes are self-testing, an erroneous internal state is detected when the state is sent to the disk node. For example, if duplication and comparison is used to implement the self-testing nodes (see Chapter 3), the internal state is generated and stored independently on two identical functional units. When the state is sent to the disk node, the two versions are automatically compared and if they are not identical, the neighbor receives an error signal. Thus, if *all* the nodes complete sending their state (*i.e.*, enter the *checkpointed* l-state), all those states must be valid. It remains to be shown that the states of the nodes that are sent to disk nodes during a particular checkpointing session are consistent with each other and are stored and retrieved correctly.

The state of each node is sent to a disk node using fail-safe *state* packets. After each "hop" these packets are checked and any error that resulted from a fault in the link is detected. Since the nodes are self-checking, any error in forwarding the *state* packets is detected immediately. The node detecting the error is always the next node on the path to the disk node. Hence, if there is any error in transmitting the state packets from their source nodes to the disk nodes, some of the packets will not arrive at their destination. If a *state* packet from some node i does not reach its destination, the corresponding disk node does not send the *saved* packet for node i to the checkpointing coordinator and the checkpointing session is never completed. Hence, a checkpointing session can be completed only if the states of all the nodes arrive at the disk nodes intact.

Since the disk nodes are self-checking, errors in the operation of these nodes is detected immediately by neighbors and causes a crash. As previously mentioned, the disk nodes use an error-detecting code when storing any information on disk. If there are any errors in transmitting the information to the disk itself or in storing the information, the error is detected when the information is retrieved. Thus, during recovery, the disk nodes either retrieve the node states correctly or detect an error and initiate a crash.

During recovery, the node states are sent from the disk nodes to their destination

nodes using *state* packets. Any errors in the transmission of the *state* packets are detected immediately and cause a crash. Each node can determine when it receives its entire state using the *seq* and *size* fields in the *state* packets. Normal operation is resumed only after all the nodes have sent a *restored* packet to the checkpointing coordinator. Thus, normal operation can be resumed only if the entire checkpointed state is retrieved correctly.

5.4.2. The Consistency of Node States in a Single Checkpoint

The state of a node changes as a result of local computation, transmission of a normal packet, or reception of a normal packet. The saved states of two neighbor nodes are consistent if no normal packets are transmitted between the nodes after the state of one of the nodes has been saved but before the state of the other node is saved [Bari83]. When a node enters the *checkp-begin* l-state, it stops local computation and transmission of normal packets. A node enters the *checkpointing* l-state only after all its neighbors have entered the *checkp-begin* l-state. Hence, after the node has entered the *checkpointing* l-state, no more normal packets are exchanged with *any* of its neighbors until normal operation is resumed. As the checkpointing session progresses, each one of the neighbors enters the *checkpointing* l-state and sends its own state to a disk node. Therefore, the saved state of each node is consistent with the saved states of all its neighbors. Thus, if a checkpointing session is not interrupted by an error, it is guaranteed that all the node states that are part of that checkpoint are consistent with each other.

The system must be able to recover from an error that is detected in the middle of a checkpointing session. Under these circumstances, some of the disk nodes may receive the *resume* packet and commit to the new checkpoint while other disk nodes are still committed to the previous checkpoint. As a result, the two groups of disk nodes may cause working nodes to "recover" with inconsistent states. The *version* variable stored with each node is introduced in order to solve this problem. During a checkpointing session, before any node i completes sending its entire state to a disk node, it (node i) can determine independently of any other node that any recovery must involve rolling back to the *previous* checkpoint rather than the one being saved. Once i completes sending its state, it can not longer determine whether recovery should involve rolling back to the previous checkpoint or to the one stored during the current checkpointing session; therefore, node i sets is *version* variable to *unknown*. After the checkpointing coordinator

has received *saved* packets for all the nodes in the system, it is guaranteed that all the disk nodes together have a complete and consistent new checkpoint. The *resume* packet sent by the checkpointing coordinator informs all the other nodes that the new checkpoint is valid. If an error is detected, the checkpointing coordinator always "knows" whether the system should be rolled back to the previous checkpoint or to the one being established. If the checkpointing coordinator is in the middle of a checkpointing session and has already received *saved* packets for all the nodes in the system, the checkpoint being established in the current checkpointing session must be used. Otherwise, only the previous checkpoint is guaranteed to be correct and the checkpoint that is in the process of being established must be discarded. Every node in the system either "knows" to what checkpoint the system should be rolled back, or "knows" that it is not able to make that determination. It is not possible for two nodes to have complementary values in their *version* variables. Since the disk nodes begin sending the checkpointed state only when their *version* variable is not *unknown*, the system is rolled back to a consistent state.

As previously mentioned, when the system commits to a new checkpoint, it also commits to the output generated since the last checkpoint. Disk output in the system is sent from the various nodes to the disk nodes during normal operation using normal packets. A checkpointing session can terminate successfully only if no errors occurred as a result of faults in nodes or links. Thus, if the checkpointing session completes, it is guaranteed that all the output received by disk nodes since the last checkpoint is correct. The disk nodes use error-detecting codes that guarantee that any error in storing the output on the disk will be detected when the information is retrieved. Thus, when the system commits to output on its disks, that output is either correct or, if it is incorrect, the error can be detected based on the error-detecting code used to store the information.

5.5. Estimate of the Overhead for Fault Tolerance

Accurate estimates of the overhead of making the multicomputer fault tolerant using the scheme proposed in this chapter require detailed simulation of the system, including the queues at the communication ports, the time it takes to move data in memory, etc. Such information can only be obtained for a particular application after a detailed design of the system is complete. In order to provide a rough estimate of the expected overhead associated with the proposed error recovery scheme, we make several assumptions about the system based on the stated application environment and on current and near-future

technology:

(1) The system includes one thousand nodes, each consisting of a high-performance (for example, 5 MIPS [Barr83]) processor. The processor state is, on the average, 256 thousand bytes. It should be noted that the state does not include code-space.

(2) The topology of the system is dense, *i.e.*, the *diameter* is proportional to the logarithm of the number of nodes. Specifically, we assume a diameter of 15.

(3) Ten of the nodes in the system are disk nodes, each handling checkpointing and recovery of the state of 100 nodes.

(4) The communication links are assumed to have a bandwidth of 1.5×10^6 *bytes/second* [Barr83, INMO84].

(5) Every fail-safe packet that is not a *state* packet is 16 bytes long, including redundant bytes for error detection.

(6) Each *state* packet is 1000 bytes long and 260 such packets are required to transmit the entire state of a node.

(7) Each node can be simultaneously receiving a packet, processing a previously received packet, and sending a previously processed packet [Barr83, INMO83].

(8) The bandwidth of the interface between the disk node and the disk drives it controls is much higher than the bandwidth of the communication links, and the node can transfer data to the disk drive simultaneously with all its other activities.

In order to initiate a checkpointing session, the *checkp* packets must propagate from the checkpointing coordinator to all the other nodes in the system. The 16 byte *checkp* packet goes through a link in 11 *μsecs*. The processing required at each node to forward the packet is relatively simple. For a 5 MIPS processor with an architecture that is appropriate for a multicomputer (e.g., the INMOS Transputer [INMO84]), 50 *μsecs* is a pessimistic estimate of the delay introduced by this processing. Since the diameter of the systems is 15, all the nodes in the system can enter the *checkpointing* l-state within one milliseconds after the checkpointing session is initiated.

The state of each node is transmitted to a disk node using 260 *state* packets. Each 1000-byte *state* packet can be transmitted through a single link in 670 *μsecs*. Hence, every disk node begins receiving *state* packets within one or two milliseconds after all the nodes enter the *checkpointing* l-state. A regular node receiving a *state* packet, can certainly verify it and forward it to the appropriate output port within the 670 *μsecs* it

takes to transmit the packet over a link. Hence, even if a disk node can only receive *state* packets through one of its ports, these packets can utilize the full bandwidth of the link. Since the bandwidth of the interface between the disk node and the disk drive has a much higher bandwidth than the communication link, the disk node can store the state packets as fast as they arrive. Thus, all 26,000 *state* packets containing the state of 100 nodes can be received by a disk node in approximately 18 seconds.

Sending *saved* packets from the disk nodes to the checkpointing coordinator is a similar process to distributing *checkp* packets from the checkpointing coordinator to the rest of the system. Hence, this process is expected to take approximately one milliseconds (see above). Similarly, distributing the resume packet from the checkpointing coordinator to the rest of the system is also expected to take approximately one milliseconds. Thus, the entire checkpointing session can be expected to take less than 19 seconds to complete.

If a checkpoint is created twice an hour, the overhead involved in maintaining the checkpoint is, approximately 1.1 percent.

The process of recovery is very similar to the process of creating a checkpoint — 26,000 *state* packets are transmitted from the disk nodes to all the other nodes in the system. Hence, recovery can also be expected to take approximately 20 seconds. When an error is detected, the system is rolled back and any computation done since the last checkpoint is lost. Since a new checkpoint is created every 30 minutes, on the average, 15 minutes of computation are lost every time the system is rolled back. If the MTBF of the system is 10 hours, the total overhead for fault tolerance during those 10 hours includes 6.6 minutes for creating checkpoints, 0.4 minutes for error recovery, and 15 minutes of lost computation. The total of 22 minutes amounts to an overhead of 3.7 percent.

5.6. Relaxing Some of the Assumptions

In this section we outline how some of the restrictive assumptions made in Subsection 5.3.1 can be relaxed. In particular, it is no longer assumed that the system must be "closed." Some communication with the "outside world" is allowed. Since transient hardware faults are at least an order of magnitude more likely to occur than permanent faults [Cast82], it is wasteful to logically remove a node or a link after it suffers from a fault. A more efficient way of dealing with transient faults is proposed. Finally,

modifications to the system that allow recovery from faults in disk nodes are discussed.

5.6.1. Input/Output from the System

The basic problem in allowing communication with the outside world is that it may be impossible to roll back the effects of such communication: if a page is printed, it cannot be erased; if a file is read and then deleted, it may not be possible to restore its contents; if input is obtained from a terminal, it is not acceptable to ask the user to retype all his commands after the system recovers from an error. The problem is especially difficult with our scheme that allows the system to continue operating incorrectly for a relatively long time (up to the time between successive checkpoints) after an error has occurred.

The multicomputer consists of nodes, links, and disk drives. We will call any other system (computer) or device that interacts with the multicomputer, a "peripheral." Some peripherals, such as another computer system, may be able to commit to checkpoints and roll back to those checkpoints upon demand. We call such "devices" *intelligent peripherals*. Most peripherals, such as printers or tape drives, cannot set checkpoints and roll back to them. We call these latter devices *simple peripherals*.

Due to similarities between the actions performed by a node that controls a disk that is part of the system and a node that interacts directly with the outside world, it is convenient to refer to nodes that interact directly with the outside world as *disk nodes*. Information transfer from a disk node to a peripheral is *output* while information transfer in the opposite direction is *input*.

5.6.1.1. Intelligent Peripherals

With intelligent peripherals, input/output may occur virtually at any time. As part of each checkpointing session, when the disk nodes commit to the new checkpoint, each disk node connected to a peripheral "commands" the peripheral to commit to any data it received from the disk node or transmitted to the disk node since the last checkpoint. If the peripheral signals an error, the disk node initiates a crash.

If an error is detected in the multicomputer and a recovery session is initiated, the peripherals are instructed to roll back to a previous checkpoint or possibly, if the error is detected in the middle of a checkpointing session, to commit to a new checkpoint. The disk node connected to each peripheral is able to inform the peripheral which of these actions to take as soon as it determines the correct value for its *version* variable (see

Subsection 5.3.6).

If a peripheral accepts or transmits erroneous data without detecting the error, it is possible that the multicomputer system will generate incorrect results. The probability of such incorrect results is reduced by using error detecting codes for all data transfers between disk nodes and peripherals.

5.6.1.2. Simple Peripherals

As mentioned earlier, only disk nodes interact directly with peripherals. Regular nodes can interact with any peripheral indirectly by sending input/output requests to the appropriate disk node. These requests are sent to the disk node using normal packets. Some of these packets may be erroneous due to faults in links that are only detected during checkpointing. Since operations on "simple peripherals" cannot be undone, the disk node does not execute any of the I/O requests until they are verified to be correct by a checkpointing session. Instead, I/O requests are accumulated in temporary files on disk drives controlled by the disk node. If the system is rolled back to a previous checkpoint, all these I/O requests are discarded.

When the disk node commits to the new checkpoint, it also places the accumulated I/O requests in a queue of verified peripheral operations to be executed. Since data transfers to/from the peripheral cannot be repeated, this queue is *not* part of the node state that may be rolled back during recovery. After the checkpointing session is completed, the disk node performs the peripheral operations in the queue and keeps track of all data transfers to/from the peripheral so that they are not repeated if the system is rolled back.

When executing input operations, the disk node first stores the data received from the peripherals in temporary files on disk and later forwards the data to the nodes that had initiated the input requests. If an error is detected and the system is rolled back, any packets transferring data from the disk node to other nodes are lost. These packets can be considered "safely on their way" only after the next checkpointing session. Hence, the disk node must keep the temporary files until the next checkpointing session so that it is able to resend any packets containing data from the peripherals if the system is rolled back. It should be noted that, *immediately* after they are created, the temporary files containing data from the peripherals are treated as though they are part of the *previous*

checkpoint. These temporary files are logically "removed" when the data is sent from the disk node. However, since they are part of the previous checkpoint, they are only physically removed during the *next* checkpointing session.

The disk node must keep track of input requests that have been verified as correct (by a previous checkpointing session) but are not yet completed since the data has not yet been sent to the node that requested it. At the same time (during checkpointing) when all I/O requests are placed in the queue of peripheral operations to be executed, the input requests alone are also placed on another queue of "input requests that are not yet completed." This queue is handled in the same way as the temporary files mentioned above — it becomes part of the *previous* checkpoint and all modifications (updates) to it are committed only during the next checkpointing session.

The above scheme does not allow interactive access to the multicomputer. However, it does allow it to accept new tasks during every checkpointing session and to produce partial outputs as the task progresses. Hence, the multicomputer is no longer required to be a "closed system." It is possible to allow regular data transfers with a a host system that interacts with the users directly, "prepares" jobs for the multicomputer, and handles the output from those jobs.

5.6.2. Handling Errors Caused by Transient Faults

Most of the errors in computer systems are a result of transient faults [Cast82]. Such errors can corrupt the state of the system so that rolling back to the last checkpoint is necessary. However, the hardware itself is not permanently affected and should be used again once a valid system state is established.

Since a link does not contain any state, no special actions are required in order to continue to use it after recovering from an error caused by a transient fault on that link. On the other hand, the computation nodes do contain state that may be corrupted by a fault thereby preventing it from cooperating with the rest of the system in establishing a "sane state." Thus, a node that fails due to a transient fault should be *reset* to some valid initial state that allows it to communicate with other nodes in the system. Once the node is in this initial state, it can obtain information about the condition of the system (for example, which nodes or links are faulty) and accept its checkpointed state so that it can resume normal operation.

If the self-checking nodes are implemented using duplication and comparison [Tami83], the "no-match" signal from the comparator can be used to reset the node to a "sane" state following a transient fault. This reset causes the node to begin executing resident code that is stored in ROM. It should be noted that neighbor nodes must not be given the authority to reset a failed state since that would allow a failed node to reset its fault-free neighbors.

Given the ability to reset a failed node, recovery from an error caused by a transient fault is simpler than recovery from an error caused by a permanent fault — there is no need to reroute packets around failed nodes or links and it is not necessary to migrate processes that were assigned to the failed node to other nodes. However, permanent faults must be distinguished from transient faults in order to prevent repeated errors caused by a single permanent fault from disrupting the operation of the system. When an error is detected, the node or link that caused the error is identified by the *location* field in the *fault* packets. Each node in the system can keep a record, in its own memory, of the causes of the last few errors. If the same node or link is the source of several consecutive errors, that node or link is considered permanently faulty by all the other nodes in the system, which make no further attempts to use it.

5.6.3. Faster Detection of Errors Caused by Faulty Links

With the fault tolerance scheme described so far, errors caused by faulty links are only detected during the next checkpointing session. Thus, after a faulty link causes an error, the system continues processing the erroneous information until the next checkpointing session. All this processing is useless since, upon detecting the error, the system is rolled back to the last checkpoint. In addition, the delay in detecting errors caused by faulty links can lead to system crashes. If two links are affected by faults during normal operation, the two independent faults may be detected simultaneously during the same checkpointing session and result in a system crash.

In order to detect errors caused by faulty links as soon as possible, every packet must include redundant check bits that are checked after every transfer over a link (see Subsection 5.3.2). When a node detects an error it can initiate a recovery session using *fault* packets (see Subsection 5.3.6). The additional overhead required by this scheme includes the communication bandwidth used to transmit the redundant bits, possible additional delay in relaying the packet at each intermediate node on its path, and

additional hardware and/or software (firmware) at each node for performing the validity checks on each packet.

It should be noted that the above scheme does not detect lost packets and therefore does *not* eliminate the need for checking the links during the checkpointing session, as described in Subsections 5.3.2 and 5.3.5. Faster detection of lost packets requires much more complicated protocols. After a node sends (or forwards) a packet to a neighbor, it waits for an acknowledgement. If no acknowledgement is received within a set time, the sender "times out" and initiates recovery. With the system discussed in this chapter, losing a packet while it is transmitted from one node to its neighbor, is a very unlikely failure mode. Hence, the additional overhead required for fast detection of lost packets is not justified.

Another possible scheme for detecting errors caused by faulty links is to verify the validity of messages only at their final destination rather then at each intermediate node. Instead of including check bits with each packet, the system may use only one set of check bits for each message (that may be sent using several packets). The check bit are generated by the source of the message and checked only by the destination. This scheme involves less overhead than verifying individual packets. However, the delay in detecting errors is greater and there are more possible errors that can only be detected during a checkpointing session.

5.6.4. Faults in Disks and Disk Nodes

With the fault tolerance scheme described so far, the system cannot recover from errors caused by faults in disks or disk nodes. The basic problem in recovering from such errors is that data may be corrupted or no longer accessible. If the only access to parts of a checkpointed state or to data required by the task is through a single disk node, there is no way to recover from a permanent fault in this node. Similarly, if parts of the checkpointed state or other data is stored on only one disk, the system cannot recover from a failure of the disk or of the disk controller.

The solution to the above problem requires storing multiple copies of critical data and providing multiple paths to the data. In several commercial systems [Borg83, Katz82] this is being done by using multiple dual-ported disk drives and dual-ported disk controllers. The two ports of each disk drive are connected to two independent disk

controllers and the two ports of each disk controller are connected to two independent nodes. Each critical file is stored on two disk drives. With this scheme, critical data remains accessible despite the failure of one of the disk nodes, one of the disk controllers, or one of the disk drives.

The hardware described in the previous paragraph is well suited to the fault tolerance schemes of most commercial systems. These schemes involve the use of "process pairs," a "primary" and a "backup," located on different nodes with the "backup" ready to take over the execution of the process if the "primary" fails. In accordance with this scheme, the process pair that interacts with the disk controller is located on the two nodes connected to the ports of the controller.

The hardware described above can be used with the multicomputer. However, unlike the fault tolerance schemes used in commercial systems, the scheme presented in this chapter does not involve maintaining a process pairs. There are two possible ways of using the pair of disk nodes connected to dual-ported controllers: (1) As long as both disk nodes are operational, only one of the nodes performs the input/output tasks of a disk node while the second node operates as a normal node but is kept ready to take over input/output operations in case the first node fails. (2) Both nodes perform input/output operations and one of them begins performing all these operations if the other one fails. In this chapter we will only discuss the first, simpler, alternative.

For each pair of nodes connected to the same disk controller, we call the node that performs input/output operations "an *active* disk" node and the other node "a *passive* disk node." The other nodes on the system initially use the active disk node but they also "know" the identity of the corresponding passive disk node and begin using it if the active disk node fails.

Each output operation is performed on both disk drives. Data is written with redundant bits for error detection. After the data is written, it is immediately read by the active disk node and verified as correct based on the error-detecting code. If an error is detected on both disk drives, the node first retries the operation using the same disk controller and disk drive. If the retry fails, the disk node switches to the other disk controller. If the disk node detects the failure of both disk drives or both disk controllers, it must initiate a *crash*.

The failure of an active disk node is detected by its neighbors just like the failure of

any other node. The rest of the system is informed of the failure of the node by the *fault* packets. The passive disk node connected to the same disk controller takes ownership of the disk controllers and begins serving as the disk node[Katz82]. All the nodes in the system are informed that an active disk node has failed and update their internal tables to indicate that the corresponding passive disk node is now the active disk node. If an active disk node fails and the other node in the disk node pair is known to be faulty, a crash is initiated.

Recovery from the failure of a disk node is similar to the recovery from the failure of any other node — the system rolls back to the previous checkpoint and resumes operation without the failed node. If an active disk node fails, the corresponding passive disk node must take over the tasks of the failed node, thereby becoming an active disk node, and every other node in the system must begin sending all input/output requests to the new active disk node. In order to be able to take over the tasks of an active disk node, the passive disk node must have some information on how data is stored on the disk. In particular the passive disk node must be able to access a prearranged location on the disk that contains pointers to the last committed checkpoint including node states and disk files.

If an active disk node fails, its passive "partner" must be able to obtain the correct version of the checkpointed state stored on the disks controlled by the two nodes even if the failure is detected in the middle of a checkpointing session. If the failure is detected before the checkpointing coordinator receives *saved* packets for all the nodes in the system, recovery involves roll back to the previous checkpoint rather than to the checkpoint currently being saved. In this case the previous active disk node has not yet begun committing to a new checkpoint and the new active disk node can access the previous checkpoint in the same way as when the error is detected during normal operation.

If the failure of an active disk node is detected after the checkpointing coordinator has received *saved* packets for all the nodes in the system, recovery requires "roll back" to the *new* checkpoint that has just been saved. At this stage the previous active disk node may have completed committing to the new checkpoint, may be in the middle of committing to the new checkpoint, or may have not yet started committing to a new checkpoint. In order to be consistent with the rest of the system, the new active disk

node must be able to access the *new* checkpoint (see Subsection 5.3.6) in all three cases.

The key to solving the above problem is that the actions performed by the disk node in order to commit to a new checkpoint are *retryable*. If the passive disk node "knows" of a prearranged location on disk that contains pointers to the new checkpoint, it can restart the task of committing to the new checkpoint from scratch since all that is required is copying pointers from the new checkpoint area to the committed checkpoint area. After a disk node completes committing to a new checkpoint, it stores the value of *version* that corresponds to the new committed checkpoint in a known place on the disk. During recovery, the disk node can compare the value of *version* of the committed checkpoint with the value of *version* that corresponds to the checkpoint it is supposed to restore. If the two values differ, the disk node first commits to the new checkpoint and then proceeds with the rest of the recovery session as usual.

References

Ande76. T. Anderson and R. Kerr, "Recovery Blocks in Action," *2nd International Conference on Software Engineering*, San Francisco, CA, pp. 447-457 (October 1976).

Bari83. G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).

Barr83. I. Barron, P. Cavill, D. May, and P. Wilson, "Transputer Does 5 or More MIPS Even When Not Used in Parallel," *Electronics*, pp. 109-115 (November 1983).

Borg83. A. Borg, J. Baumbach, and S. Glazer, "A Message System Supporting Fault Tolerance," *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 90-99 (October 1983).

Bozy82. M. Bozyigit and Y. Paker, "A Topology Reconfiguration Mechanism for Distributed Computer Systems," *The Computer Journal* **25**(1) pp. 87-92 (February 1982).

Cast82. X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and Calibration of a Transient Error Reliability Model," *IEEE Transactions on Computers* **C-31**(7) pp. 658-671 (July 1982).

Dole81. D. Dolev, "Unanimity in an Unknown and Unreliable Environment," *22nd Annual Symposium on Foundations of Computer Science*, Nashville, TN, pp. 159-168 (October 1981).

Elki82. S. A. Elkind, "Reliability and Availability Techniques," pp. 63-181 in *The Theory and Practice of Reliable System Design*, ed. D. P. Siewiorek and R. S. Swarz, Digital Press (1982).

Gray78. J. N. Gray, "Notes on Data Base Operating Systems," pp. 393-481 in *Operating Systems: An Advanced Course*, ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin (1978). Lecture Notes in Computer Science 60.

INMO83. INMOS, *IMS T424 Transputer*, Advance Information. November 1983.

INMO84. INMOS, *IMS T424 Transputer Reference Manual*, November 1984.

John84. D. Johnson, "The Intel 432: A VLSI Architecture for Fault-Tolerant Computer Systems," *Computer* **17**(8) pp. 40-48 (August 1984).

Jone79. A. K. Jones, R. J. Chansler, I. Durham , K. Schwans, and S. R. Vegdahl, "StarOS: A Multiprocessor Operating System for Support of Task Forces," *Proc. 7th Symp. on Operating Systems Principles*, Pacific Grove, pp. 117-127 (December 1979).

Kast83. P. S. Kastner, "A Fault-Tolerant Transaction Processing Environment," *Database Engineering* **6**(2) pp. 20-28 (June 1983).

Katz82. J. A. Katzman, "The Tandem 16: A Fault-Tolerant Computing System," pp. 470-480 in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill (1982).

Kubi82. C. Kubiak, J. P. Andre, B. Grandjean, D. Mathieu, and J. Rolland, "Penelope: A Recovery Mechanism for Transient Hardware Failures and Software Errors," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 127-133 (June 1982).

Lee78. P. A. Lee, "A Reconsideration of the Recovery Block Scheme," *The Computer Journal* **21**(4) pp. 306-310 (November 1978).

Lee80. P. A. Lee and K. Heron, "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers* **C-29**(6) pp. 546-549 (June 1980).

Peas80. M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM* **27**(2) pp. 228-234 (April 1980).

Plat80. D. G. Platteter, "Transparent Protection of Untestable LSI Microprocessors," *10th Fault-Tolerant Computing Symposium*, Kyoto, Japan, pp. 345-347 (October 1980).

Prad82. D. K. Pradhan and S. M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," *IEEE Transactions on Computers* **C-31**(9) pp. 863-870 (September 1982).

Rand75. B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering* **SE-1**(2) pp. 220-232 (June 1975).

Rand78. B. Randell, P. A. Lee, and P. C. Treleaven, "Reliability Issues in Computing System Design," *Computing Surveys* **10**(2) pp. 123-165 (June 1978).

Séqu83. C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," pp. 299-326 in *VLSI Architecture*, ed. B. Randell and P.C. Treleaven, Prentice Hall, Englewood Cliffs, NJ (1983).

Siew78. D. P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part I - Experiences with Fault Tolerance in Multiprocessor Systems," *Proceedings IEEE* **66**(10) pp. 1178-1199 (October 1978).

Stro83. H. R. Strong and D. Dolev, "Byzantine Agreement," *COMPCON*, San Francisco, CA, pp. 77-81 (March 1983).

Taji77. W. D. Tajibnapis, "A Correctness Proof of a Topology Information Maintenance Protocol for a Distributed Computer Network," *Communications of the ACM* **20**(7) pp. 477-485 (July 1977).

Tami83. Y. Tamir and C. H. Séquin, "Self-Checking VLSI Building Blocks for Fault-Tolerant Multicomputers," *International Conference on Computer Design*, Port Chester, NY, pp. 561-564 (November 1983).

Wake76. J. F. Wakerly, "Microcomputer Reliability Improvement Using Triple-Modular Redundancy," *Proceedings of the IEEE* **64**(6) pp. 889-895 (June 1976).

Wens78. J. H. Wensley, L. Lamport, J. Golberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: The Design and

Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings IEEE* **66**(10) pp. 1240-1255 (October 1978).

Witt81. L. D. Wittie, "Communication Structures for Large Networks of Microcomputers," *IEEE Transactions on Computers* **C-30**(4) pp. 264-273 (April 1981).

Wood81. W. G. Wood, "A Decentralized Recovery Control Protocol," *11th Fault-Tolerant Computing Symposium*, Portland, Main, pp. 159-164 (June 1981).

Chapter Six
Implementation Considerations

The previous five chapters presented the basic principles of an approach to implementing fault tolerance in a VLSI multicomputer and discussed the advantages and disadvantages of this approach when compared with alternative approaches. However, the material presented is, by no means, a complete detailed design of a high-performance fault-tolerant multicomputer. Such a design must take into account the mix of applications for which the system is intended, the required performance and reliability, the properties of the particular implementation technology, the environment in which the system is expected to operate, and the acceptable range of system cost. A discussion of some of the issues and implementation tradeoffs that must be considered is presented in this chapter.

The key to the fault tolerance technique presented in the previous chapters is the use of self-checking nodes implemented with duplication and comparison. As discussed in Chapter 3, one of the potential weaknesses of duplication and comparison is that if the two functional modules fail simultaneously in exactly the same way, the failure is not detected and incorrect results are accepted as correct by the rest of the system. Techniques for reducing the probability of such *common mode failures* are presented in Section 6.1. This section includes a discussion of the possible causes of common mode failures and some basic definitions. It is shown that it is not possible to entirely eliminated common mode failures. Instead, there are some practical implementation techniques for reducing the probability of these failure in the context of commonly used NMOS and CMOS circuits.

The technique presented in Section 6.1 is an important implementation detail that can increase the effectiveness of the self checking nodes. Many other design choices and implementation details must be considered. A brief overview of some of these issues is presented in Section 6.2.

6.1. Reducing Common Mode Failures in Duplicate Modules

We have discussed the use of duplication and comparison to implement self-checking nodes. This technique is obviously also applicable to the implementation of any other *self-checking functional module* (henceforth, *SCFM*). Hence, for generality, SCFM will be used instead of "self-checking node" throughout this section. A general SCFM is shown in Fig. 6.1.

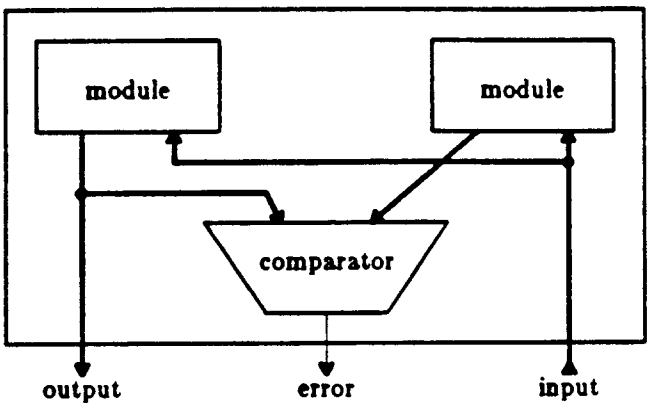


Fig. 6.1: A Self-Checking Functional Module (SCFM)

Modules that perform identical functions may fail simultaneously in exactly the same way and produce identical incorrect results. Such *common mode failures* (henceforth, *CMFs*) may be caused by environmental factors such as power supply fluctuations, pulses of electromagnetic fields, or bursts of cosmic radiation, that can affect both modules at the same time, triggering similar design weaknesses and causing simultaneous identical failures of both modules. Simultaneous module failures may also be caused by faults that occur at different times in parts of the modules that suffer from identical design weaknesses and are infrequently exercised.

With advances in VLSI technology it will soon be possible to implement an entire SCFM (such as a self-checking node in a multicomputer), including the two functional modules and the comparator, on the same chip. In addition to providing error detection during normal operation, the self-checking capability of the chip may also be used to simplify the testing of the chip throughout its life: from wafer probe testing that is part of the manufacturing process through the acceptance tests by users and diagnostic testing for repair and preventive maintenance of the system containing the chip. The simplification of testing is achieved by eliminating the need to store the correct responses to long test sequences and compare them with the actual responses of the chip during testing. Testing

can proceed at the normal system clock rate and only the outputs of the comparator need to be monitored.

Unfortunately, if the two modules are fabricated on the same chip, the probability of CMFs during normal operation is greater than if they are on separate chips. This increased probability of CMFs is due to the tighter electrical and physical coupling between the two modules and to similar weaknesses in the two modules that may be caused by fabrication flaws specific to the wafer containing the chip. Furthermore, CMFs may be a significant problem if the self-checking capability of the chip is also be used to simplify its testing, especially fabrication testing. In chips that have never been tested, (i.e., have not yet gone through wafer probe testing), as a result of fabrication defects, CMFs may be relatively common, especially if the modules are physical duplicates. Hence, if wafer probe testing relies on the self-checking capability of the chip, different physical implementations of the two modules must be used.

The simplest way to implement modules that perform identical functions is to use identical physical duplicates. For such modules the meaning of the term "common mode failures" appears obvious. However, if the two modules perform identical functions but are physically different, there is no direct correspondence between physical faults in the two modules, and the meaning of the term is unclear. Hence, there is a need for a definition of CMFs that is applicable to modules that are physically different.

In the rest of this section, F will denote the set of all *single faults*, where a single fault is a fault caused by a single physical defect. In discussing the failure of the two modules in a SCFM, a "double faults" (f_1,f_2) occurs when $f_1 \in F$ affects one of the modules while $f_2 \in F$ affects the other module.

The two modules are denoted by A and B. When both modules are fault-free, both are implementations of some function Z. The implementation of Z by module A is denoted by Z_A. For every input I, $Z_A(I) = Z_B(I) = Z(I)$. When the module A is affected by a fault $f \in F$, it performs the function Z_A^f. The two modules may produce identical incorrect results due to unrelated faults that just happen to affect the outputs in the same way. In this situation, f_1 affects A, f_2 affects B $(f_1,f_2 \in F)$, and there is an input I such that $Z_A^{f_1}(I) = Z_B^{f_2}(I)$ even though $Z_A^{f_1}(I) \neq Z(I)$. Hence, there is a non-zero probability that a supposedly self-checking SCFM will fail to flag erroneous output. Thus, the SCFM is <u>not</u> *fault-secure* [Wake78] with respect to certain "double faults" that affect both

modules even if the modules are not physical duplicates.

In the worst case, the new functions, $Z_A^{f_1}$ and $Z_B^{f_2}$, performed by the faulty modules are identical, and the fault is *never* detected since for *every* input I, $Z_A^{f_1}(I) = Z_B^{f_2}(I)$. In this case, the SCFM is not even *self-testing* [Wake78] with respect to the "double fault" (f_1,f_2).

While it is clearly impossible to ensure that the SCFM will be fault-secure with respect to every double fault $(f_1,f_2) \in F \times F$, one might hope that appropriate implementation of the modules can ensure that the two module functions, as modified by the faults (f_1,f_2), are not identical, so that the fault is detectable. If this is done, the SCFM is *partially self-checking* [Wake78] with respect to all double faults $(f_1,f_2) \in F \times F$. We thus make the following definitions:

Def. 6.1: The two modules in a SCFM are said to be affected by *common mode failures*, if and only if, there exists at least one input vector for which both modules produce incorrect outputs, *and* for every input, the outputs from the two modules are identical.

Def. 6.2: Two modules are said to have *independent failure modes* with respect to a fault set F, if and only if, for every double fault $(f_1,f_2) \in F \times F$, such that f_1 affects one of the modules and f_2 affects the other, there exists at least one input that results in different outputs from the two modules.

In the definition above, F does not include faults on the input and output lines of the modules since it is clearly impossible for the two modules to have independent failure modes with respect to such faults. The technique for handling such faults for the self-checking node in a multicomputer has been discussed in Chapter 3.

6.1.1. Implementing Modules with Independent Failure Modes

For a particular function it is sometimes possible to find two different implementations with independent failure modes. For example, consider the combinational logic function defined by the truth table in Fig. 6.2. Fig. 6.3 contains two possible implementations of this function. It can be shown that these two implementations have independent failure modes with respect to the single stuck-at fault model.

Unfortunately, it is usually very difficult or impossible to find implementations with independent failure modes for even simple combinational functions and under the

ab \ cd	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	1	1	1	1
10	1	0	1	0

Fig. 6.2: A Combinational Logic Function

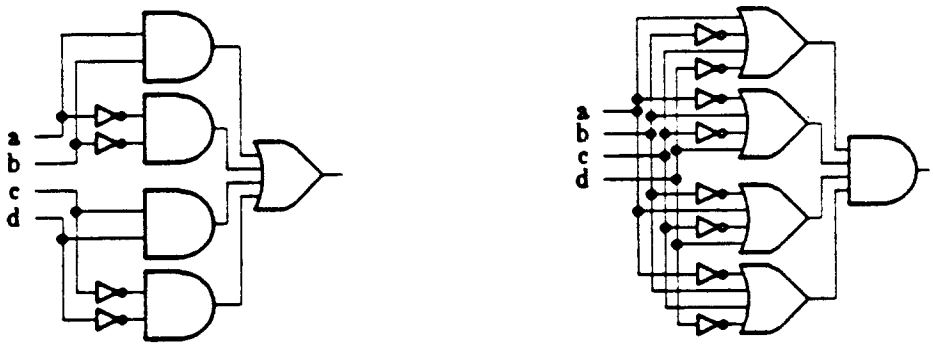


Fig. 6.3: Implementations of the Function Defined by Fig. 6.2

assumptions of a simplistic fault model, such as the single stuck-at model. As an experiment, the implementations of several simple combinational functions were considered. Except for the function described by the truth table in Fig. 6.2, implementations with independent failure modes with respect to single stuck faults could not be discovered. It is likely that such implementations do not exist for most combinational functions.

As noted in Chapter 3, one of the benefits of using duplication and comparison for self-checking subsystems is that relatively little extra design effort is required in order to implement the self-checking property. Even if it is possible to design very simple modules that have independent failure modes with respect to single stuck faults, it is unlikely to be practical and economically feasible for complex functional modules (such as microprocessors), especially if we take into account the more realistic fault model described in Chapter 2.

Assuming that there is no practical way of implementing modules that have independent failure modes with respect to all double faults, we concentrate our efforts on reducing the probability of those double faults that are more likely to occur than random double faults. The technology and circuits used to implement the modules in an SCFM

determine which double faults are more likely to occur and whether they are detectable. Hence, a particular implementation technology and "representative" example circuits are considered rather than attempting to apply uniform analysis to all possible circuits. In particular, only NMOS and CMOS VLSI implementations are considered. As a "representative" circuit we consider the Berkeley RISC microprocessor [Patt82] for which there is an NMOS VLSI implementation [Sher84] as well as a nearly complete CMOS layout [Taka83].

Many months (or years) are devoted to the design of VLSI chips in order to achieve maximum functionality, performance, and reliability with the given technology. In most cases it is unacceptable to double the design time and development cost of a VLSI chip simply to achieve more reliable error detection by reducing the probability of CMFs. Completely independent implementations of the two modules in the SCFM are therefore not practical. The use of duplicate physical modules in the SCFM is the lowest cost alternative. However, given the time and resources spent on designing a VLSI chip, it is worthwhile to spend a few additional weeks on the implementation of both modules in order to minimize some of the performance and yield costs of using duplication and comparison. A practical approach to implementing modules with independent failure modes involves spending most of the effort designing and optimizing one module and then "designing" the second module by modifying the first one. In the following sections we discuss how this overall approach can be applied for representative circuits in the RISC microprocessor.

6.1.2. Dual Implementations

For every combinational Boolean function $f(x) = f(x_1, x_2, \cdots, x_n)$ there is a corresponding *dual* function g such that $g(x) = \overline{f}(\overline{x})$ for every x. In the circuits C_f and C_g that implement the functions f and g, respectively, voltage levels represent the logic values. If the circuits are implemented using *positive-logic*, the "high" voltage level represents a logic 1 and the "low" level represents a logic 0. Because of the above relationship between the functions f and g, C_g is a *negative-logic* implementation of the function f and C_f is a *negative-logic* implementation of the function g. The circuits C_f and C_g are said to be *dual implementations* of the function f, and C_f and C_g are said to be *dual circuits*.

Dual implementations of arbitrarily complex sequential logic circuits are also

possible. If the inputs to the negative-logic implementation are complements of the inputs to the positive-logic implementation, the corresponding outputs from the two implementations are complements of each other.

Sedmak and Liebergot [Sedm80] have suggested that the probability of CMFs in a SCFM can be reduced by using dual modules rather than pairs of identical modules. The inputs to the SCFM are passed unmodified to the positive-logic module (henceforth called the *p-module*), and are complemented for the negative-logic module (*n-module*). If the two modules are operating correctly, their outputs are complements of each other and can be "compared" using a two-rail code checker [Cart68] (Fig. 6.4).

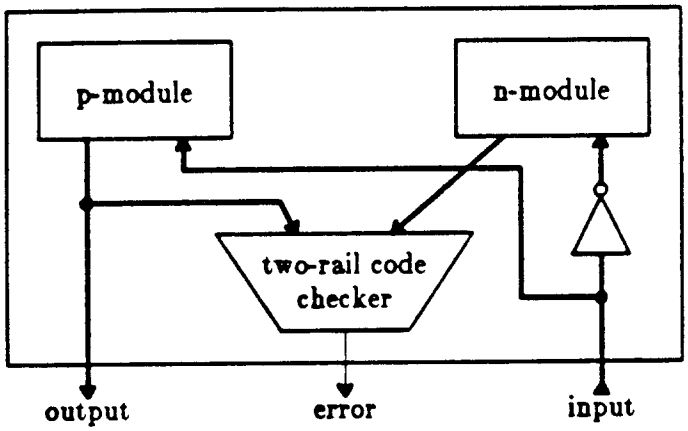


Fig. 6.4: An SCFM Based on Dual Implementations

There are several advantages to the use of the above scheme over the use of two modules that are physical duplicates: (1) If the modules are VLSI chips and the same masks are used in fabricating both modules, circuit design faults and faults in the masks result in identical incorrect results. With the dual implementations, different masks must be used since the circuits are different [Sedm80]. (2) Some pattern sensitive faults, such as those caused by electromagnetic coupling between lines or marginal design of the circuit timing, may be more likely to cause errors during voltage transitions in one direction. With dual circuits, the voltage transitions on corresponding lines in the two modules are in opposite directions; this reduces the probability of identical pattern sensitive faults occurring in the two modules simultaneously. (3) If the two modules are physical duplicates, all lines in both modules change value in the same direction at the same time. As a result there may be "spikes" in the power supply lines to the SCFM which can trigger intermittent faults. With dual circuits the problem is alleviated since values in the two modules change in opposite directions.

If SSI technology is used, dual logic implementation is relatively straightforward — the positive-logic module can be designed first and then converted into a functionally equivalent negative-logic module by a simple one-to-one replacement of gates and flip-flops with their negative-logic equivalents. Both modules have the same structure and the logic values on corresponding lines of the two modules are identical. However, since a logic 1 (logic 0) in the n-module is represented by the same voltage as a logic 0 (logic 1) in the p-module, the voltages on corresponding wires of the two modules are complements of each other. Following De Morgan's theorem, and "labeling" gates with their positive-logic functionality, for every OR (AND) gate in one of the modules there is a corresponding AND (OR) gate in the other. Similarly, for every positive-edge-triggered flip-flop there is a negative-edge-triggered flip-flop, and vice versa [Sedm80]. In this environment the structure of the module and the performance of the corresponding "building blocks" is identical (or very similar), so the extra design time for the negative-logic module is small and there is no performance penalty.

If VLSI technology is used, dual implementations is more problematic since it is not possible to convert an existing positive-logic chip to negative-logic by a simple replacement of standard building blocks. Even the conversion of NOR gates and NAND gates to negative-logic (*i.e.*, replacing NOR with NAND and vice versa) may be quite difficult due to two main factors: (1) The different gates have different topologies so the layout of the entire chip may have to be modified in order to accommodate the new gates. (2) The fan-in capability of different gates may be different — for example, in NMOS, it is possible to implement a NOR gate with a large number of inputs while a NAND gate with more than three or four inputs is not practical. Furthermore, the circuit is not simply a collection of standard logic gates and may contain transmission gates, precharged buses, register files, PLAs, decoders, dynamic logic subcircuits, etc. In a given technology, converting some of these types of circuits to negative-logic may require significantly more area and/or result in lower performance.

In the rest of this subsection we will evaluate the dual implementations approach to reducing CMFs by considering the conversion of positive-logic VLSI modules to negative-logic. This conversion does not necessarily involve converting the entire module at the lowest level (i.e., individual FETs) to negative-logic. It may be preferable to design the n-module so that some of the subcircuits in the p-module have direct negative-logic

equivalents in the n-module while other subcircuits are used unmodified in the n-module. The only critical requirement is that the n-module "behave" as the negative-logic equivalent of the p-module at the interface between the n-module and the rest of the SCFM. We will discuss the possible choices of subcircuits to be converted and the consequences of these choices in terms of design effort and the types of CMFs that can thus be eliminated.

6.1.2.1. NMOS Implementation

Standard NMOS circuits are fundamentally asymmetrical. The available devices are enhancement mode FETs (EFETs) and depletion mode FETs (DFETs). The EFETs are turned on by the "high" gate voltage and turned off by the "low" gate voltage. The DFETs are always "on" but have a higher conductivity when their gate voltage is high. There is no device that can perform the dual function of the EFET, i.e., be turned on by a low gate voltage and off by a high gate voltage. There are important consequences to this asymmetry:

(1) One of the useful building blocks of NMOS circuits is the transmission gate that can be implemented using only one EFET without power or ground connections (Fig. 6.5-A). The dual implementation of this function requires three FETs as well as a power and ground connection since the control signal must be inverted (Fig. 6.5-B).

A. Positive Logic **B.** Negative Logic

Fig. 6.5: An NMOS Transmission Gate

(2) Static logic gates use passive pull-up devices (DFETs). These gates are able to drive capacitive loads from high to low much faster than from low to high.

(3) As mentioned earlier, positive-logic static NOR gates with a large number of inputs can be implemented. However, a correspondingly simple and fast NAND gate cannot be implemented since the delay of an NMOS ratioed logic NAND circuit increases in direct proportion to the number of inputs [Mead80].

(4) Precharged buses are often used in VLSI chips as a space-efficient method of allowing

a large number of data sources to write to the bus. Since EFETs are the only devices that can be completely turned off, both the pull-up and the pull-downs must be EFETs. Since EFETs make better pull-downs than pull-ups, it is much more efficient to precharge the buses to high and drive data on the bus with pull-downs than the other way around.

The above constraints on NMOS circuits prevent the simple conversion of many of the common subcircuits in an NMOS VLSI chip to negative-logic. One of the difficulties is that many of the control lines in such a chip are connected to pass transistors that are selectively turned on depending on the clock phase and the operation performed: buses are precharged and discharged through EFETs selected by control lines, the inputs to the ALU are selected with a multiplexer implemented with pass transistors, data is "loaded" to latches through pass transistors, etc. Due to the large number of these pass transistors, it is not feasible to replace them with their negative-logic equivalents that require much more area and power (Fig. 6.5). Given that it is impossible to convert the entire chip, including all control circuitry, to negative-logic, we will consider selective conversion of some subcircuits and study the effects of this conversion on the sensitivity of the system to CMFs.

In terms of design effort, the most efficient way to implement the n-module is to use the original p-module and complement all its inputs and all its outputs. Unfortunately, this approach has no benefits in term of reducing the probability of CMFs and results in a performance penalty due to the delays of the inverters.

In order to reduce the probability of CMFs, more differences in the implementations of the two modules must be introduced. The next "step up" in this direction is to implement an n-module in which all data is stored and transferred in negative-logic but positive-logic subcircuits from the p-module are used for data processing and for control. The input data to the n-module is already in negative-logic (Fig. 6.4) and is transferred through internal buses and stored in internal registers without modification. The registers and buses require no circuit modification in order to store and transfer negative-logic data. Since the data on internal buses is negative-logic while the data processing subcircuits are designed for positive-logic inputs, the inputs and outputs of subcircuits such as the ALU must be complemented at their interface with the rest of the chip.

This approach avoides the problems with control circuits described earlier: buses, multiplexers, and latches are not modified and the transmission gate EFETs or pull-down

EFETs they contain are controlled by signals with the same polarity in both modules. Since the instructions, as well as the data, are complemented before the n-module, some modifications to the various decoders are necessary. Fortunately, decoders often require that each input will be available in both complemented and uncomplemented form. In the NMOS RISC chip, the opcode decoder, the register file decoder, the shift amount decoder, and the "jump-condition-code" decoder, all already use inverters in order to generate the complemented form of their inputs. Due to the regular structure of the decoders, modifying them for the n-module is a trivial task: the connections made to the complemented and uncomplemented versions of each input are interchanged.

In RISC, the main "data processing" subcircuits are the ALU, the shifter, and the program counter incrementer. As previously indicated, it is possible to use the p-module implementation of these subcircuits in the n-module if they are preceded and followed by inverters. In order to make room for the additional inverters, major parts of the circuit must be moved. With appropriate design tools, making such a modification is not difficult. However, these inverters require additional area and increase the power consumption. Furthermore, the identical data processing circuits in both modules may be a source of CMFs which originate from both hardware defects and design weaknesses.

Converting the ALU to negative-logic is suprisingly simple. Both the sum and the carry circuits of a full adder are their own self-duals[Take80]. Thus, *no* modification is required for that part of the circuit. In addition to the arithmetic sum, the RISC ALU also generates the logical AND, OR, and XOR (exclusive OR) of its inputs. The actual output of the ALU is determined by a 4-to-1 multiplexer. By interchanging two of the control lines to that multiplexer, the postive-logic OR can be selected by the AND instruction and the postive-logic AND can be selected by the OR instruction. The only function that requires modification is the XOR. For this particular case, the simplest solution is to connect an inverter to the output of the postive-logic XOR function. Since the performance of the ALU is determined by the worst-case addition time, the delay of the extra inverter in the XOR circuit does not affect system peformance.

One of the necessary modifications to the shifter is the conversion of the shift amount decoder to accept negative-logic inputs. As discussed earlier, this modification is very simple. The only other problem is with logical shifts that shift in logic 0's to replace bits that are shifted out. In the n-module the "high" voltage level must be shifted in

instead of the "low" voltage as in the p-module. This change can be done with a small modification to the control circuitry that drives the shifter.

There is no simple modification to the program counter incrementer. However, the basic cell of this circuit is so small that a complete negative-logic replacement can be developed very quickly.

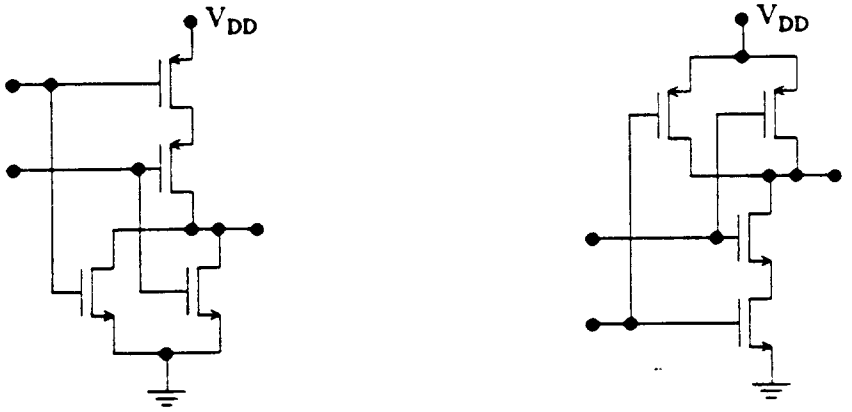
There are numerous ways in which the circuit modifications described above can help reduce CMFs. For example: (1) Shorts between data lines carrying complementary values usually result in both lines at the low voltage. Thus, both lines in the p-module change to logic 0 while the corresponding lines in the n-module that are similarly shorted change to logic 1. (2) Buses that fail to precharge in both modules will be interpreted as all zeroes in the p-module and all ones in the n-module. (3) If timing is not properly designed and there is insufficient time to drive the bus from one of its sources, different lines on the bus will be affected (the ones that must be discharged), and the failure will be detected. (4) The worst case delay for the ALU is determined by the carry propagation. If the ALU is modified as described above, the worst-case propagation for the two modules occurs for different inputs since the sum and carry circuits are identical while the ALU inputs in the p-module are always complements of the ALU inputs in the n-module. Hence, ALU failure, due to careless design of the timing or a particular fabrication run that yields especially slow devices, is unlikely to occur in both modules simultaneously.

Since most of the control circuits used in the n-module are identical to those used in the p-module, one might assume that there are many CMFs possible due to identical defects in those circuits in the two modules. This situation can be improved if the various decoders in the chip are modified as described in Subsection 6.1.3. Furthermore, many identical defects in the control circuitry lead to different effects on the data in the two modules. For example, if several bus sources (pull-downs) are selected at the same time (e.g., due to a fault in the opcode decoder), the resulting value on the bus will be the AND function of all the sources in the p-module and the OR of all the sources in the n-module.

6.1.2.2. CMOS Implementation

The *p*-channel FETs (PFETs), available in CMOS circuits, are turned on by the "low" voltage and turned off completely by the "high" voltage thereby providing the dual function of the *n*-channel FETs (NFETs). As a result, at first glance, it appears that with

CMOS technology it is relatively simple to convert the positive-logic module to negative-logic. Specifically, it can be shown that a positive-logic, ratioless CMOS circuit can be converted to a negative-logic circuit by replacing all NFETs with PFETs, replacing all PFETs with NFETs, connecting all V_{DD} lines to ground, and connecting all ground lines to V_{DD} (Fig. 6.6).



A. Positive Logic **B.** Negative Logic

Fig. 6.6: A CMOS NOR Gate

Unfortunately, due to the different mobilities of the majority carriers in NFETs and PFETs, these devices are not completely symmetrical. The W/L ratio of a PFET has to be approximately twice the W/L ratio of an NFET in order to achieve similar drive capability. Thus, in order to optimize performance when similar high-low and low-high propagation times are required, the PFETs used must be approximately twice the size of the corresponding NFETs. Since the gate capacitance is proportional to the size of the device, the delay caused by the PFETs due to their gate capacitance is larger than the delay caused by NFETs with equal drive capability.

Due to the advantages of NFETs, even in the CMOS RISC layout many more NFETs than PFETs are used. For example, NFETs are used in the shifter, which is basically an array of pass transistors. In the register file, the word lines, that select the register whose value drives the bus, do so by turning on a column of NFET pass transistors. In both these cases, PFET pass transistors and buses that are "precharged" low could be used. However, a design based on PFETs would be significantly larger and/or slower, as discussed above. Due to similar reasoning, NFETs are also used in the pull-down arrays of PLAs and decoders, while large PFETs are used for precharging lines

to the V_{DD}. Even with static gates used for random logic, the PFET pull-ups are approximately twice the size of the corresponding NFET pull-downs.

In order to maintain similar performance and module area, the p-module cannot be converted to an n-module by the simple procedure outlined earlier. The difficulties in achieving an efficient conversion are often similar to the difficulties encountered for NMOS circuits. Thus, similar solutions and considerations apply. On the other hand, the availability of PFETs can, at times, simplify the conversion. For example, in RISC, a large 32-input NOR gate is used to generate the Z flag, which is set when the result of an operation is zero. This gate is dynamic, with a single pull-up and a column of NFET pull-downs connected to a latch holding the result of the operation. In NMOS there is no simple way to convert this zero-detect circuit to negative-logic: a column of 32 inverters must be used to invert the output of the latch and drive the pull-downs of the large NOR. With CMOS, a large negative-logic NOR gate can be implemented using a single NFET pull-down and a column of PFET pull-ups connected to the output of the latch. If the performance of the circuit is critical, the PFETs will have to be larger than the corresponding NFET pull-downs in the p-module. However, the PFETs do not increase the power consumption, and the extra area of the larger PFETs is much smaller than the area required by a column of inverters.

6.1.3. Other Implementation Techniques for Reducing CMFs

As indicated in Subsection 6.1.2, not all the subcircuits in a VLSI chip are amenable to dual implementations. In those cases where dual implementations lead to unacceptable costs in terms of area and performance, other techniques for reducing CMFs are needed. The general "rule of thumb" is that the probability of CMFs can be reduced by increasing the "differences" between the modules. These differences may be introduced not only in the low-level circuits but also in the high-level module structure and in the fabrication process.

Modules that are likely to fail in different ways may be developed from the same specifications by two independent teams or, in the not too distant future, by two different "silicon compilers." [Aviz82] The main problem with this approach is, of course, increased design cost, which makes it impractical for most applications.

If the two modules are *not* on the same chip, chips fabricated by different companies

may be used. Platteter [Plat80] utilized this idea in constructing a fault-tolerant processor from three functionally identical microprocessors manufactured by different companies. Obviously, this can be done only with modules that are "popular" chips for which there are "second sources."

Even when it is not possible to convert a subcircuit to negative logic, it may still be possible to modify its structure without changing its function. We have previously discussed the modification of decoders for use with negative logic inputs. Another simple modification to the decoder is to change the order of output lines in the layout so that shorts between adjacent lines will affect logically different lines in the two modules. Similar restructuring can also be done in a PLA where the order of both the product term lines and the output lines may be changed.

If the register file decoder is restructured as suggested above, this also implies a "restructuring" of the register file itself. Different registers are next to each other and different registers are at the periphery of the register file where they may interact with other subcircuits and cause a module failure.

6.2. An Overview of Design and Implementation Tradeoffs

The complexity of a fault-tolerant VLSI multicomputer system implies that the designer of such a system is faced with a very large number of design choices. At the highest level, choices include the topology of interconnections between the nodes and the principles of the fault tolerance scheme to be used. Lower level choices include the design of the nodes and communication links. Implementation details such as power distribution, clock signal distribution, packaging, and cooling may be as important as the higher level design choices in determining the system's performance and reliability.

As mentioned earlier, design and implementation decisions must take into account the properties of the particular implementation technology, the environment in which the system will operate, and the characteristics of the intended applications. The effects of these factors on several key design and implementation issues in fault-tolerant VLSI multicomputers are discussed in this section.

8.2.1. Fault Tolerance at the Component Level

In the previous chapters we have discussed system-level fault tolerance techniques that can increase the reliability of a multicomputer system. An alternative approach to increasing the reliability of a multicomputer is to use fault tolerance techniques at the component level to increase the reliability of individual components. The choice of "how much" fault tolerance should be implemented at the component level and how much at the system level is of critical importance.

The system-level fault tolerance technique discussed in the previous chapters is only effective if the reliability of the individual nodes is high. If nodes fail "too often," the system will spend all of its time recovering from faults and never do useful work. Thus, some fault tolerance techniques (such as TMR) must be used at the *node* level to increase the reliability of the individual nodes. Similarly, if the communication links are subjected to "too much" noise, local techniques must be used to increase the reliability of the communication. For example, it is possible to use error correcting codes that allow the correct information to be recovered from a packet that has been "damaged" by noise.

The disadvantage of fault tolerance at the component level is that the required redundancy (overhead) is significantly higher than with system-level fault tolerance. If the components in the system do not "cooperate," they cannot share spare resources and use them to recover from faults. Instead, *each component* must contain spare resources to be used for recovery. If faults are rare, most of these spare resources are not needed, so the component-level scheme is inefficient. For example, assume that within the "mission time" of some multicomputer system, consisting of one thousand nodes, only one processor module in only one node is likely to fail. If fault tolerance is implemented at the component level, TMR can be used in each node. Thus, approximately two thirds of the system hardware is "wasted" for fault tolerance. On the other hand, if fault tolerance is implemented with the system-level scheme described in the previous chapters, duplication and comparison is used in each node for error detection and the system must contain one spare node that will be able to take over the tasks of the failed node. With this system-level scheme, only about half the hardware is dedicated to fault tolerance.

Given the inherent inefficiency of component-level fault tolerance schemes (no sharing of spare resources), they should only be used when necessary. The use of such schemes is necessary if the components are not sufficiently reliable or the particular

application has special requirements, such as instantaneous recovery (see Chapter 5), that cannot be met by a system-level scheme. Furthermore, even if some component-level fault-tolerance scheme is necessary, it is, in general, inefficient to attempt to achieve the reliability requirements of the system by relying *solely* on such scheme. Instead, the component-level scheme should be used only to increase reliability of the components to the point that it is possible to implement a system level scheme while meeting the other requirements of the system.

6.2.2. The Interconnection Topology

The interconnection topology of the system is a major factor in determining both its performance and its reliability. Ideally, the system would be completely connected so that there would be a communication link between every pair of nodes. This would minimize communication delays and maximize reliability since the ability of any two nodes to communicate would not be dependent on the correct operation of any other nodes. Unfortunately, it is not feasible to implement a large fully connected system due to the number of links required as well as the number of communication ports in each node. For example, a fully connected system with 100 nodes requires 4950 links and each node must have 99 communication ports.

The number of communication links per node is of critical importance in determining the interconnection topology. In order to minimize the complexity of the nodes as well as the difficulties of interconnecting the nodes (packaging), the number of ports per nodes should be small. Since the nodes are implemented with a small number of VLSI chips, the technological limitations on the number of pins per chip also limit the number of ports. The limitation on the power that can be dissipated on a chip implies a limit on the total bandwidth for transmitting information from the chip. Thus, even if the pin limitation is ignored, there is a tradeoff between a small number of high-bandwidth ports and a large number of low-bandwidth ports. Given this tradeoff, simulation studies have shown that the best performance can be achieved with between three and five ports per node [Fuji83, Séqu83].

In order to maximize performance and reliability, the diameter of the interconnection topology must be minimized while the connectivity must be maximized. A small diameter leads to low communication delays while large connectivity implies that a large number of nodes or links may fail before the system is partitioned into two disconnected networks.

It is also desirable for the topology to require that all the nodes have the *same* number of ports so that only one type of node has to be designed and implemented. Another desirable feature of the topology is the ability to use simple "algorithmic" routing rather than rely on table-driven routing that requires extensive hardware and software support [Prad82].

There are many different classes of topologies that achieve near-optimal diameter and connectivity under the above constraints. Rather than discuss them all, we will mention one example. Pradhan [Prad83] has developed a class of topologies with the following characteristics: (1) the number of ports per node is r, (2) the number of nodes in the system is $(r-1)^m$ with $m \geq 3$, (3) the diameter of the system is $2 \times m - 1$, (4) the connectivity of the system is $r-1$. As an example, a system with 1024 nodes, each with 5 communication ports, has a diameter of 9 and a connectivity of 4. With these topologies simple algorithmic routing is possible not only when all the nodes are operational, but also after some of the nodes have failed.

6.2.3. System Timing and Communication

The physical size of a multicomputer system with hundreds or thousands of nodes and the high clock-rate at which it operates preclude the implementation of the system as one synchronous unit with a single clock. Not only will it be impossible to distribute the clock without significant clock skews, but the failure of this single clock may result in the failure of the entire system. Higher reliability can be achieved using a large number of independent clocks rather than one clock. In particular, each node or collection of a small number of nodes can operate with their own crystal-controlled clock. Thus, the nodes operate asynchronously.

Since the nodes are asynchronous, there is a non-zero probability of errors in packet transmission due to *synchronization failures* even if there is no noise on the link [Seit80]. However, the INMOS Corporation claims that in their implementation of the Transputer they have achieved a rate of synchronization failures of 0.1 per billion part hours through the use of appropriate circuitry and communication protocols [INMO84]. Since this failure rate is at least three orders of magnitude lower than the failure rate of VLSI chips, it is not expected to be a significant factor in choosing a fault tolerance scheme.

6.2.4. Power Distribution

The distribution of power and ground throughout the system is a difficult problem, even if the issue of reliability is ignored. Power and ground must be routed to every board in the system, to every chip within each board, and to every gate within each chip. The noise on these lines must be minimized and care must be take to ensure that the correct voltage levels are available throughout the system.

Unfortunately, just like every other part of the system, power supplies and power lines can fail at all levels of the system. The complete failure of the power delivery system is often easy to detect but difficult to tolerate. If the system has only one power supply and that power supply shorts, the entire system will stop operating. With such a catastrophic failure there is no danger of accepting incorrect results as correct. However, recovery is impossible. To combat the problem of catastrophic power supply failures, most fault-tolerant systems employ multiple power supplies[Katz82]. Using special circuitry on each board to "mix" the outputs of multiple power supplies, it is possible to ensure that the board will continue to operate despite the failure of one of the supplies[Katz82].

A technique similar to the above may be possible at the chip level. Specifically, power lines from multiple supplies may be routed to each chip and "mixed" internally. The disadvantages of this scheme are the resulting increased complexity of the boards, the chip area devoted to this "mixing" (which will have to be very large due to the current levels involved), and the extra pins on each chip devoted to multiple supplies. Thus, in most systems this scheme is impractical.

If the system is to tolerate the failure of multiple nodes due to problems with the power supply, special care must be taken in the construction of the system so that the failure of a set of nodes that depend on a particular supply "route" will not partition the system. This must also be taken into account in allocating processes to nodes and in the error recovery and reconfiguration schemes.

Failures of the power supply lines may also have less catastrophic effects that are more difficult to detect but easier to tolerate than the effects discussed above. For example, despite a break in the power supply line to a particular module inside the chip (such as the shifter in a microprocessor chip), the chip may continue to perform many of its tasks correctly yet occasionally produce incorrect outputs.

In general, any error detection scheme must be able to detect the effects of faults in the power supplies and their interconnections at all level of the system. The error detection scheme discussed and Chapter 3, Chapter 4, and Section 6.1 fares quite well in this respect: (1) The outputs from the comparator (two-rail code checker) are supposed to be 01 or 10. If the power to either the entire node or just to the comparator is disconnected, the output will be 00 and the error will be detected immediately by a neighbor. (2) Since dual implementation is used for the two functional modules within each node, if power is disconnected to both modules (or the same submodules within each module) their outputs are identical (zero volts) rather than complementary, and the comparator detects the error. (3) Since duplication and comparison is used, there is no need to analyze in detail all the possible effects of breaks and shorts in the power supply lines internal to the chip containing the functional modules—error detection is guaranteed even if the power-supply-line fault has the effect of multiple faults on logic lines and causes some arbitrary submodule within the module to produce incorrect results.

References

Aviz82. A. Avizienis, "Design Diversity - The Challenge of the Eighties," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 44-45 (June 1982).

Cart68. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *IFIPS Proceedings*, Edinburgh, Scotland, pp. 878-883 (August 1968).

Fuji83. R. M. Fujimoto, "VLSI Communication Components for Multicomputer Networks," CS Division Report No. UCB/CSD 83/136, University of California, Berkeley, CA (1983).

INMO84. INMOS, *IMS T424 Transputer Reference Manual*, November 1984.

Katz82. J. A. Katzman, "The Tandem 16: A Fault-Tolerant Computing System," pp. 470-480 in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill (1982).

Mead80. C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).

Patt82. D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer* **15**(9) pp. 8-21

(September 1982).

Plat80. D. G. Platteter, "Transparent Protection of Untestable LSI Microprocessors," *10th Fault-Tolerant Computing Symposium*, Kyoto, Japan, pp. 345-347 (October 1980).

Prad82. D. K. Pradhan and S. M. Reddy, "A Fault-Tolerant Communication Architecture for Distributed Systems," *IEEE Transactions on Computers* **C-31**(9) pp. 863-870 (September 1982).

Prad83. D. K. Pradhan, "Fault-Tolerant Architectures for Multiprocessors and VLSI Systems," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 436-441 (June 1983).

Sedm80. R. M. Sedmak and H. L. Liebergot, "Fault Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," *IEEE Transactions on Computers* **C-29**(6) pp. 492-500 (June 1980).

Seit80. C. L. Seitz, "System Timing," pp. 218-262 in *Introduction to VLSI Systems*, ed. C. Mead and L. Conway, Addison-Wesley (1980).

Séqu83. C. H. Séquin and R. M. Fujimoto, "X-Tree and Y-Components," pp. 299-326 in *VLSI Architecture*, ed. B. Randell and P.C. Treleaven, Prentice Hall, Englewood Cliffs, NJ (1983).

Sher84. R. W. Sherburne, M. G. H. Katevenis, D. A. Patterson, and C. H. Séquin, "A 32-Bit NMOS Microprocessor with a Large Register File," *IEEE Journal of Solid-State Circuits* **SC-19**(5) pp. 682-689 (October 1984).

Taka83. M. Takada, "Two-Phase CMOS RISC Design," Unpublished Report, CS Division, University of California, Berkeley, CA (October 1983).

Take80. K. Takeda and Y. Tohma, "Logic Design of Fault-Tolerant Arithmetic Units Based on the Data Complementation Strategy," *10th Fault-Tolerant Computing Symposium*, Kyoto, Japan, pp. 348-350 (October 1980).

Tami84. Y. Tamir and C. H. Séquin, "Reducing Common Mode Failures in Duplicate Modules," *International Conference on Computer Design*, Port Chester, NY, pp. 302-307 (October 1984).

Wake78. J. F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*, Elsevier North-Holland (1978).

Chapter Seven
Summary and Conclusions

The current technology used for implementing high-end computing systems is fast approaching fundamental physical constraints, such as the speed of light, that limit the speed at which computations can proceed. The performance requirements of future high-end computers will only be met by systems that facilitate the exploitation of the parallelism inherent in the algorithms that they execute. One such system is a *multicomputer* composed of a large number of independent computers (computation nodes) interconnected by high-speed dedicated links. With a multicomputer high performance is achieved by dividing each task into a large number of subtasks that are executed simultaneously on different nodes.

Due to recent advances in VLSI technology, two important types of chips are, or will soon be, commercially available: (1) general-purpose processors whose performance exceeds that of current mini-computers, and (2) sophisticated communication processors that can efficiently support high-bandwidth communication in point-to-point networks. With these chips, the implementation of a multicomputer consisting of hundreds or thousands of VLSI computation nodes is technically and economically feasible.

Some of the important applications of high-end computers, such as large circuit simulation, weather forecasting, and aeronautical design, require continuous correct operation of the system for many hours (or even days). Due to the rate of failure of VLSI chips, this requirement cannot be met in a system that operates correctly only if all of its chips are fault free. The reliability requirements of a multicomputer can only be achieved with *fault tolerance* techniques that prevent component failure from leading to system failure. Compared to other architectures, a multicomputer is particularly well-suited to fault tolerance techniques since it does not contain any single component (such as a common memory or bus) whose performance is critical to the operation of the system.

The effective implementation of highly reliable systems requires the use of a combination of hardware and software techniques, carefully tailored to the characteristics of the implementation technology and the intended applications. In this dissertation we have investigated the use of fault tolerance techniques to increase the reliability of VLSI

multicomputers. Many aspects of the design and implementation of the system were considered: its basic architecture, details regarding the VLSI layout of key circuits, and high-level protocols that can use this hardware effectively to achieve high reliability with only a small penalty in performance. The techniques presented were developed in the context of the entire system, taking into account all of the above-mentioned aspects of the proposed general approach to implementing fault tolerance in the multicomputer.

A fault-tolerant system must be able to identify erroneous information produced by faulty hardware. The detection of an error implies that the state of the system has been corrupted. In order to recover from an error and resume correct operation a valid system state must be restored. The proposed general approach to implementing fault tolerance in a multicomputer involves a combination of hardware that performs error detection and system-level protocols that handle error recovery and fault treatment.

It is shown that a very high probability of error detection can be achieved with self-checking nodes that are implemented using duplication and comparison. This approach seems wasteful since it more than doubles the required hardware. However, this cost is justified by the resulting low design complexity, high fault coverage, and ability to handle transient faults effectively.

With duplication and comparison, all errors caused by hardware faults are detected as long as two requirements are met: (1) the comparator is fault-free and (2) the two modules never produce identical incorrect outputs. A comparator failure may mask a mismatch between the outputs of the two functional modules so that the rest of the system may accept erroneous outputs from the node as correct. It is imperative that faults in the comparator be detected soon after they occur so that the system can be informed that the node has lost its self-checking capability. This requirement is fulfilled by using a *self-testing* comparator that signals its own faults during normal operation. Based on a new fault model for PLAs, it was shown that with both NMOS and CMOS technologies a PLA can be used to implement such a comparator.

Unfortunately, it is not possible to guarantee that the two modules that perform identical functions do not fail simultaneously in exactly the same way and produce identical incorrect results. Such *common mode failures* may occur as a result of environmental factors, common design weaknesses, as well as unrelated faults that just happen to cause the same incorrect results to be produced. Practical technique were

developed for implementing pairs of VLSI modules that perform identical functions but are less susceptible to common mode failures than pairs of identical circuits. Based on examples of NMOS and CMOS circuits, it was shown that the likelyhood of common mode failures can be reduced, at a relatively low cost, using a combination of techniques carefully tailored to the functional and physical characteristics of the different types of circuits in a typical VLSI chip.

An error recovery scheme for use in a multicomputer executing non-interactive applications has been presented. The scheme is based on periodically checkpointing of the entire system state and rolling back to the last checkpoint when an error is detected. Since the nodes in a multicomputer operate asynchronously, special protocols are required to ensure that the saved states of all of the nodes in the system are consistent with each other. The proposed scheme involves first "freezing" the entire system in a consistent state and then saving the frozen state of each node individually. No restrictions are placed on the actions of the application tasks, and the communication protocols used during normal computation are simpler than those required by most other schemes. The scheme includes efficient handling of transient faults, input/output operations, and disk failures. For a "typical" multicomputer system with one thousand nodes, the performance degradation due to periodic checkpointing is expected to be a few percent.

Although this dissertation does not provide a complete detailed design of a high-performance fault-tolerant multicomputer, it does include a discussion of some practical design and implementation tradeoffs. A particular system must be tailored to the details of the intended applications, the operating environment, and the implementation technology. Based on this dissertation, a multicomputer implementation that follows the general techniques presented and uses the proposed self-checking nodes and error recovery scheme can provide a general-purpose, high-performance computing environment in which the fault tolerance features are completely transparent to the user.