Preprint from Proceedings of the Summer School on VLSI Tools and Applications,
Switzerland, July 1986, Kluwer Acadmic Publishers
W. Fichtner and M. Morf editors.

FAULT TOLERANT VLSI MULTICOMPUTERS

*Carlo H. Sêquin and Yuval Tamir**
Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

An approach is presented to increasing the reliability of future high-end systems beyond what is possible with technological solutions alone. The system consists of computation nodes and communication nodes, interconnected by high-speed dedicated links. These components are relied upon to detect errors while system level protocols are used for error recovery and reconfiguration. The use of duplication and matching for implementing the self-checking nodes allows us to restrict a detailed analysis of the impact of all possible faults to the comparator, a circuit that can be implemented in a relatively straight-forward way in NMOS or CMOS technology.

1. INTRODUCTION

Certain computational problems such as weather forecasting, simulations of complex systems, or design optimizations, exceed the capabilities of current computers. They require a substantial increase in compute power and, since such computations may run for an extended amount of time, improved systems reliability. There are fundamental limitations to the gains in reliability and performance that can be obtained from advancing technology alone. A more important contribution will have to come from organizational improvements in such computing systems: performance can be enhanced by exploiting parallelism, while the limits on reliability can be overcome using fault tolerance techniques.

If the computational problem can be subdivided into a sufficient number of simultaneously executing tasks, then this inherent parallelism of the problem can be used to achieve high performance by a system that comprises many computational nodes. A possible systems architecture that is compatible with the constraints of VLSI interconnects these computation nodes using high-speed dedicated links, and *communication nodes* which provide hardware support for communication functions such as message routing.[23] Computation nodes my consist of a single processor chip and several memory chips surrounded with the associated glue

*Now with the Computer Science Dept., University of California, Los Angeles, CA 90024.

logic, forming a powerful self-contained computer. The communication node has several ports through which it is connected to computation nodes and other communication nodes. Such a system is called a *multicomputer*. Ideally, the two types of nodes and the links between them are building blocks that can be used to construct multicomputers with a wide range of organizations and performance.

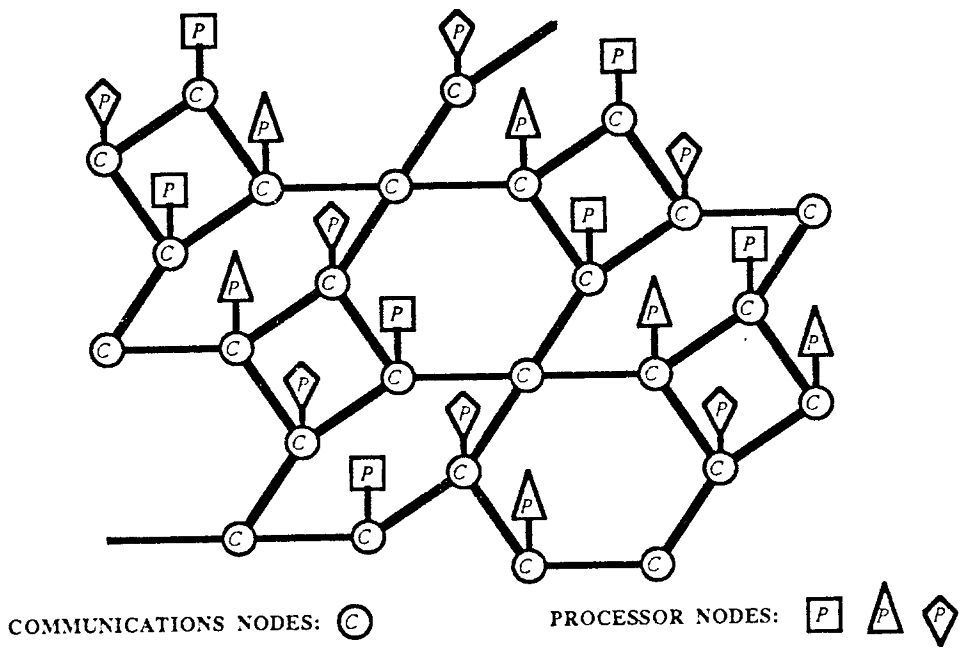

Figure 1: *Conceptual View of a Multicomputer.*

When the behavior of a system deviates from its specification at the interface with the "outside" world, we say a *system failure* [1] has occurred. System failure is often the result of a failure of one of its components. However the failure of a component does not necessarily imply that a system failure must occur. The system is *fault-tolerant* if it can continue operating correctly despite the failure of some of its components. Various techniques can be employed to provide *fault tolerance* at the different levels of the system hierarchy.

A multicomputer is particularly well suited for reliability enhancement using fault tolerance techniques since it is naturally divided into fairly independent modules of substantial "intelligence" — the above mentioned nodes. Fault-free components can adjust their behavior to the changes in faulty components and continue their operation in such a way that the overall output of the system remains correct despite the occurrence of a fault.

The design of a VLSI chip or of a multicomputer system is in itself a very hard task. Adding the extra demands of fault tolerance may just make this task unmanageable, unless we simplify the task by using principles of regularity and repetition. Using the multicomputer system as an example, it will be demonstrated how the concerns of fault tolerance can be concentrated on a few critical components, and how, by a suitable modular approach, the whole system can become fault tolerant, without undue penalty to either system design time or system performance.

2. FAULT TOLERANCE

The reliability of any system can be enhanced by increasing the reliability of its components through *fault prevention*[1] techniques, such as specialized design methodologies, stringent quality control, and extensive validation and testing. These techniques typically result in more complex designs,[8] greater cost, and lower performance.[20] Furthermore, the effectiveness of these techniques is limited by our inability to exhaustively test complex VLSI chips.[19]

The reliability of components can also be increased by employing *fault tolerance* techniques at the component level. These techniques attempt to ensure that each component will continue to perform according to its specifications despite the failure of its subcomponents. Unfortunately, no component can tolerate an unbounded number of faults. Thus, the system must be able to handle component failure. The contamination of the system by incorrect output from a faulty component can be prevented only if, at some stage, other system components find out about the failure of the component and physically or logically isolate it from the rest of the system.

If a node fails due to a transient fault, it need not be removed from the system, but rather should be reset to a proper state, and then continue to be a useful part of the system. If the failure is detected by a neighboring node, then this node must have the authority to initiate some action that might eventually lead to a resetting of the node. However, the same authority also gives a *failed* node the potential to invoke the resetting of an operational neighbor, so that a single node failure could result in a total system failure. To prevent this undesirable situation, each node must be responsible for its own reset. Hence the node should include a mechanism to detect its own *erroneous states* and to initiate a reset.

System level fault tolerance techniques need not rely on the components to report faults themselves. Instead, system level protocols could be used for detection and recovery from the component failure. For example, each task may be performed in parallel on three nodes and a "majority vote" taken on the results. Such a system with triple modular redundancy[31,33] can continue to produce the correct output even if one of the nodes fails. While the scheme does not make any assumptions about the nature of the individual subcomponents, it requires the system level protocols to ensure that the parallel task execute on different nodes and that the messages between themselves and the initiation node travel via independent paths. Hence this method leads to very high overhead in the use of the computation nodes as well as in message traffic. Additional problems concern locating failed components and effective handling of transient faults.

Many of the deficiencies of fault tolerance techniques that rely only on hardware or only on system-level protocols can be overcome by using a combination of hardware error detection in self-checking components and system-level protocols that perform error recovery and fault treatment. Errors caused by faults in the communication links are detected through the use of error-detecting codes. All nodes are self-checking and signal to the rest of the system when their output is incorrect. In addition, failed nodes attempt to reset themselves and reestablish a sane state. The immediate neighbors are informed whenever a node fails. If the node does not reset itself or fails too often, the neighbors can logically remove it from the system by refusing to communicate with it. The diagnostic status information is distributed throughout the system so that, eventually, no fault free node will attempt to use the faulty component.

On top of the self-checking hardware there is a low-overhead, application-transparent, distributed error recovery scheme. It involves periodic checkpointing of the entire system state and rolling back to the last checkpoint when an error is detected (Section 7).

3. SELF-CHECKING NODES

For all likely faults, a *self-checking* component must either produce the "correct" output (according to its specifications) or somehow indicate that its output is incorrect. A component that satisfies this requirement is said to be *fault secure.*[22] If the component does not produce an error indication immediately following the first fault, it is possible for several faults to exist in the component simultaneously without any indication to the rest of the system. Even if the component is fault secure with respect to any single fault, several faults together may lead to the failure of the self-check mechanism and, eventually, to incorrect output from the component. In order to prevent this situation, the component must also be *self-testing.*[22] In the presence of one or more faults, a self-testing component is guaranteed to produce an error indication before additional faults can occur that may lead to the failure of the self-check mechanism. Components which are fault-secure as well as self-testing are said to be *totally self-checking*[22] (*TSC*).

Error detecting or correcting codes can be used to implement *TSC* nodes. Redundant information is carried by busses, memories, and registers in order to detect (and possibly correct) errors.[22] Unfortunately, different coding schemes must be used for different parts of the node. This increases the complexity of the design task and makes design verification and testing more difficult. As a result, failure modes that are harder to predict and "tolerate" are more likely to occur.

An alternative is to construct the *TSC* computation or communication node using two identical, *independent* modules, each performing the function of the node. Inputs from neighbor nodes are fed to both modules. Except for the, hopefully nearly-impossible, case where both modules produce identical *incorrect* output (Section 6), if the modules operate synchronously, errors can be detected by simple comparison of the outputs of the modules. The comparator that performs this function is part of the node, and its output is connected to neighboring nodes through dedicated wires. The output from one of the two modules is the "functional" output from the node (Fig. 2). A "no-match" signal from the comparator is used locally as a reset signal and is also sent to all neighbors as a failure indicator. Similar failure indicators from the neighbors cause an interrupt and invoke system-level routines that handle the node failure.

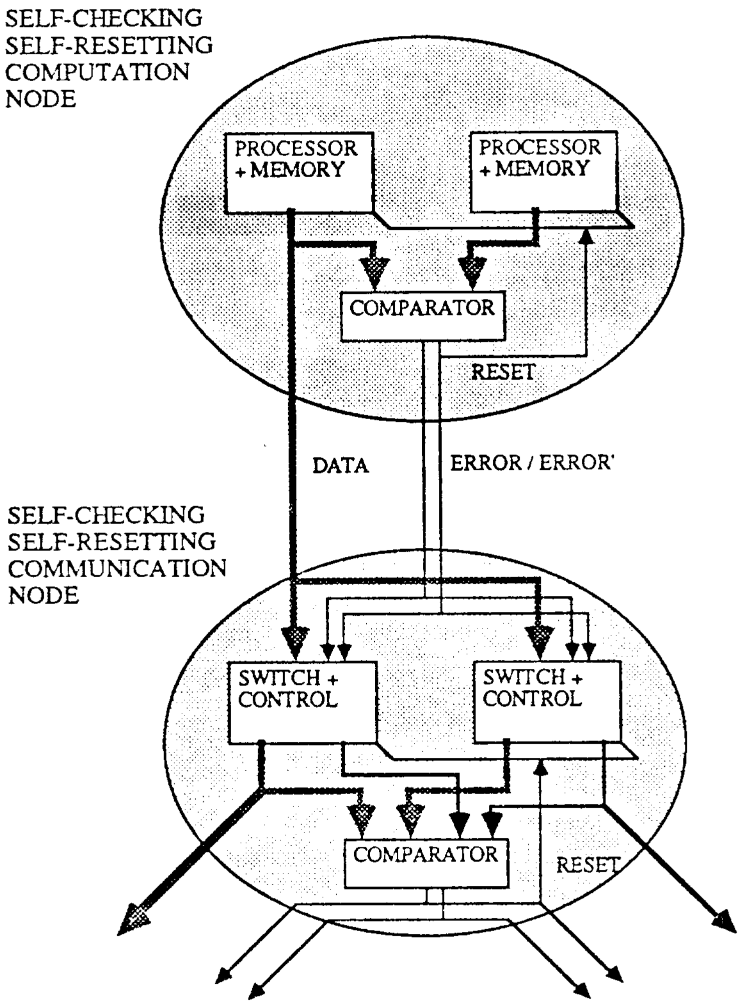

Figure 2: *Self-Checking Nodes for Multicomputers*

Implementing the *TSC* property in a component using duplication and comparison may appear wasteful since it more than doubles the required hardware. However, this scheme becomes more attractive when issues such as design complexity, fault coverage, reliability prediction, and the ability to recover from transient faults are taken into account.

Traditional fault models are not adequate for VLSI.[10,30] As a result, low-cost error detection schemes, that are based on these models, may no longer be sufficient. With duplication and comparison, errors are detected as long as the comparator remains functional and the two modules produce different outputs the first time one or both of them fail. Since a faulty comparator can mask faulty functional modules, faults in the comparator must not go undetected, *i.e.*, the comparator must be self-testing. Thus a detailed analysis of the effects of all likely faults on the comparator is required.

4. DEFECTS AND FAULTS IN VLSI

The design of self-checking circuits requires an understanding of the physical defects that commonly occur in VLSI and of the resulting logical faults. In the past the stuck-at fault model has been widely used to model, at the logical level, the effects of physical defects in circuits. This model does not cover many of the possible defects in VLSI.[7,10,30] The fabrication flaws and physical processes that can cause malfunction of NMOS and CMOS VLSI circuits are summarized in this section.

VLSI chip failures may be caused by design or fabrication flaws, may be due entirely to environmental factors, or are the end result of a degenerative process invoked by operational and environmental stresses but often attributable to design or manufacturing flaws.[9,22] Fabrication defects in MOS chips consist mainly of shorts and opens in each interconnection level, (metallization, diffusion, and poly-silicon), shorts between different levels, and large imperfections such as scratches across the chip.[10] 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.[9] During the operation of the chip, faults may be caused by electromigration, corrosion, electrical breakdown of oxide, cracks due to thermal expansion, power supply fluctuation, and ionizing or electromagnetic radiation.[9]

At the logical level, most of the faults can be represented in a circuit model consisting of switches, loads (for NMOS), and interconnection lines that directly correspond to the transistors and interconnections in the actual circuit.[10] Most of the physical defects, such as opens and shorts, can be represented in this model in an obvious way.[7] A "switch" may be permanently on or permanently off, corresponding to a gate input stuck-at-1 or stuck-at-0, respectively. Shorted NMOS loads (pullups) are equivalent to an output line s-a-1. Disconnected gate inputs are usually equivalent to s-a-0 or s-a-1 faults.

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 gates connected to its output may "interpret" it 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 some of the gates as logic 0, we call it a *weak 1* fault. Conversely, if the line is supposed to be a logic 0 but is interpreted by some of the gates as logic 1, we call it a *weak 0* fault. A single physical defect, resulting in a single weak 0 or weak 1 fault, has the same effect as multiple s-a-1 or s-a-0 faults, respectively.

In CMOS, a transistor which is permanently off or a break in a line can result in a high impedance state where the output of a combinational logic gate is dependent on the previous output rather than the current input.[30] Such a fault (called a *stuck-open* fault) may escape detection even if all possible input vectors are used to test the circuit.[30]

5. SELF-TESTING COMPARATORS IN VLSI

The duplication and matching scheme relies entirely on a self-testing comparator to detect faults in the functional modules. Implementing such a comparator requires knowledge of how different faults will affect the circuit. Fortunately, a comparator is a simple circuit that can be implemented with a regular structure and is therefore amenable to thorough analysis. Hence, we can have confidence in our ability to predict the likely physical defects, develop a valid fault model, and prove that the implementation we propose is indeed self-testing.

We assume that physical defects in the node occur one at a time. A fault that is the result of a single physical defect is called a *single fault*. It is assumed that there is a negligible probability that the time interval between the occurrence of successive single defects in the comparator or between a single defect in the comparator and an arbitrary collection of defects in the functional modules, is less then some value T. In order to ensure that faults in the comparator will not mask future faults in the functional units, during normal operation, the comparator must "test itself" for any single fault in less than time T.

5.1. Single Stuck-At Faults

As a first step to constructing a comparator which is self-testing with respect to *any* single fault, we will discuss the implementation of a comparator which is self-testing with respect to any single *stuck-at* fault.

In this context "two-rail" codes prove useful. They consist of all words (bit vectors) such that a specified half of the word is the complement of the other half. If the output of one of the modules in a self-checking node is complemented, a two-rail code checker can serve as a "comparator" that checks the validity of the output (Fig. 3). Such a code checker, which is self-testing with respect to any single stuck-at fault, can be implemented as a two level NOR-NOR PLA (Fig. 4).[6,26,32] The output from the checker is a two-bit two-rail code that is 01 or 10 (*code output*) if the input is a two-rail code word (*code input*), and 00 or 11 (*noncode output*) otherwise (*noncode input*). It can be shown that if any single stuck-at fault exists in the checker, there is a two-rail code input word that results in a 00 or 11 output, thereby "detecting" the fault.[32]

The requirement that the checker must be self-testing with respect to any single stuck-at fault poses severe constraints on its implementation. It can be

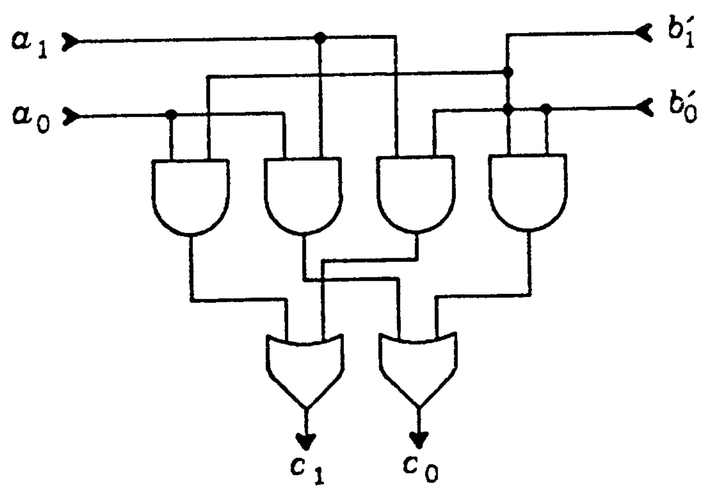

Figure 3: *Self-Testing Two-Rail Code Checker*

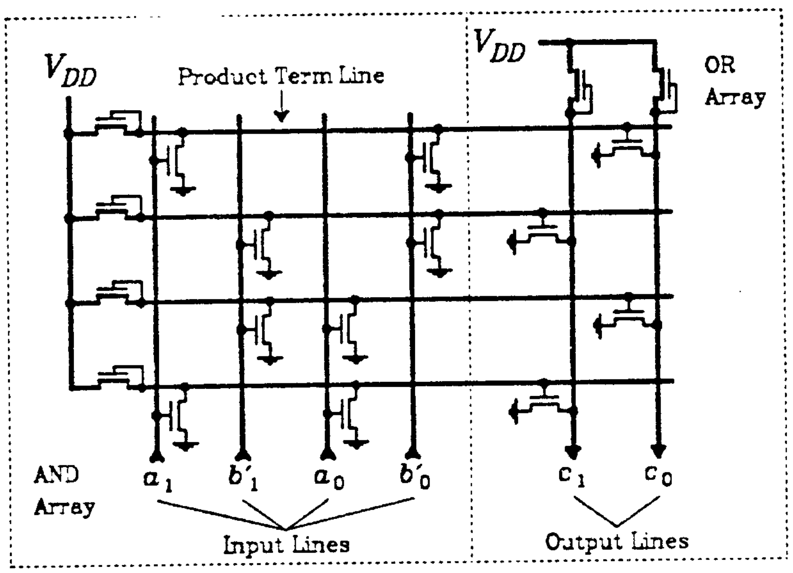

Figure 4: *NMOS Implementation of Code Checker*

shown that *any* two level AND-OR (or NOR-NOR) implementation for an input of $2n$ bits (n bits from each module) must use 2^n product terms, one for each code input.[26,29] If the output from each module is, say, 16 bits, this implementation is impractical since it requires $2^{16} = 65536$ product terms. Furthermore, all possible (2^n) code words must appear at the checker's inputs for it to perform a complete self-test.

Several small self-testing two-rail code checkers can be used as "cells" for constructing a self-testing checker for a wide input word (Fig. 5).[12,22] While the self-testing property is preserved, the number of input patterns required for a complete self-test is dependent only on the size of the largest "cell".[12]

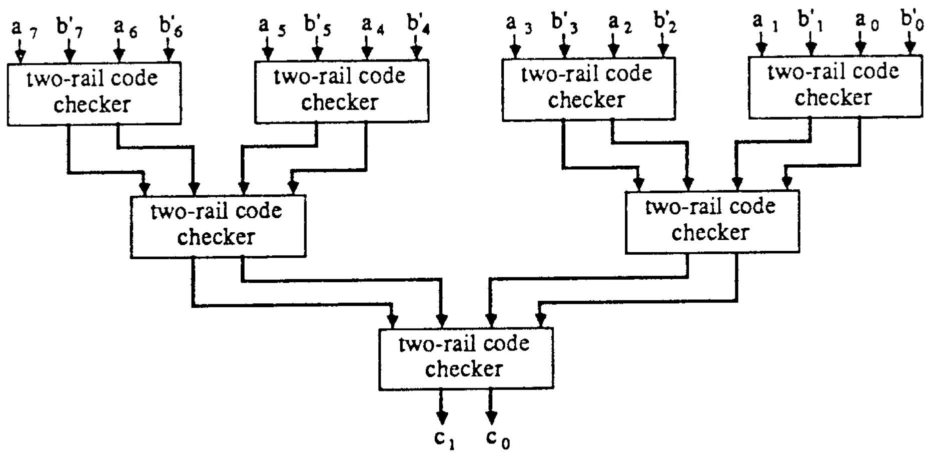
a_7 b'_7 a_6 b'_6
a_5 b'_5 a_4 b'_4
a_3 b'_3 a_2 b'_2
a_1 b'_1 a_0 b'_0
two-rail code checker
c_1 c_0

Figure 5: *A Self-Testing Two-Rail Code Checker Tree*

5.2. Other Single Faults

The faults that commonly occur in a MOS PLA are stuck-at faults, shorts between adjacent lines, breaks in lines, and contact faults that include missing or extra devices at crosspoints.[13,32] In addition, weak 0/1 faults can occur on the input or product term lines. Fortunately, it turns out that it is possible to implement a two-rail code checker that is self-testing with respect to any one of the aforementioned single faults. The implementation is a NOR-NOR MOS PLA which is laid out according to a few simple guidelines described in[26] and[29]. The rest of this section contains a partial informal "proof" of this claim; a more formal proof can be found elsewhere.[26,29] Faults in the input lines, product term lines, output lines, AND array crosspoints, and OR array crosspoints, are considered separately.

Any single stuck-at fault or short in the input lines will cause one or more 0's to change to 1's *or* one or more 1's to change to 0's (but not both) for some code input. It can be shown that such an error (called a *unidirectional error*[13]) on the input lines results in noncode output.[32] The effect of a break in an input line depends on its location. A break in the input line outside the AND array is equivalent to the line stuck-at-0 or stuck-at-1. A break in the middle of the AND array affects only some product terms. For an affected product term, if the break is equivalent to a stuck-at-1, the one code input that is supposed to select this product term won't, and a noncode output will result. If the break is equivalent to a stuck-at-0, there exists a code input that results in a noncode output since it selects two product term lines each of which is connected to a different output line.[29]

An extra device in the AND array is equivalent to the corresponding product term stuck-at-0. The code input that is supposed to select that product term results in a noncode output. If there is a missing device in the AND array, there exists a code input that produces a noncode output since it selects two product term lines, each of which is connected to a different output line.[29]

An extra device in the OR array means that one of the product terms is connected to both outputs. A missing device in the OR array is equivalent to the corresponding product term stuck-at-0. In either case, the code input that selects the relevant product term will result in a noncode output.

If the output lines are shorted, their values are equal and that is a noncode output. If one of the lines has a stuck-at fault, there exists a code input that causes the other line to have the same value, so the output is noncode. For some code input, a break in one of the output lines is equivalent to a stuck-at-1 or stuck-at-0 fault on that line.

A stuck-at-0 fault on a product term line will result in a noncode output if the input is the code word that is supposed to select that product term line. A stuck-at-1 fault on a product term line will result in a noncode output to any input that selects a product term line that is connected to the other output line. A break in a product term line is equivalent to a stuck-at fault on that line since each product term line is connected to only one output line. A short between two product term lines will result in a noncode output if the input selects either one of these lines.[29]

Product term lines are not susceptible to weak 0/1 faults since each product term line is connected to only one output line (fanout of one) so that a weak 0/1 fault is equivalent to a *single* stuck-at fault. Input lines have a fanout greater than one and are thus susceptible to weak 0/1 faults. A weak 1 fault on an input line is equivalent to one or more missing devices in the AND array. Each product term that is connected to a "missing device" will be selected by an input code word that also selects a product term line that is connected to the *other* output line.[29] Thus, a noncode output will result. A weak 0 fault on an input line is equivalent to one or more product term lines which are stuck-at-0. Any code input that is supposed to select one of these product terms will result in a noncode output.

In CMOS chips, PLAs are usually implemented in dynamic "pseudo NMOS".[30] All product term and output lines are precharged during every clock cycle before being selectively discharged according to the input. Therefore, no state is preserved from one cycle to the next, and the circuit is combinational despite any opens in the precharge or discharge paths.[29] Hence the PLA used in CMOS chips is only susceptible to the same faults as the traditional static PLA used in NMOS chips.

This analysis shows that for all single faults in our fault model, there exists a code input that results in a noncode output from the proposed two-rail code checker PLA. Thus, the checker is self-testing with respect to any likely single fault. Based on this result, it can be shown that the checker constructed as a tree of smaller self-testing checkers (Fig. 3) is also self-testing with respect to any likely single fault.[29]

6. IMPLEMENTATION ISSUES

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 Section 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. Thus we have to look at the causes of such *common mode failures* and at techniques for reducing their probability of occurrence. While it is not possible to entirely eliminated common mode failures, there are some practical implementation techniques for reducing the probability of these failures in the context of commonly used NMOS and CMOS circuits.

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, affecting both modules at the same time, triggering similar design weaknesses, and causing simultaneous identical failures of both modules. If the two modules to be matched are physical duplicates, then design weaknesses are a particularly worrisome source of CMFs. Any pattern-sensitive marginal performance is likely to trigger the same erroneous output in 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 rarely exercised.

Advancing VLSI technology will soon make it possible to implement an entire self-checking module, such as a computation or communication node in a multicomputer, on a single chip. This would provide nice logical building blocks[25] for the construction of powerful and reliable computer systems. Furthermore, such chips offer some advantages in production testing. 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. However, the danger of CMFs masking design flaws may prohibit this approach for the case where the two modules are copies of the same physical design.

Unfortunately, if the two functional modules (and the comparator) 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. Thus, having physical copies on the same chip enhances the possibility of CMFs to the point where it might defeat the overall purpose of fault tolerance. One must thus consider implementing different modules with the same desired behavior but with independent failure modes.

As noted in Section 3, one of the benefits of using duplication and comparison for self-checking subsystems is that relatively little extra design effort is required to implement the self-checking property. Creating two different implementations for every function clearly violates this goal. The question thus arises, how much extra effort is required to design and fabricate two modules for the given function whose implementations are sufficiently different to reduce the chance of CMFs to an insignificant level.

How the two modules should differ to achieve independent failure modes depends on the implementation technology. In the following, approaches are outlined that are suitable for NMOS and CMOS VLSI.

6.1. 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 $g(x) = \bar{f}(\bar{x})$ 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*.[27,29]

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[21] have suggested that the probability of CMFs in a self-checking functional block can be reduced by using *dual* modules rather than pairs of identical modules. To make use of dual modules, the inputs to the self-checking block 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[6] (see Fig. 6).

There are some immediate advantages to the use of dual modules. The difference in the two modules forces the use of different masks, and thus it is not possible that a mask defect gives rise to identical behavioral problems in both modules. Since one module is a *negative-logic* version of the other, electromagnetic pulses or noise on the power line will almost certainly produce different effects in the two modules. Finally, crosstalk problems within a module itself will typically appear at different times in the two modules because the sensitivity to electrical pickup at a particular circuit node is sensitive to the polarity of the voltage transition, and with dual circuits, the voltage transitions on corresponding lines in the two modules are in opposite directions.

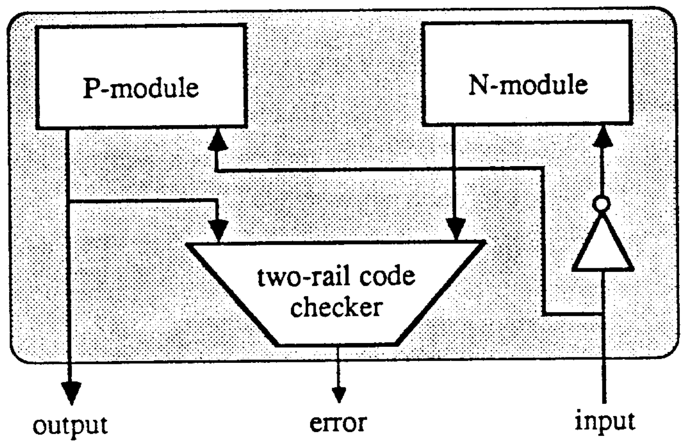

Figure 6: *Self-Checking Block Based on Dual Modules*

In SSI technology, the realization of a dual circuit is relatively straightforward; the negative-logic module can readily be derived from the positive-logic module by a simple one-to-one replacement of gates and flipflops by their negative-logic equivalents. In VLSI technology, the implementation of dual circuits 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 replacement of NOR gates with NAND gates is difficult. First, the different gates have different cell topologies and sizes, and the layout of the entire chip may have to be modified in order to accommodate the replacement gates. Second, the fan-in capability of the two gates may be different for example, in NMOS, it is possible to implement a NOR gate with a large number of inputs while NAND gates are limited to about four inputs. Finally, practical circuits are often not simply a collection of standard logic gates; they may contain transmission gates, precharged busses, register files, PLAs, dynamic logic subcircuits, etc. For some technologies, converting such circuits to negative-logic may require significantly more area and/or result in lower performance.[27,29]

Thus, a practical 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 to the outside world.

6.2. Partial Conversions

Standard NMOS circuits are fundamentally asymmetrical. The available devices are enhancement mode transistors (EFETs) and depletion mode transistors (DFETs). 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. These constraints prevent a simple conversion of many common NMOS subcircuits into negative-logic. A more practical approach is to selectively convert only some of the circuits and keep others unchanged. If this is done judiciously, the sensitivity of the system to CMFs can still be strongly reduced.[27]

Tamir proposes an approach[29] in which the N-module essentially stores and transfers all data in negated form, but where processing and control is done by positive-logic subcircuits. This approach avoids many problems with the conversion of control circuits: busses, multiplexers, and latches are not modified, and the transmission gates and pull-down transistors in them are controlled with signals of the same polarity in both modules.

Even though there are a lot of similarities between the two implementations of the modules, the probability of CMFs is greatly reduced. Shorts between data lines carrying complementary values usually pull both lines to the low voltage. Thus, both lines in the P-module change to logic 0 while similarly shorted lines in the N-module change to logic 1. Busses that fail to precharge in both modules will be interpreted as all zeroes in the P-module and all ones in the N-module. 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. The extra design effort with this approach is quite moderate.

CMOS technology offers switches of both polarities. Specifically, it can be shown that a positive-logic, ratioless CMOS circuit can be converted to a negative-logic circuit by simply 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}. It thus appears that it should be simple to convert a P-module to negative logic.

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. Therefore, a typical CMOS processor may employ many more NFETs than PFETs. In order to maintain similar performance and module area, the P-module cannot be converted to an N-module by simply complementing all FETs, and 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 simplify the conversion.[27]

6.3. Two Independent Implementations

Modules that are independently developed from the same specifications by two separate teams, are likely to fail in different ways. This approach is normally impractical because of the increased design costs. However, this situation may change for two reasons.

The generic modules needed to build VLSI multicomputers may become so popular, that different companies will develop the same product. Two modules fabricated by different companies can then be used to build self-checking nodes. Platteter[14] utilized this idea in constructing a fault-tolerant processor from three functionally identical microprocessors manufactured by different companies.

The other avenue to obtaining different implementations for a functional module[2] will come from the emergence of "silicon compilers". Before too long, design systems will get powerful enough to produce competitive macro modules or even whole chips in a fully automatic manner. The same set of specifications can then be run through two different compilers, or through the same compiler but with additional constraints that force two different implementations. At this stage most of the design effort will go into producing a full and unambiguous set of specifications. Once these specifications exist, obtaining different versions of the same module is only a matter of a few extra hours on a fast computer.

7. SYSTEM LEVEL PROTOCOLS

The *internal state* of a system is the ordered set of the *external states* (set of output values) of all of its components.[1] When a component fails, its external state is erroneous. Thus, *component failure* implies an erroneous internal system state, and an erroneous internal state can lead to *system failure*, i.e., incorrect output. Special measures, beyond simply detecting the error, must be taken in order to prevent system failure. In particular, in order to *recover* from the error, a *valid* internal system state must be restored, and the system must then be reconfigured so that it will not continue to use the faulty component. These actions require coordination between several (perhaps all) components. Hence, they involve system-level protocols.

7.1. Error Recovery

Most techniques for performing error recovery can be classified into two groups:[18] *Forward error recovery* techniques attempt to modify an erroneous system state so that it becomes a valid state. *Backward error recovery* techniques involve resetting (rolling back) the system to a previous valid state rather than trying to modify the current state.

Forward error recovery techniques are based on anticipating the types of errors that may occur and devising specific techniques for handling those errors. These techniques often involve special actions by the application program running on the system. On the other hand, backward error recovery techniques can cope with *unanticipated* errors. These techniques involve periodically recording the state of the system. When an erroneous state is detected, it is abandoned, and the system is reset to the previously recorded error-free state, called a *recovery point* or a *checkpoint*. The process of creating a recovery point is called *checkpointing*. 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.

Many error recovery schemes are designed for a system where all communication is over a common bus or Ethernet.[4,15] This allows the implementation of a "recording node" that keeps a record of all inter-node messages transmitted in the system[15] and facilitates the implementation of an efficient *atomic* operation that transmits a message to a "primary" process and to its "backup" that resides on another node.[4]

On a *multicomputer*, communication is point-to-point between nodes. In order to keep track of messages that are transmitted throughout the system, they must be explicitly forwarded to the "recording node"[15] or to the "backup node".[4] This requires extra delays in processing: Before any action can be taken which counts on a message having been transmitted reliably, an acknowledgement from the destination and the backup node must be received.

Barigazzi and Strigini propose an error recovery procedure that 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.[3] The critical feature of this procedure is that all interacting processes are checkpointed together, so that their checkpointed states can be guaranteed to be consistent with each other. Therefore, the *domino effect* that may require backing up to successively older states[18] cannot occur. As a result, it is sufficient to store only one "generation" of checkpoints.

With the recovery scheme described in [3] a large percentage of the memory is used for backups rather than for active processes. The resulting increased paging activity leads to increases in the average memory access time and the load on the communication links. This load is increased further by the required acknowledgements of all messages and the transmission of redundant bits for error detection. The communication protocols, which are used to assure that the message "send" and "receive" operations are atomic, require additional memory and processing resources for the kernel. Thus, performance is significantly reduced relative to an identical system where no error recovery is implemented.

7.2. A Low-Overhead Error Recovery Scheme for Multicomputers

As we described previously,[28,29] the technique of simultaneously checkpointing the state of all processes belonging to the same "task" can be taken a step further: simultaneous checkpointing of the complete state of all the user and system processes on the system. A new global checkpoint is periodically stored on disks. When an error is detected, diagnostic information is distributed throughout the system. Normal operation is resumed after all the operational nodes are set to a consistent system state using the last checkpoint.

Creating and saving a global checkpoint is expensive; 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 still small. With the proposed scheme, in a large multicomputer the expected time to establish a new checkpoint is less than one minute. 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 details of the proposed scheme are described elsewhere[28,29] and will not be repeated here. The technique consists of two major components: a scheme for saving a consistent global checkpoint of the entire system and a scheme for rolling back the system to a previously saved checkpoint once an error is detected. The technique relies heavily on the self-checking property of the nodes that ensures that faulty nodes are detected before erroneous information from them is allowed to spread throughout the system. As mentioned above, the technique is useful only for a system running non-interactive applications.

The scheme for saving a consistent global checkpoint is an adaptation of the standard two-phase commit protocol used for preserving consistency in distributed data base systems.[11] Initially, a designated node, say node 1, is assigned to serve as the *coordinator* for establishing global checkpoints. If the coordinator fails, all the other nodes are notified, and the next node, according to a 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.[3]

The checkpointing coordinator initiates checkpointing by stopping all local normal processes and notifying all of its neighbors that checkpointing is in progress. Each node in the system, in turn, repeats this process. Once all the neighbors are informed, each node begins to send its state to a node with a disk where the state is saved. When the entire state of the node is saved, the checkpointing coordinator is informed. After all the nodes have saved their states, the checkpointing coordinator directs the entire system to resume normal operation.

Since the nodes are self-checking, the failure of a node is detected by its neighbors. The neighbors "spread the word" throughout the system, indicating which node has failed and that recovery is in progress. When a node with disk storage finds out that recovery is in progress, it begins sending the previously saved state to all nodes that used it for checkpointing. Each node that receives a complete previous state informs the coordinator. After all the nodes have obtained their previous states, the checkpointing coordinator directs the entire system to resume normal operation.

It is possible to obtain a rough estimate of the overhead of the proposed system by making several specific assumptions about such system based on the intended application environment and on current and near-future technology. We base our assumptions on the use of the INMOS Transputer chip as the node.[34] We assume a system with 1,000 nodes, each with 256,000 bytes of memory, connected in a network with a *diameter* of 15. With communication link bandwidth of 1.5×10^6 bytes/second, checkpointing or recovery are expected to take less than 20 seconds.[28] If the system has a mean time between failures of 10 hours and a checkpoint is saved twice an hour, the total overhead for checkpointing and recovery will be approximately 3.7 percent.

7.3. Reconfiguration

Following error recovery, it is easiest to resume normal system operation if no changes are made in the operation of the nodes or the interconnection between them. This is possible if the error was caused by a node that failed due to a transient fault. Following recovery the node can resume its previous role in the system if it is capable of resetting itself to a "sane state" at the same time it informs the neighbors of the failure (see Section 2). However, if the node fails due to a permanent fault, the system must be capable of continuing normal operation without this node.

One of the requirements for the interconnection topology of the system is that the failure of any one node does not partition the system into two independent networks that cannot communicate. More generally, the maximum number of nodes that can fail without the possibility of partitioning the system, is a critical parameter in determining system reliability.

Nodes that fail due to permanent faults are effectively removed from the system. The algorithms used to route messages between nodes in the system must adapt to such changes in the topology of the system. If the system uses table-driven routing, the routing tables throughout the system must be updated following error recovery.[5,24] If the system uses "algorithmic" routing that does not require routing tables, the interconnection topology must allow such routing even after some of the nodes are removed.[17]

If a node fails due to a permanent fault, processes that were executing on it must be moved to a different node and continue to execute there. Thus, following recovery, messages from processes that were communicating with processes on the failed node must somehow be redirected to the new node. This ability to transparently *migrate* processes between nodes is a critical requirement for the operating system of a fault-tolerant multicomputer. Powell and Miller propose one possible scheme for such process migration in multicomputers.[16]

8. CONCLUSIONS

As the number of switching elements in a VLSI system starts to exceed a few hundred millions, the reliability and thus the fault-tolerance of the system must become a major concern. The design of a VLSI system is in itself a very hard task, and adding fault-tolerance may just make it unmanageable, unless we use the principles of regularity and repetition to simplify the task.

Using the example of a multicomputer system, consisting of hundreds or thousands of VLSI computation nodes interconnected by dedicated links, we have demonstrated how the concerns of fault-tolerance can be concentrated on a single critical component, and how, by a suitable modular approach, the whole system can become fault tolerant, without undue penalty to either system design time or system performance. The discussed scheme combines hardware that performs error detection with system-level protocols for error recovery and for fault treatment.

We have shown that a high probability of error detection can be achieved with self-checking nodes implemented using duplication and comparison. These nodes use two modules that perform identical functions but are not susceptible to simultaneous identical failures. The output of these modules is compared in a self-testing code checker that has been thoroughly analyzed for all likely defects in present-day VLSI circuits.

The proposed low-overhead, application-transparent error recovery scheme for the system 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. 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. A multicomputer system that follows the general principles outlined in this paper can provide a general-purpose, high-performance computing environment in which the fault tolerance features are completely transparent to the user.

Acknowledgements

This research was supported by the State of California MICRO program and the Defense Advance Research Projects Agency (DoD).

References

1. T. Anderson and P. A. Lee, "Fault Tolerance Terminology Proposals," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 29-33 (June 1982).
2. A. Avizienis, "Design Diversity - The Challenge of the Eighties," *12th Fault-Tolerant Computing Symposium*, Santa Monica, CA, pp. 44-45 (June 1982).
3. G. Barigazzi and L. Strigini, "Application-Transparent Setting of Recovery Points," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 48-55 (June 1983).
4. 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).
5. M. Bozyigit and Y. Paker, "A Topology Reconfiguration Mechanism for Distributed Computer Systems," *The Computer Journal* **25**(1), pp. 87-92 (February 1982).
6. W. C. Carter and P. R. Schneider, "Design of Dynamically Checked Computers," *IFIPS Proceedings*, Edinburgh, Scotland, pp. 878-883 (August 1968).
7. 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).

8. R. P. Davidson, M. L. Harrison, and R. L. Wadsack, "BELLMAC-32: A Testable 32 Bit Microprocessor," *1981 International Test Conference Proceedings*, Philadelphia, PA, pp. 15-20 (October 1981).

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

10. 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).

11. 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.

12. 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).

13. 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).

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

15. M. L. Powell and D. L. Presotto, "Publishing: A Reliable Broadcast Communication Mechanism," *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 100-109 (October 1983).

16. M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. 9th Symp. on Operating Systems Principles*, Bretton Woods, NH, pp. 110-119 (October 1983).

17. D. K. Pradhan, "Fault-Tolerant Architectures for Multiprocessors and VLSI Systems," *13th Fault-Tolerant Computing Symposium*, Milano, Italy, pp. 436-441 (June 1983).

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

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

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

21. 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).

22. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press (1982).

23. 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).

24. 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).

25. 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).

26. 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).

27. 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).

28. Y. Tamir and C. H. Séquin, "Error Recovery in Multicomputers Using Global Checkpoints," *13th International Conference on Parallel Processing*, Bellaire, MI, pp. 32-41 (August 1984).

29. Y. Tamir, "Fault Tolerance for VLSI Multicomputers," Ph.D. Dissertation, CS Division Report No. UCB/CSD 86/256, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA (August 1985).

30. 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).

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

32. 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).

33. 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).

34. C. Whitby-Strevens, "The Transputer," *12th Annual Symposium on Computer Architecture*, Boston, MA, pp. 292-300 (June 1985).